Screen space features for real-time hatching synthesis

Zoltán Lengyel[1], Tamás Umenhoffer[2], and László Szécsi[3]

[1] Budapest University of Technology and Economics, Department of Control Engineering and Information Technology
katzlengyel@gmail.com
[2] Budapest University of Technology and Economics, Department of Control Engineering and Information Technology
umenhoffer@iit.bme.hu
[3] Budapest University of Technology and Economics, Department of Control Engineering and Information Technology
szecsi@iit.bme.hu

Abstract. In this paper we propose a real time hatching synthesis algorithm. Hatching lines are placed evenly in screen space, their direction and bending are controlled by screen space features that can describe the underlying geometry well. We examined several features that can be easily obtained using common render packages, or can be computed from common geometry data. These include lighting coefficients, screen space depth or normal vectors, tone gradients, screen-projected projected normal vector directions, and the principal curvature directions. We examined several typical models to find the strength and weakness of each approach. We implemented the algorithm on the GPU using OpenGL and GLSL shaders, and achieved real time performance.

1 Introduction

Three-dimensional computer graphics techniques usually focus on simulating real world physics to produce images as close to reality as possible. These techniques belong to photorealistic rendering. In contrast, a wide range of techniques target non-photorealistic or illustrative image production, among them the synthesis of hand-drawn art. These usually mimic pen and ink illustrations, pencil drawings, watercolor or oil paintings.

In case of pen and ink illustration and pencil drawings, the lighting and shadows and the shape of the objects are represented with the density and orientation of thin hatch lines. Hatching-style rendering algorithms should mimic this line drawing process while processing 3D geometry taking into account that the artists work strictly in 2D. Thus, they should ensure that hatching is consistent in screen space, meaning stroke width or length does not depend on the distance of the drawn objects from the camera, and their individual screen-space appearance does not change during an animation. Hatch lines should be placed more densely in shadowed areas, and they should be completely missing in lit

areas. Their orientation and bending should be set in a way an artist would convey shape and illumination differences.

Our goal was to find a method to display hatch lines using screen space features only. Their orientation and bending are calculated using rendered buffers containing information like illumination tone, screen space normal vectors and depth. These values can easily be rendered using both real time systems and common production rendering packages. We wanted to decouple the generation of hatch lines from the processing of the original geometry, to avoid complex geometry preprocessing, and to make the performance independent from geometric complexity. The screen space approach has the advantage that uniform screen space density can be obtained easily, but it is challenging to make the lines look to follow the original surface.

2 Previous work

Hatching techniques can be divided into two major groups according to how the lines are rendered onto the screen. One group uses textures of hatch lines, and makes use of multiple texturing and texture blending [8] [6] [7] [10] [9] [15]. The typical problem with texture based techniques is that a proper parameterization is needed. Parameterization can be obtained by creating an UV layout manually, but this is a rather time consuming task and complex geometries often cannot be flattened without visible seams. 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 [13] [1] [5]. 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 [4] [12]. 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 curvateure 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 result-

ing in a so called *shower door effect*, where hatching lines appear to float over moving objects instead of moving with them.

In this paper we propose an image based technique. We place hatch lines uniformly on screen and determine their orientation according to pre-rendered buffers containing surface information like illumination, surface normal and camera space depth. We investigated several features that can be calculated from these buffers and can be used to determine the direction of hatch lines. We applied the technique to objects with typical geometries to compare the applicability and quality of each feature. We avoided any complicated geometry preprocessing, and used buffers that can be produced easily by any common rendering environment including hardware rendering and production rendering systems.

3 Hatching synthesis

Hatching lines should have a uniform distribution in screen space, which means that distant objects should not have denser hatching than close ones. 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. Hatching density also depicts current lighting conditions, thus illuminated image regions should have coarser hatching density than areas in shadows. This later requirement can make the implementation cumbersome, as we should examine the rendered image during rendering each line, whether the line should be placed, or it will make the local neighborhood too dark. The 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 do this we assigned a *priority* value to each random sample and reject the corresponding line if its priority is below the locally desired density value. To achieve the desired density the sample priorities should have a uniform distribution in image space. This technique is called *rejection sampling* which is a standard Monte Carlo technique.

