Vis Comput (2015) 31:1511–1519
DOI 10.1007/s00371-014-1030-6

ORIGINAL ARTICLE

Rendering chamfering structures of sharp edges

Ling-Yu Wei · Kan-Le Shi · Jun-Hai Yong

Published online: 25 September 2014

Abstract In most realtime applications such as 3D games, in order to reduce the complexity of the scene being rendered, objects are often made by simple and large primitives. Thus, the phenomenon of *edge highlighting*, which would require chamfering structures made by lots of small patches at the seaming, is absent and is often faked by "highlights" drawn on the texture. We proposed a realistic realtime rendering procedure for highlighting *chamfering structures*, or *rounded edges*, by considering specified edges as thin cylinders and obtained the intensity via integration. We derived a brief approximated formula generalized from Blinn's shadow model, and used a precomputed integration table to accelerate the render speed and reduce resources needed. The algorithm is implemented with shader language, and can be considered as a post-process on original result. Evaluation shows that the effect on rendering speed is limited even for scenes with large scale of vertices.

Keywords Real-time rendering · Chamfering structure · Edge highlighting

L.-Y. Wei (✉)
The Institute for Theoretical Computer Science (ITCS), Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
e-mail: cosimo.dw@gmail.com

L.-Y. Wei
Geometric Capture Lab, Computer Science Department, University of Southern California, Los Angeles, United States

K.-L. Shi · J.-H. Yong
School of software, Tsinghua University, Beijing, China
e-mail: shikanle@gmail.com

J.-H. Yong
e-mail: yongjh@tsinghua.edu.cn

1 Introduction

Photorealistic image synthesis is one of the most important goals in Computer Graphics. While realistic images can be generated using offline rendering algorithm such as ray-tracing with detailed models and measured bidirectional reflectance distribution function (BRDF), real-time rendering often has to sacrifice the detail of models and adopt approximated algorithms and shading models. Although the shading models [1–3] have been widely investigated and can perform acceptable highlight effects on a smooth curve, an important phenomenon, *edge highlights*, can not be generated properly using such models if no thin curved patch is defined at the sharp seaming of adjacent surfaces.

In the field of online rendering, most models do not define the chamfering structures on the edges in order to reduce the amount of primitives emitted to the pipeline. Thus edge highlights can hardly be seen in realtime applications. However, there is little work being proposed to generate even approximate edge highlights.

Several works from Takahashi's group [4–6] has been considered edges as partial cylinder and obtain the precise visible area and integration of intensity to generate accurate highlights. Although their works are still too slow to implement and can hardly ported on GPU architecture, the idea of treating edges as cylinders can be further improved.

This paper presents an approximate algorithm implemented using shader language, therefore supports realtime rendering while producing acceptable edge highlights. The input contains the original model and certain edges of it, marked either manually or automatically. Such edges are assumed as very thin partial cylinders with widths less than 1 pixel. Then a separated shader draws such edges using integrated intensity calculated from our reflection formula. This image is blended with the normally rendered scene with the

alpha channel depending on the distance from edges to the camera, to generate a final result.

Although we also consider edges as thin cylinders, a completely different method on computing intensity is used. And we have made the following improvements comparing to work from Tanaka et al. [6]:

- We used a two-step rendering method by assuming the edge occupies each whole pixel it covers and then adjust it according to controllable parameters. Therefore we do not need to compute the exact occupied area as Tanaka et al. [6] does, but renders more efficient and supports potential visual effects.
- The process on vertices is discarded, because it reduces the rendering speed while hardly noticeable.
- We adopt a precomputed integration table, so that the rendering time does not depend on the highlight coefficient for different materials. In traditional Blinn's model, a higher shinning coefficient often cost more time because of the exponent operation. And for integration the formula would become much more complex and time-consuming.
- The algorithm is suitable for parallel rendering on GPU, thus makes realtime edge highlighting possible. This is the biggest improvement we made.

In this paper, conventional offline methods on edge highlighting are reviewed in Sect. 2. Then Sect. 3 introduces the detailed implementation and derivation. Section 4 presents the performance and limitation of our algorithm. Finally, the conclusion is shown in Sect. 5.

2 Related work

Edge highlighting is relatively easy to implemented for offline rendering. The conventional method is to build the chamfering structure using thin curved patches. Most CAD softwares can automatically generating such structures [7]. However, in order to obtain smooth but edge highlighted results, the size of patches has to be very small compared to the entity of model, and therefore the complexity of model increases, and most triangle facets of a model gathers along its edges, which is a small area compared to the whole model. This may cause floating error or degenerate cases unexpectedly.

For modern computer games, coarse models are widely used and artists draw shadows or highlights directly on their textures. Such kind of fake highlighting can not reflect to the movement or intensity of the light source, but is still useful for static scene and non-important objects. Other techniques like drawing edge lines with a lighter color [8] or detecting edges online [9] exists but add extra constraints on the angle of neighboring facets, which are also impractical.