We can produce uniformly distributed priorities by assigning increasing priorities to each sample. The quality of the distribution depends on the chosen random generation algorithm. We used two Halton low-discrepancy sequences for the two coordinates to generate random points on screen. Halton sequences have an important feature namely that a sampling with lower density can be obtained easily by truncating the sequence. This feature makes possible to use the sequence index as sample priorities, and ensures that rejecting samples with higher priorities will still produce uniform sample distribution. This stands for both locally and globally desired density values. Desired density is defined by a luminance image, which is typically the total illumination of the surface points in case of a 3D renderer framework.

After identifying the position of the sample we draw a textured triangle strip to form a single hatch line. The direction of the line will be determined by a vector field which is calculated from local properties. The calculation of the vector field is described in Section 4. The lines can be drawn directly on screen with orthographic projection, as image space samples do not need any 3D transformations, nor visibility testing. Lines can also be bended to better convey surface curvature, which is implemented as drawing the line strip along an arc. The amount of bending, thus the radius of the arc is also determined locally according to a scalar field created simultaneously with the direction field.

Figure 1 shows the basic elements of our hatching synthesis. Image (a) shows the positions of the generated lines. Image (b) shows the lines rotated and bent according to the calculated feature vector field (lines placed on the background are also dropped). Image (c) shows the effect of rejecting lines on lit areas. We also used a depth gradient based contouring to denote screen space object boundaries and discontinuities (Image (d)).

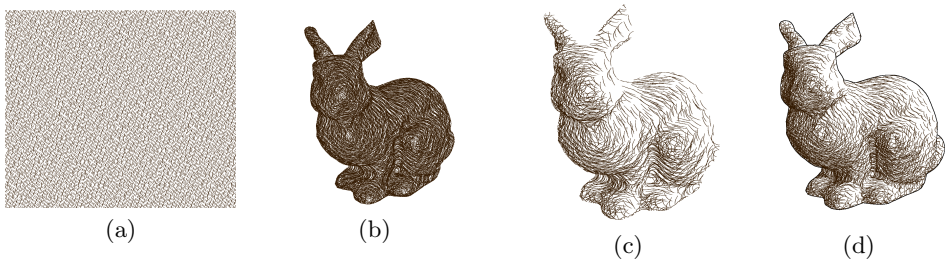


Fig. 1. Basic elements of the hatching rendering algorithm from left to right: hatch line positions in screen space (a), rotated and bent lines (b), rejected lines according to lighting (c), adding edge detected contours (d).

4 Screen Space Features

The direction of hatching lines is crucial in conveying the underlying surface. Even if the hatching lines are not bent at all, the curvature of the surface can be well expressed with appropriately oriented straight lines. In the following we will examine several features that can be used to calculate the direction field. We apply these hatching techniques to several test objects with different geometric properties, to show the strength and weakness of each selected feature. Line rejection is based on an illumination buffer that stores lighting information. We used hemisphere lighting with one light source. Hemisphere lighting avoids the flattening of unlit (back facing from the point of view of light source) areas with mapping a diffuse shading to the [0, 1] range.

Figure 2 shows renderings of the test objects with the use of a constant direction field. Though constant line direction with cross hatching is often used in

architectural drawings, it can hardly capture surface changes. Rendering with constant directions produces a flat look, especially in case of the Moria scene (image (f)), where —without contour lines — the shapes of the pillars are impossible to make out.