Previous work [6] regards edges as partial cylinders and tries to compute the exact area the edge occupies in one pixel using Cross Scanline Algorithm. The result is theoretically precise excluding floating errors brought by operations, but needs a long time to do the scanning. In addition, the scanline based algorithm is not compatible for GPU's highly parallel architecture thus hard to be converted to shader language. On the other hand, since the result was done using Blinn's approximate model, it can not reflect the Bidirectional Reflectance Distribution Function (BRDF) for different materials. This indicates its incompatibility with offline rendering techniques nowadays.

3 Image generation

We adopts the Blinn's shadow model, of which the rendering model is simplified into diffuse and specular parts, and both of them are described by simple formulas, thus we can derive the exact form of integration and evaluate the value without doing actual integration on the GPU.

Previous works by Saito et al. [4] introduced precise calculation on rounded edges using Blinn's model and offline rendering method. It mainly focused on the rendering of rounded corners where multiple edges intersected together. However, in realtime rendering we choose to ignore such corners because our assumption of edge width ensures that each corners occupied no more than 1 pixel in the rendered image, while their precise computations cost significantly larger than the computations on edges. Also such calculations are hard to efficiently implement in parallel framework.

On the other hand, our work considers edges as partial cylinders and derives a very simple integrating form of the intensity, which simply depends on the spanning angles and the shinning factor. Then we use precomputed table to get approximated value. Such methods can be efficiently performed on modern GPUs using shader language.

It is convenient to represent chamfering edges as lines, with two vectors per vertex recording the normals of its adjacent facets. Such a structure of primitive (line) is supported by OpenGL and most modern GPUs. We assume the structure is tiny enough that each edge's width is not larger than one pixel. A separate shader is used to render the visible edges of the scene on a transparency layer above the layer containing original results. Specifically, in our experiment each model is described as face primitives and line primitives. A standard shader is applied to render faces only, while another shader renders the visible lines, then a third shader blends the two results with a computed weight. The first two fragment shaders are run in parallel, with each enables basic multisampling antialiasing [10]. Our algorithm can be described as two steps: firstly we applied the formula for computing intensity under the assumption that the width of edge is exactly 1, and

then interpolate this result with the origin result, in which has no highlighting, to get the final intensity. They are done by the second and the third shader described above. Following subsections describe how to derive the formula and how to blend the separated edge-only result with the original rendering.

3.1 Integration of specular reflection

Highlight effects are computed per pixel in the fragment shader part using N_1, N_2, V^{light}, V^{half} [V^{half} is the halfway vector between camera (viewer) and V^{light}, the light source], thus the two normals for each endpoint of an edge are passed through the vertex shader and then interpolated to restrict the range of spanning normals at a certain pixel. By Blinn's model, the intensity of the reflected light can be represented as:

$$I_{\text{out}} = (R_{\text{d}}(N \cdot V^{\text{light}}) + R_{\text{s}}(N \cdot V^{\text{half}})^n) I_{\text{in}} \tag{1}$$

where R_{d} and R_{s} are reflectance coefficients for diffuse and specular lights.

Calculating the intensity of diffuse part $(N \cdot V^{\text{light}})$ is simply a special case of specular part (integration of $(N \cdot V^{\text{half}})^n$) with $n = 1$, therefore we just describe how to efficiently evaluate the integration of specular reflection in this section.

The specular intensity is calculated by:

$$I_{\text{s}} = \int_A R_{\text{s}}(N \cdot V^{\text{half}})^n I_{\text{in}} \mathrm{d}S \tag{2}$$

where A represents the area of the pixel (which is projected into a 1×1 square in the camera's view). Assuming that the view point and light sources are both far from this area, we can ignore the variation of R_{s}, V^{half}, I_{in} comparing to the variation of N. So the intensity can be represented by:

$$I_{\text{s}} = R_{\text{sC}} I_{\text{inC}} \int_A (N \cdot V_{\text{C}}^{\text{half}})^n \mathrm{d}S$$

where R_{sC}, $V_{\text{C}}^{\text{half}}$, I_{inC} are the parameters for an arbitrary point C within A. Remember that for the diffuse part we can also use $V_{\text{C}}^{\text{light}}$ to ignore the change of V^{light}. As we can consider A as a partial cylinder, the integration becomes:

$$I_{\text{s}} = R_{\text{sC}} I_{\text{inC}} \int_{H_L} (N \cdot V_{\text{C}}^{\text{half}})^n \mathrm{d}h$$

where L is the path (arc) shown in Fig. 1, and H_L the projection of the arc on the direction (denoted as P) perpendicular to the orientation of camera and the axis (hence we can do a cross product to obtain P). We can not simply use $\int_L (N \cdot V_{\text{C}}^{\text{half}})^n \mathrm{d}l$, as the segments on the arc may contribute differently when projected into the camera. By assumption we force $H_L = 1$ and the projected height of the cylinder

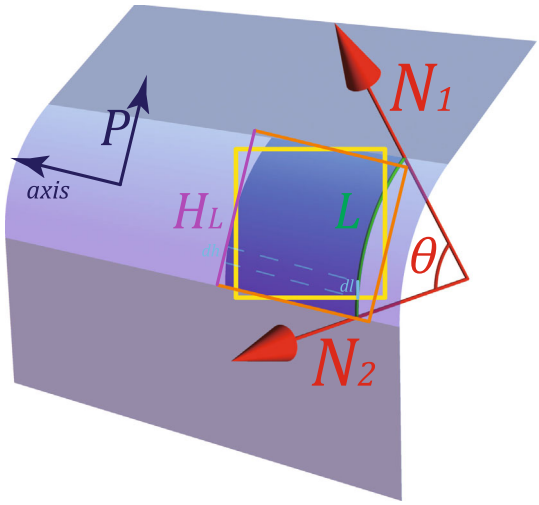


Fig. 1 Terminology for rendering a pixel. Note that we calculate the intensity of the *orange square* along the edge, instead of the *yellow one*

is also 1, in order to keep the project surface area equals to 1 square pixel. Also we are ignoring the change of cylinder width because normally the edge of a model would be much longer than 1 pixel, especially for coarse models, which means that the height of the whole cylinder is much longer than its radius, hence the change of width in one pixel is negligible. We strongly suggest readers to check Fig. 1 to understand all terminologies proposed.

However, we can transform this form into the integration along the arc L, by noticing that $\mathrm{d}h$ is actually a projection of $\mathrm{d}l$ on P. Hence the integration becomes:

$$I_{\text{s}} = R_{\text{sC}} I_{\text{inC}} \int_L (N \cdot V_{\text{C}}^{\text{half}})^n (\cos\sigma_l \cdot \mathrm{d}l)$$

where σ_l is the angle between P and the tangential direction at the point where $\mathrm{d}l$ is located.

Since N is always in the planar spanned by N_1, N_2, we can simply consider the projection of $V_{\text{C}}^{\text{half}}$ on this planar when calculating $N \cdot V_{\text{C}}^{\text{half}}$, meaning that the integration can be transformed into the following form:

$$I_{\text{s}} = R_{\text{sC}} I_{\text{inC}} \int_L \cos\sigma_l (N \cdot V_{\text{Cproj}}^{\text{half}})^n \mathrm{d}l \tag{3}$$

Now we are going to reorganize the formula under a specified coordinate system (e_x, e_y, e_z), as Fig. 2 shows: e_x is parallel to $V_{\text{Cproj}}^{\text{half}}$. e_z is parallel to $N_1 \times N_2$. $e_y = e_z \times e_x$.

$N \cdot V_{\text{Cproj}}^{\text{half}}$ can be written as $\|V_{\text{Cproj}}^{\text{half}}\| \cos\theta$. And $\cos\sigma_l$ can be derived by considering it a dot product between P and the tangential direction. P is a constant vector during this integration and can be represented by coefficients (P_x, P_y, P_z). And the tangential is $(-\sin\theta, \cos\theta, 0)$, since L is a circular

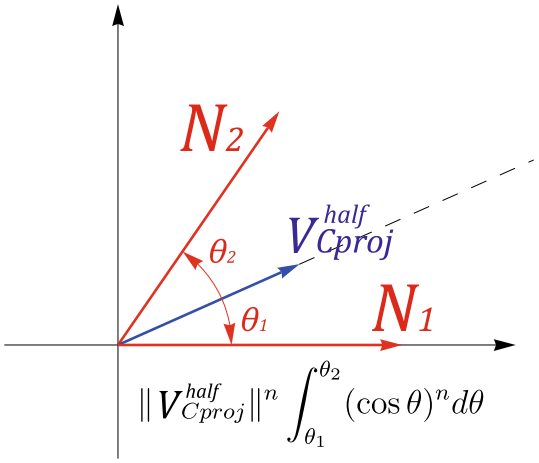


Fig. 2 Integration in polar coordinate system

arc. Hence we have:

$$\cos\sigma_l = -P_x \sin\theta + P_y \cos\theta \tag{4}$$

Also, we have $\mathrm{d}l = r\mathrm{d}\theta$ and since we are assuming the projected length of L is 1, which is equivalent to assume that the projected distance between the two endpoints of L is 1. Therefore, we have $|rN_1 \cdot P - rN_2 \cdot P| = 1$, meaning that $r = \frac{1}{|P(N_1 - N_2)|}$. Therefore, the final form of integration is:

$$\begin{aligned} I_\mathrm{s} &= \frac{R_\mathrm{sC} I_\mathrm{inC} \| V_\mathrm{Cproj}^\mathrm{half} \|^n}{|P(N_1 - N_2)|} \int_{\theta_1}^{\theta_2} (\cos\theta)^n (P_y \cos\theta - P_x \sin\theta)\mathrm{d}\theta \\ &= G_1 \int_{\theta_1}^{\theta_2} (\cos\theta)^{n+1} d\theta - G_2 \int_{\theta_1}^{\theta_2} (\cos\theta)^n \sin\theta \mathrm{d}\theta \end{aligned} \tag{5}$$

where we use G_1 and G_2 to denote coefficients.