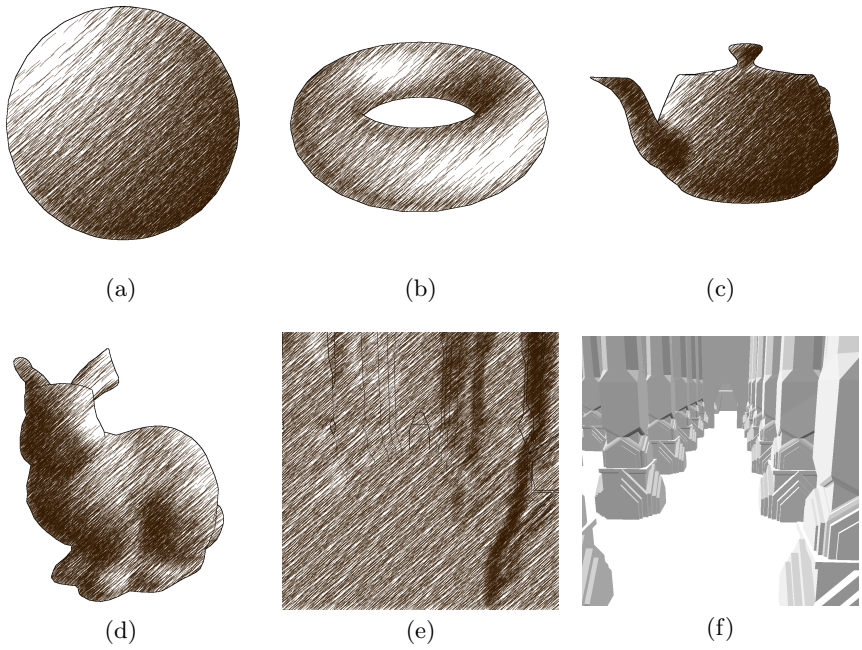


Fig. 2. Hatching using constant hatching direction. This technique produces a flattened look, surface features are hard to detect.

4.1 Luminance

One natural feature that can be used to calculate hatching directions is tone. Artists often use hatching direction to depict the direction of illumination changes, and draw lines perpendicular to the luminance dropoff direction. This is an extremely useful choice as luminance can be computed from any input, without any special surface information. Thus the hatching algorithm can be run as a pure post processing effect. Common rendering frameworks also have a built-in shading information rendering feature, thus no special plugins or shaders are needed.

In our real time system we take the image of the lit scene as an input, and calculate its gradient with a Sobel filter. This gradient will define the direction where illumination principally changes, and hatch lines are drawn perpendicular to this direction. We also use the magnitude of the gradient vector to define

the amount of bending of lines. We can assume that light changes more drastically on areas where the original surface has high curvature, thus we bend lines proportional to the magnitude of the illumination gradient.

Here we should consider that the gradient calculation is performed in screen space, but we should make lines appear to follow the original surface, thus light changes should be measured on the surface. We should calculate the amount of offset implied by moving one unit along one axis in screen space and scale the gradient accordingly. If we also have a buffer containing surface normal directions in screen space, the screen space projection of the normal vector to the main screen space coordinate axes will provide us the necessary scaling factors. This means that gradient values corresponding to areas with grazing view angles are scaled down to compensate the higher bending factor. We should note that we are using *weak perspective* here, as we do not take surface depth and perspective projection into account. We found that weak perspective has a pleasing effect as line density and length also does not change on camera depth.

We should also note that hatch lines are defined in 2D screen space. However, we want the lines to appear to follow the original surface. Thus, they sould appear to be rotated according to the surface normal in 3D space. This rotation has the same visual effect in screen space as decreasing the bending of the hatch lines whose direction is nearly perpendicular with the screen space projection of the normal vector. We should use these scaling factors not only for luminance based hatching but also for all other features.

Figure 3 shows our results with luminance based hatching direction and bending. This approach has a quite appealing result especially on curved surfaces. The method showed its strength on the sphere model, none of the other techniques can convey the underlying surface so well with this geometry. In general the method works well for curved surfaces as they have nice lighting changes along their surface. On the other hand, depending on the lighting conditions, flat areas can have no lighting change at all. Figure 4 a. shows the Moria scene lit by a directional light. For areas where the gradient is undefined we use a constant direction. As we are using directional lighting and flat surfaces the gradient is near zero all over the image, which leads to a noisy inconsistent hatching. Figure 4 b. shows an other weakness of the method namely luminance gradient direction can be different from the direction along the surface changes the most ie. the curvature, which makes hatching lines not to follow the surface correctly. The artifact can be clearly seen on the handle and the beak of the teapot model.