The second part of the formula can be computed easily:

$$\begin{aligned} G_2 \int_{\theta_1}^{\theta_2} (\cos\theta)^n (-\sin\theta \mathrm{d}\theta) &= G_2 \int_{\theta_1}^{\theta_2} (\cos\theta)^n \mathrm{d}\cos\theta \\ &= G_2 \int_{\cos\theta_1}^{\cos\theta_2} x^n \mathrm{d}x \\ &= \frac{G_2}{n+1} [(\cos\theta_2)^{n+1} - (\cos\theta_1)^{n+1}] \end{aligned}$$

And the first part is obtained via a pre-computed table storing the value of $F(\varphi, n) = \int_0^\varphi (\cos\theta)^n \mathrm{d}\theta$ (It is passed as a texture to the fragment shader). Therefore, the whole integration can be computed efficiently. In our experiment, this table has $1,024 \times 128$ entries, for $\varphi \in [0, \pi/2]$ and $n \in [1, 128]$, while it is still possible to use a larger table for more precise results.

The approximation error that is introduced by the tabulation can be bounded as follows. For a fixed integer n, the linear interpolation when doing texture mapping is actually a piecewise linear interpolation. Therefore from Lagrange interpolation formula the error is:

$$\varepsilon(x) = \frac{(x - x_i)(x - x_{i+1})|F_n''(\xi)|}{2} \le \frac{h^2}{8}|F_n''(\xi)|$$

where $x_i = \frac{\pi/2}{1023} i$ for $i \in \{0, \ldots, 1023\}$. And $h = \frac{\pi/2}{1023}$ is the interval between every pair of x_i and x_{i+1}. Since:

$$F_n''(x) = \left((\cos\theta)^n\right)' = -n \cos x^{n-1} \sin x$$

we can obtain the maximum value by using mathematical software and the final error bound is about 2.0×10^{-6} for our table, which is small enough for common application.

3.2 Blending with original image

By using the method stated above to render the edges, we can obtain a separated image with only visible chamfering edges drawn on it. Such edges are all in 1-pixel width and therefore we can not simply over-draw the result on normally rendered scene, otherwise highlight edges far away from the view point will be overemphasized to have the same effect as near ones. Note that in Eq. 3 the intensity is proportional to the edge's radius r_e, whose maximum value could not exceed $r_\mathrm{max} = \frac{1}{|P(N_1 - N_2)|}$, or else the projected width would be larger than 1. For models with predefined radius on each edge, this can be an indicator to determine whether our method is sufficient or other technique may be needed to render over-thick edges. For radius not exceeding the threshold, we can get the actual intensity brought by chamfering edges by multiplying $r_\mathrm{e}/r_\mathrm{max}$. Note that if $r_\mathrm{e} < r_\mathrm{max}$, we also need the intensity from the flat primitive to get the actual intensity from the certain pixel.

Therefore, the new image is chosen to be blended with the original image using its alpha channel, which is also a basic feature that most display cards support. The weight in the alpha channel is computed by the edge's actual width and the distance between it and the viewpoint. It can be considered as the "viewed width" appears in the rendering, and for realistic rendering we should have this weight proportional to the view angle spanned by the structure. Hence the following formula is chosen in our experiment:

$$W(x, y) = \begin{cases} \frac{K \cdot r_\mathrm{e}}{\mathrm{d}(x,y) r_\mathrm{max}}, & \text{if pixel } (x, y) \text{ is drawn as edge } e; \\ 0, & \text{if pixel } (x, y) \text{ is not on any edge.} \end{cases}$$

where W denotes the weight stored in the alpha channel and $W(x, y)$ represents the weight for pixel (x, y); r_e is the predefined radius of that edge (typically we should set $r_\mathrm{e} \in [0, 1]$); $\mathrm{d}(x, y)$ means the distance between the viewpoint and the structure, which can be obtained directly. Note that r_e could be stored for each line, per vertex or be considered as a constant, depending on the manufacturer of the