4.2 Surface depth

An other feature which can be examined is the change in surface depth. Depth buffers are easy to produce, and widely supported by common rendering softwares. Just like in case of luminance based hatching, we should find the depth gradient direction and magnitude to control the hatching lines. To calculate the depth gradient we can use a Sobel filter. However this makes tessellation clearly visible. Tessellation was not visible in luminance hatching, as we used a Phong shading with interpolated normal vectors, which approximates a smooth surface.

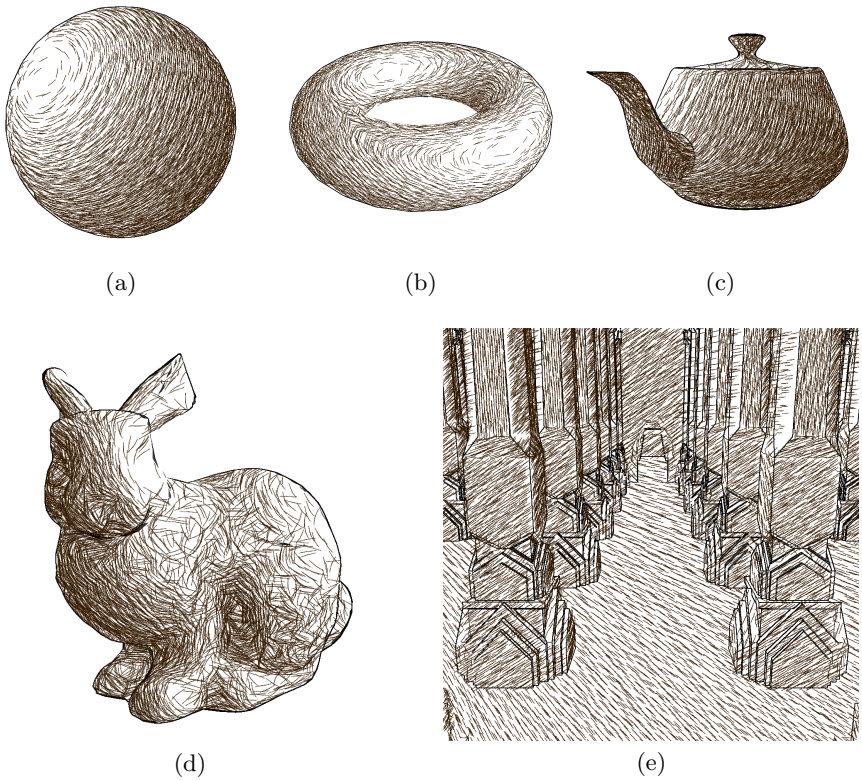


Fig. 3. Hatching using luminance gradient. Shape is well expressed.

On the other hand normal vectors also describe the depth change, thus can be used as smoothed gradient directions. Our surface depth based hatching uses the direction of projected normal vectors, and the magnitude of depth gradient calculated by a Sobel filter.

Figure 5 shows our results with depth based hatching direction and bending. Using this technique surface changes are well depicted, and even the flat areas of the pillars are handled robustly. However this technique creates concentric rotation of hatch lines around areas facing toward the camera in case of curved objects. These areas will catch the viewers attention, but it is not reasonable by any artistic means. These areas are also changing with camera or object animation which makes them even more obvious.

4.3 Curvature

So far we tried to make the hatching lines follow the underlying surface. In differential geometry there exists a quantity which describes exactly the direction of surface change: principal curvature direction. Surface curvature measures how

(a) (b)

Fig. 4. Artifacts of luminance based hatching: a. directional lighting produces large areas with zero gradients which makes the method unpredictable, b. lines on the handle and the beak of the teapot are not following the surface.

the surface bends along a given direction at a given point. Principal curvature is the direction which shows the highest curvature value. There exist several curvature measures among which the normal curvature suits our needs the best. Normal curvature along a direction is the reciprocal of the radius of the circle that best approximates the surface along that direction. As we wish to draw line arcs rotated according to the surface change, principal curvature direction and principal curvature values look to be our best choice [2]. Principal curvature directions and values can be calculated directly from the rendered mesh, which is a costly preprocessing step [11]. Though GPU implementation of geometry processing also exists [3], current main rendering frameworks do not have built in support for curvature buffer rendering. If we would like to free our technique from geometry processing, and want to have a widely usable solution, we should search for a screen space calculation technique.