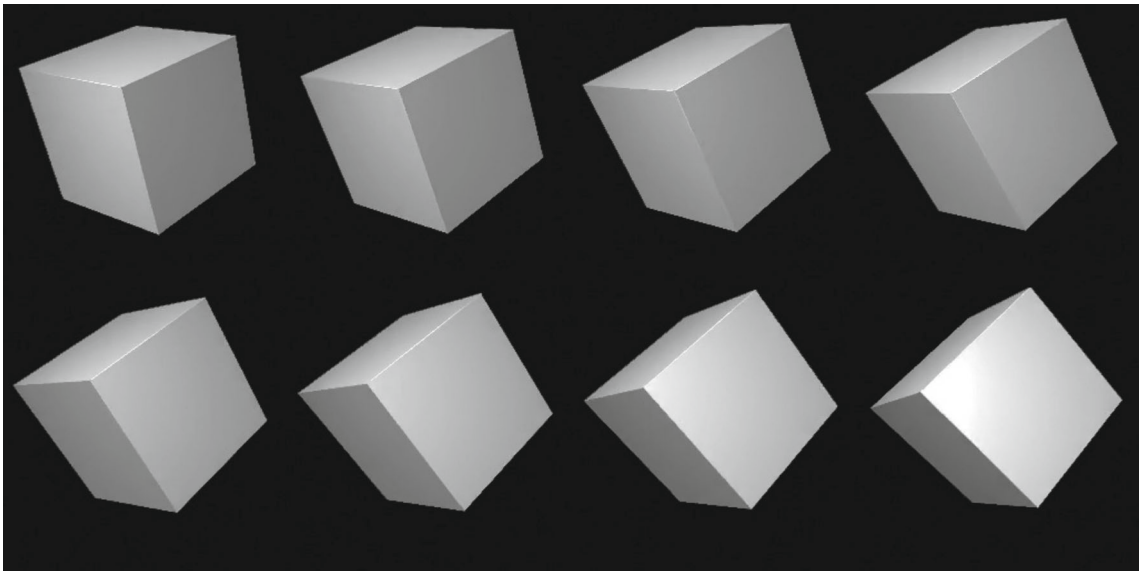


Fig. 3 Consecutive screen shots for a simple cube, edge highlighted (150 dpi, 8 nodes, 12 faces, 12 edges)