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:

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

where $\boldsymbol{u}$ and $\boldsymbol{v}$ are orthonormal tangent vectors on the surface. In our case, they are the projections of the screen space unit $\boldsymbol{x}$ and $\boldsymbol{y}$ 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. Image space

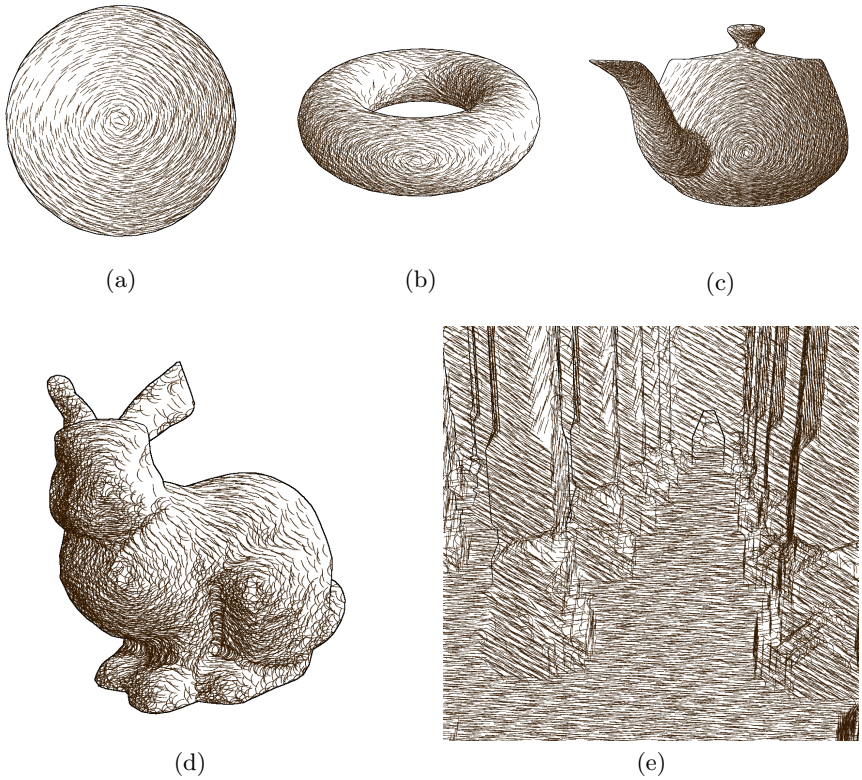


Fig. 5. Hatching using depth gradient. Shape is well expressed, flat areas are handled robustly, but annoying concentric patterns may appear.

methods using the curvature field for contour rendering usually use an iterative anisotropic diffusion technique to smooth out final curvature directions [14]. However as hatching line positions sample the direction field relatively coarsely, the tessellation is not visible on the hatching image.

Figure 6 shows our results with curvature based hatching direction and bending. Though curvature based hatching works very well on objects that have well defined principal curvature directions (like a torus), its behavior is undefined in case of surfaces with no principal curvature directions (like a sphere) or flat surfaces (which do not have any curvature at all). The method had unpredictable results for the Moria pillars, but even the teapot object — which has well defined curvatures — showed unnatural hatching. In case of the teapot model the algorithm worked too well, it found the main surface elements that the object was built up from. Even our eyes do not recognize these features so clearly, that is why an artist would probably choose different hatching directions.

The problem with flat objects and sharp corners can be overcome with the rounding of edges, introducing more tessellation at these areas, and with a slight

perturbation of the surface if needed. However, this shifts all responsibility to modellers, which is really not an effective option. Figure 7 shows hatching rendering of the Moria scene using a state-of-the-art object space technique [13] and our curvature based technique. Both rendering used a refined, smoothed geometry to handle originally flat surfaces.