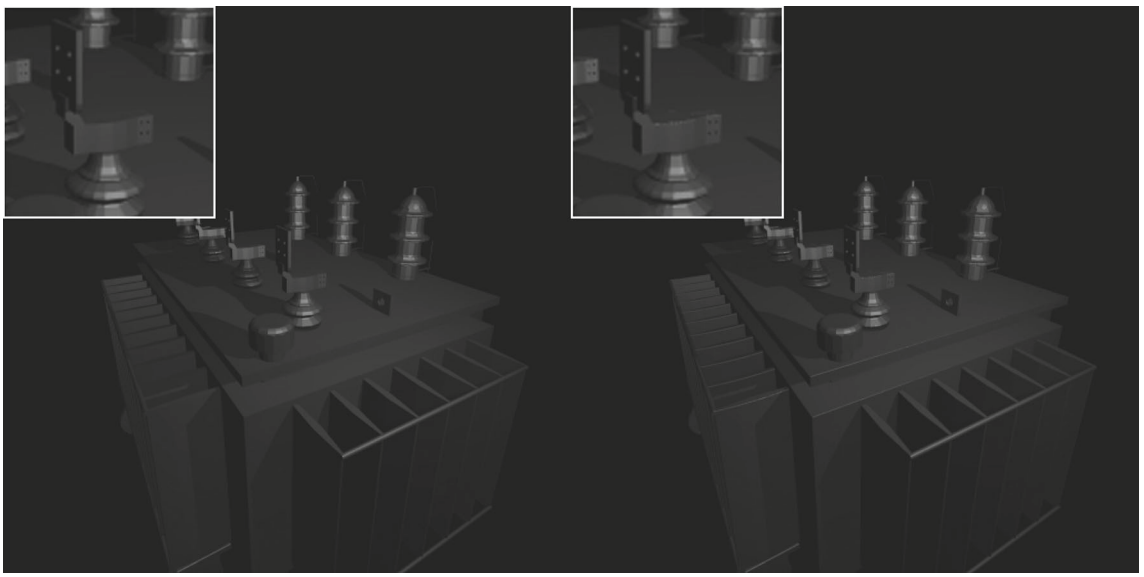


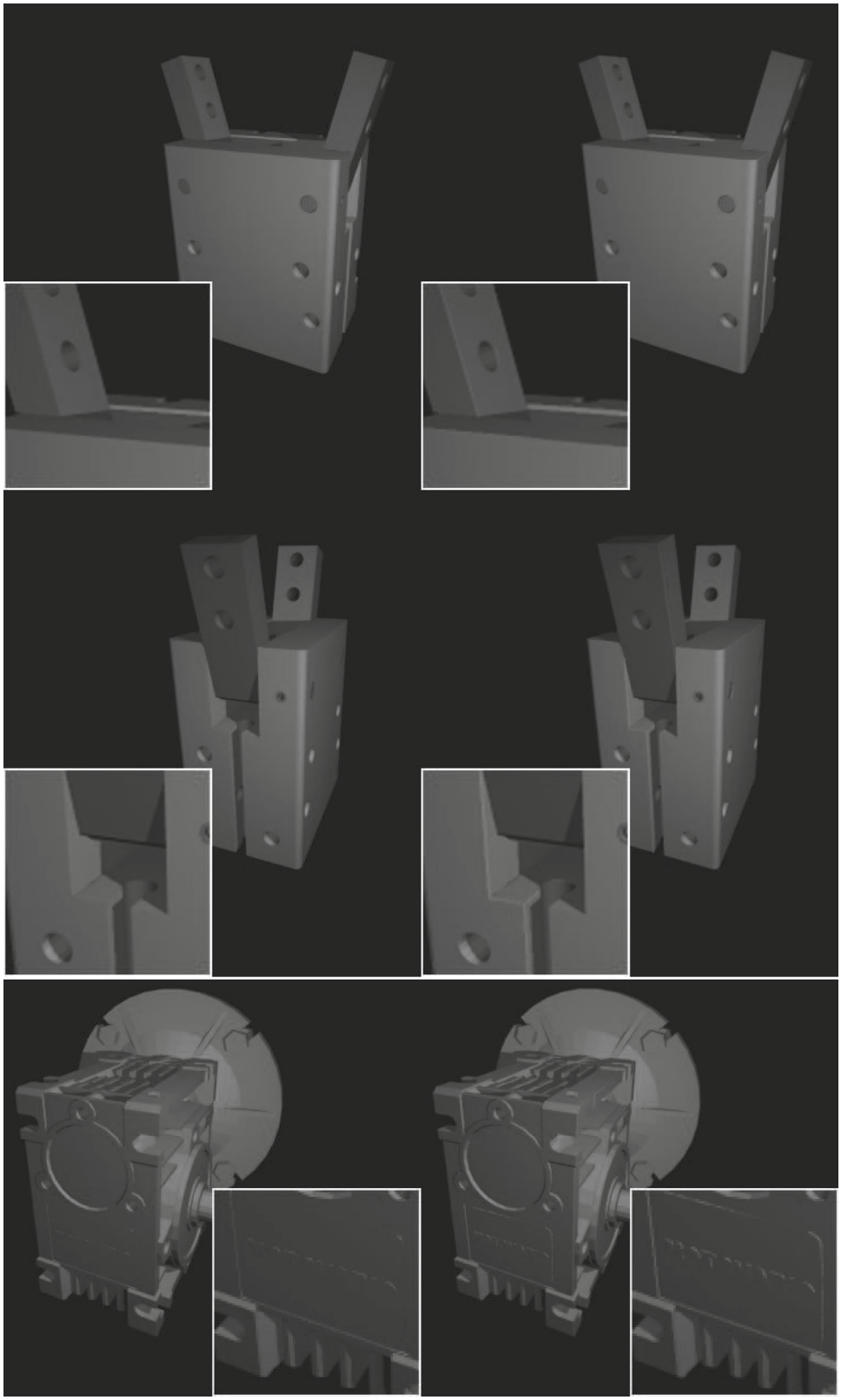
Fig. 4 **a** Comparison for rendering with (*right*) and without (*left*) edge highlighting (150 dpi, 83,702 nodes, 22,716 faces, 18,790 edges). **b** Gripper: 13,695 nodes, 5,259 faces, 2,036 edges; motor: 25,666 nodes, 8,858 faces, 5,424 edges (128 dpi)

model. For common storing formats such as OBJ or PLY, custom properties can be easily embedded into each primitive, so that our models can be handled via existed IO engines easily. K is an adjustable parameter to control the blending performance. Normally K should be set as the focal length of the camera (possibly the distance between the barycenter of the model and the viewpoint), so that the predefined width means exactly the weight of the shinning colors for lines at the focal plane. But the control of K allows model viewers to enhance the edge highlighting effect or remove such edges, without modifying the model or the rendering pipeline. This could be useful when designers are facing a practical software with little access to its core. For pixels that are not on any chamfering structure, their weights are 0 to keep identical with the origin image. The final image is blended as:

$$I(x, y) = W(x, y)I_{\mathrm{s}}(x, y) + (1 - W(x, y))I_{\mathrm{origin}}(x, y)$$

where I_{s} is the image rendered using only the chamfering structures and I_{origin} is the original image without such structures. In our implementation W are stored at the alpha channel of the buffer, and such a color blending procedure is standard for all modern graphic cards.