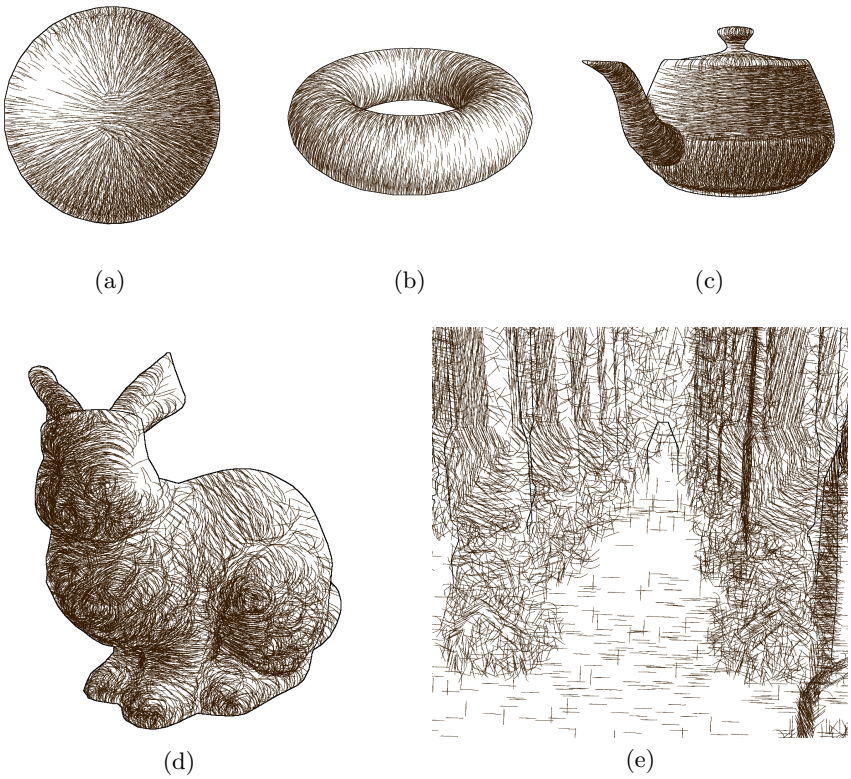
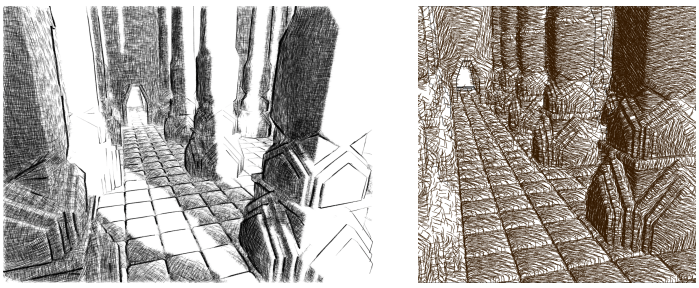


Fig. 6. Hatching using principal curvature values and directions. Shape is well preserved in case of smooth objects with well defined principal curvature directions. However flat areas and sphere like areas (with no exact principal curvature direction) are unpredictable.

5 Implementation

We implemented the hatching synthesis algorithm in a standalone application using OpenGL and GLSL shaders. Direction field calculation and the Sobel filter was implemented with full screen quad image processing. Hatching lines were rendered as triangle strips. We should note that on current hardware hatch



(a) (b)

Fig. 7. Hatching on smoothed surfaces using object space (a.) and the proposed curvature based technique (b.).

lines can be rendered as point primitives and can be extruded to a bent and textured triangle strip with the use of a geometry shader. However, to support older hardware and hardware with limited capabilities (like handheld devices), we used a vertex shader to rotate and bend a basic triangle strip.

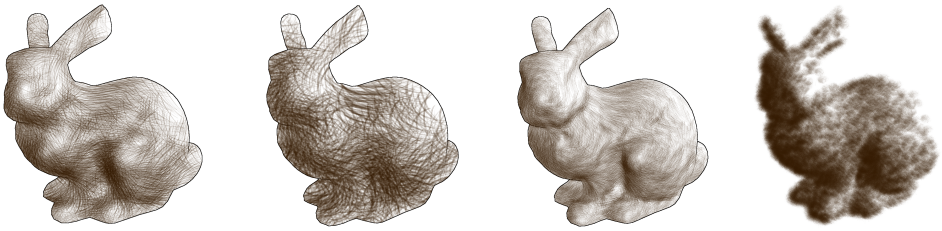
The performance of the hatching line rendering does not depend on the actual geometry. But we should take into account that most of our features require an extra rendering pass to store additional surface information, thus performance is afterall influenced by geometry complexity. Using multiple render targets on newer hardware can greatly decrease this drawback. On the other hand, screen resolution has a great effect on render speed, as we use image processing operations and calculations on a per pixel basis. Screen resolution only slightly influences the final rendering of lines, as the number of lines does not depend on resolution, here the only limiting factor is the fill rate of the GPU.

The images presented in this paper were generated using 50000 textured hatching lines and 1200x1000 resolution application window. All renderings run at real time speed: 40 FPS for simpler geometries (torus, sphere, teapot, and bunny), and 30 FPS for the Moria scene which had much higher polygon count. We did not notice performance differences between the techniques using different surface features. We measured the performance on a Geforce GTX 480.

6 Conclusion

We presented a realtime hatching rendering algorithm that randomly places textured hatch lines on the image plane. Line rejection according to lighting conditions is efficiently achieved with low-discrepancy sequences and rejection sampling. We examined several screen space features that can be used to calculate hatching directions and hatching line bending amount. Hatching lines can have a wide variety of styles by adjusting line density, width, length, maximal bending and applying artistic textures (see Figure 8). Our GPU implementation runs at realtime frame rates, but the proposed technique is also suitable for common production rendering frameworks. As a future work we should examine the

possibility to combine the proposed features and choose the best available one. We can use the illumination gradient magnitude, the curvature values and their ratios to make a good decision. We can make enhancements to force temporal coherence, too.



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

Fig. 8. Hatching synthesis with different artistic styles achieved by applying different line texture, density, width and length.

Acknowledgements

This work has been supported by projects TÁMOP-4.2.2.B- 10/12010-0009 and OTKA PD-104710.

References

1. Gershon Elber. Interactive line art rendering of freeform surfaces. *Comput. Graph. Forum*, 18(3):1–12, 1999.
2. Ahna Girshick, Victoria Interrante, Steven Haker, and Todd Lemoine. Line direction matters: an argument for the use of principal directions in 3d line drawings. In *NPAR*, pages 43–52, 2000.
3. Wesley Griffin, Yu Wang, David Berrios, and Marc Olano. Gpu curvature estimation on deformable meshes. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 159–166, New York, NY, USA, 2011. ACM.
4. Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *PROCEEDINGS OF SIGGRAPH 2000*, pages 517–526, 2000.
5. 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.
6. 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.
7. 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.

8. 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.
9. 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.
10. Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *In Proceedings of SIGGRAPH 2001*, pages 579–584. ACM Press, 2001.
11. Szymon Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. In *Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, 3DPVT '04, pages 486–493, Washington, DC, USA, 2004. IEEE Computer Society.
12. Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive penandink illustration. In *In Proceedings of SIGGRAPH 94*, pages 101–108, 1994.
13. Tamás Umenhoffer, László Szécsi, and László Szirmay-Kalos. Hatching for motion picture production. *Comput. Graph. Forum*, 30(2):533–542, 2011.
14. Romain Vergne, Romain Pacanowski, Pascal Barla, Xavier Granier, and Christophe Schlick. Light warping for enhanced surface depiction. *ACM Trans. Graph.*, 28(3), 2009.
15. Johannes Zander, Tobias Isenberg, Stefan Schlechtweg, and Thomas Strothotte. High quality hatching. *Comput. Graph. Forum*, 23(3):421–430, 2004.