Fig. 4 continued



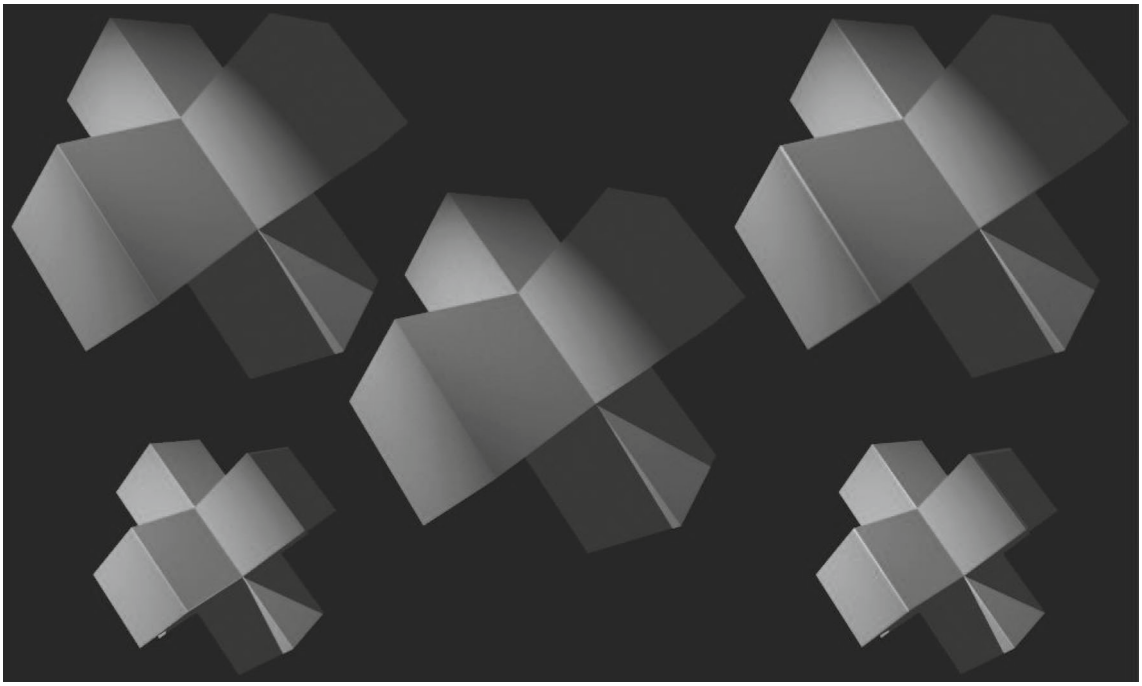


Fig. 5 Screen shots for the same block without edge highlighting (*middle*), with chamfered structure (*right*) and edge highlighted by our proposed algorithm (*left*). Note that for edges whose width >1 our algorithm may offer limited yet plausible result (150 dpi)

Since the extent of edge emphasis highly depends on the computation of W, different methods considering materials or visual effects may be adopted to achieve various purposes such as edge enhancement or wireframe extraction.

4 Performance and discussion

We used Panda3D [10] to organize the rendered scene and shaders. And we enabled the default anti-aliasing to get a smoother outcome. Specifically, we activate anti-aliasing for outputs from the standard shader and our edge-only shader, and then two images are blended using a weighting image W', anti-aliased from the weighting image W obtained in Sect. 3.2.

Figure 3 shows serial screen shots captured two times per second. The input model is a cube with no chamfering structure on its edges. Using our method the edge highlighting can be easily noticed thus more realistic. Related videos with 30 fps are also available[1] and we highly suggest readers to watch them, as the printed version cannot reflect the motion of highlights.

Figure 4 shows several models rendered with and without highlighting at the same orientation. Highlight edges are automatically added by iterating the whole model and add line primitives on every edge. The printed figures are in 128 or 150 dots-per-inch (dpi), which are approximately the same as typical computer screens. The program is run on an nVIDIA GTX 650 Ti graphic card and Fig. 6 compares the average rendering time per frame (i.e. inverse of framerate) in three cases: without highlighting, with Panda3D's default *autoShader*, and with our edge highlighting shader. AutoShader is a basic shader providing shadow map, highlight on faces, and anti-aliasing properties.

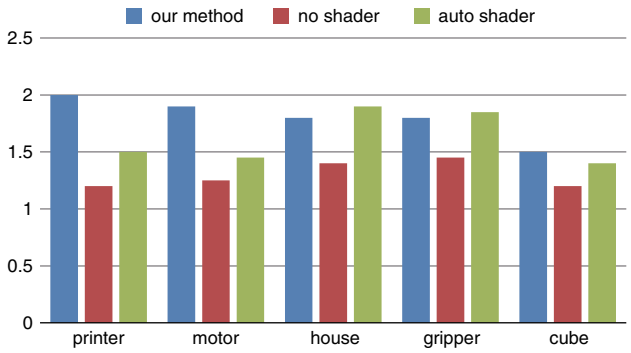


Fig. 6 Average rendering time (in millisecond) per frame by *different shaders*

Note that the time spent for the printer and the motor are much longer than others. The reason of this may be the huge amount of edges such models have, making the render of our shinning-edge-buffer slower. If a pixel is not on an edge we can simply discard the fragment shader to

[1] DropBox folder: https://db.tt/bOASPvpT.

speed up rendering. This is often profitable because generally most pixels are on faces instead of edges, but for those models its benefit decreases. However, for most cases the time of our algorithm is comparable to those of autoShader. In addition, since modern monitors can only function under constant framerate at around 60–75 Hz (13–17 ms), our algorithm actually add little burden to the overall rendering pipeline.

However, our method has its limitation at the same time. Our method assumes the chamfering structures has width of at most 1 pixel, thus the fragment shader can be applied directly for drawing the lines. Chamfering structures too large or close enough to the viewpoint can not be properly rendered. Figure 5 compares the output of our algorithm on coarse model with model that has small chamfering patches on the edge. Note that for upper case, we cannot properly render the closet edge with width larger than 1, although the overall performance of its highlights is plausible. And for small blocks their result are similar, which shows the correctness of our method. Under such a situation, proper patches are still needed to render the obvious structures. This can be efficiently detected though, by comparing the actual width with $r_{\max}$ (introduced in Sect. 3.2) for certain pixels, at the very beginning of the shader's rendering procedure to avoid unnecessary computation or memory reading (Fig. 6). In DirectX11 [11], for instance, we can customize the Hull Shader so that if $r_{\max}$ exceeds 1 in certain line primitive, the Tessellator is invoked to generate small patches on the seaming.

5 Conclusion

We present a realtime method for rendering small chamfering structures automatically. Such a method reduces the complexity of input models while maintaining edge highlighting phenomenon, which is often ignored in real-time rendering. We show that such an approximate method can produce similar visual effects as adding detailed structure on models, while still in an efficient rendering speed. This can be used to improve performance on applications like computer games that require high speed rendering.

This is the first step for rendering edge highlighting realtimely, and our future works will focus on larger chamfering structures with width more than 1 pixel, or highlighting under other shadow models.

Acknowledgments Thanks for the first anonymous reviewer for this article who pointed out a mistake in the derivation of Eq. 3. And because of this I found out a deeper and more serious bug, making me overwrite almost the whole Sect. 3.1. The first author was supported by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the NSFC (61033001, 61361136003). The second author was supported by the Chinese 973 Program (2010CB328001) and the NSFC (61035002, 61272235). The third author was supported by the NSFC (91315302, 61173077).

References

1. Phong, B.T.: Illumination for computer generated pictures. Commun. ACM **18**(6), 311–317 (1975). doi:10.1145/360825.360839
2. Blinn, J.F.: Models of light reflection for computer synthesized pictures. SIGGRAPH Comput. Graph. **11**(2), 192–198 (1977). doi:10.1145/965141.563893
3. Cook, R.L., Torrance, K.E.: A reflectance model for computer graphics. ACM Trans. Graph. **1**(1), 7–24 (1982). doi:10.1145/357290.357293
4. Saito, T., Shinya, M., Takahashi, T.: Highlighting rounded edges. In: Earnshaw, R., Wyvill, B. (eds.) New Advances in Computer Graphics, pp. 613–629. Springer, Japan (1989). doi:10.1007/978-4-431-68093-2_40
5. Tanaka, T., Takahashi, T.: Precise rendering method for edge highlighting. In: Patrikalakis, N. (ed.) Scientific Visualization of Physical Phenomena, pp. 283–298. Springer, Japan (1991). doi:10.1007/978-4-431-68159-5_16
6. Tanaka, T., Takahashi, T.: Precise rendering method for exact antialiasing and highlighting. Vis. Comp. **8**(5–6), 315–326 (1992). doi:10.1007/BF01897118
7. 3ds Max. AutoDesk. http://www.autodesk.com/products/autodesk-3ds-max/overview (2014)
8. EdgeShade. Cinema4D Plugin by Biomekk. http://www.biomekk.com/index.phppage=1&cat=107&itm=18 (1.03.001)
9. Sloan, P.P., Shirley, P.: Edge-aware shaders. In: GPU Technology Conference (2012)
10. Panda3D. Carnegie Mellon University. www.panda3d.org (1.8.1) (2013)
11. Microsoft Corporation. Tesellation Overview for DirectX11. http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340 (2009)

Ling-Yu Wei is currently a PhD student in the Geometric Capture Laboratory, University of Southern California. He had been studied in Tsinghua University in Theoretical Computer Science till 2014. His research interests include digital image processing and computer graphics, mainly about model reconstruction, computational geometry, optimization and real-time rendering.

Kan-Le Shi is an assistant professor at the School of Software, Tsinghua University. He received his B.Sc. and Ph.D. degrees from Tsinghua University. His research interests are geometric modeling, computer-aided design, and computer graphics.

Jun-Hai Yong is currently a professor in School of Software at Tsinghua University. He received his B.S. and Ph.D. in computer science from the Tsinghua University, China, in 1996 and 2001, respectively. He held a visiting researcher position in the Department of Computer Science at Hong Kong University of Science & Technology in 2000. He was a post doctoral fellow in the Department of Computer Science at the University of Kentucky from 2000 to 2002. He obtained a lot of awards such as the National Excellent Doctoral Dissertation Award, the National Science Fund for Distinguished Young Scholars, the Best Paper Award of the ACM SIGGRAPH / Eurographics Symposium on Computer Animation, the Elsevier Outstanding Service Award, and several National Excellent Textbook Awards. His main research interests include computer-aided design and computer graphics.