QUID 2017, pp. 345-351, Special Issue N°1- ISSN: 1692-343X, Medellín-Colombia

HYBRID SHADER FOR SIMULTANEOUS EDGE DETECTION AND AMBIENT SHADING

(Recibido el 18-05-2017. Aprobado el 11-08-2017)

Ilya Vladimirovich Tsivilskiy
Kazan Federal University,
Republic Tatarstan, Russian Federation

Ruslan Rinadovich Gaisin
Kazan Federal University,
Republic Tatarstan, Russian Federation

Vlada Vladimirovna Kugurakova
Kazan Federal University,
Republic Tatarstan, Russian Federation
vlada.kugurakova@gmail.com

Abstract. In this work we present a novel post-processing effect that can be applicable for both edge detection visualization and for emulation of ambient shading. Unlike different classic shaders suitable for these tasks, our effect has single shader code and does both the effects simultaneously: with a certain settings, the same codebase works as the edge detector or an ambient occlusion effect or as both of them at the same time, that is why the shader is useful for rendering of industrial design, architectural works and engineering. The algorithm uses only depth texture without screenspace normal texture and without additional noise texture that is a key component of a rendering pipeline of most of known screenspace ambient occlusion algorithms. The shader consist of 2 stages: standard render pass (that produces scene texture with no lights and shadows) + contours extraction and (or) ambient occlusion pass. Presented algorithm is relatively simpler in number of operations, being more amenable for mobile platforms.

Keywords: line extraction, ambient occlusion, shading, silhouette, post-processing

1. INTRODUCTION

In most cases in 3D rendering we need to select some objects to get user's attention to it. The most obvious way to do such a highlighting it is to draw a set of lines around the "selected" object. These lines can show many different things, including various combinations of lighting, surface discontinuities and contour of the object. There are lot of approaches and methods to do non-photorealistic or photorealistic rendering, but no one of them can do combined task in one shader. Occluding contours critically depend on the surface normal: they are zeros of the dot product between the normal and the view direction. Ridges and valleys are defined as local maxima of curvature (Rusinkiewicz, 2004). "Brute-force" way is slow: it requires simplifications of geometry and spatial data structures to work in real time. Screen space ambient occlusion (Langer & Buelthoff, 2000) is also useful for producing both interior and exterior silhouettes, however, primarily it was developed as algorithm of shading (Fox, & Compton, 2008). Both these post-processing effects (Miller, 1994) have a deal with comparison of color of near-lying pixels. Basing on this, in present work we made universal screen post-processing shader that can work as edge-detection filter and as an ambient-occlusion filter.

2. PREVIOUS RELATED WORK

There are several algorithms for extracting most kinds of feature lines from 3D objects of the scene. A very simple way to produce occluding contours is to render the model with white direct lighting (but without color and textures), and then perform color thresholding (Burns, Klawe & Rusinkiewicz, 2005). Any region darker than a threshold is set to black (or the line color), and anything above the threshold is set to the background color. To extract contours within faces of a 3D mesh, one can use interpolated values of *n* dot *v* (*n* is a face normal vector, *v* is view direction) (McGuire & Hughes, 2004). Unfortunately, this technique depends on geometrical complexity of the 3D scene. Real-time ambient occlusion techniques are often implemented as screen-space ambient occlusion (SSAO) and horizon-based ambient occlusion (HBAO) (Shanmugam, & Arikan, 2007) (Bavoil, & Sainz, 2008). It uses data from the depth buffer (a color that depends on depth) instead of rendering n dot v. The main drawback is an overocclusion issue, which looks as dark halo artifacts in zones with large differences of depth buffer.

None of known edge detection techniques can produce ambient occlusion effect using single shader codebase at the same time. Moreover, implementation of ambient occlusion effect runs slow on middle-grade mobile GPUs due to complexity of shader instructions. Our goal is to produce shader that does contour extraction and ambient occlusion simultaneously and is fast enough to work on mobile devices.

3. IMPLEMENTATION

3.1 Basic idea

The key idea of the algorithm is to average dot product of screen's normal vector and a tangent vector of all visible surfaces. To compute screen space tangents we use encoded depth texture. The screen z coordinate of each generated pixel of a scene is stored in a depth texture representing depth buffer (z-buffer). The depth buffer will allow the method to reproduce the real screen space coordinates of each rendered point of the scene.

It can be done by commonly used Laplacian operator (Ritschel, Grosch & Seidel, 2009). Unfortunately, it cannot produce contours of individual object: it works for the whole scene. To select the object of interest (i.e. to draw its contours) we need to render it separately:

- Initialize new screen texture. Let it be b/w.

- Clear the whole screen by black solid color.

- Render the object of interest filled by white solid color.

- Apply the Laplacian operator to each pixel of the screen texture. Use finite difference approximation of Laplacian operator: $L(x,y) = c(x,y+1) + c(x,y-1) + c(x+1,y) + c(x-1, y) - 4*c(x,y)$. Where $c(x,y)$ is a color of a pixel of b/w screen texture. Now one can see the outline of the object of interest:

- Invert screen texture. Now object's edges are black, and void space is white.

- Initialize new screen texture and render the entire scene with textures, lighting, etc.

- Multiply b/w and scene texture. The object of interest has black outline.

On the other hand, this algorithm is also applicable for the whole scene after exclusion of the first three steps.

3.2. Depth encoding

To encode z-values as RGB colors of the depth texture, we use well-known approach described in (Umbaugh, 2010). Fragment shader projects vertices of a scene models to the normalized screen space by means of perspective model-view-projection matrix. We use Adobe Graphics Assembly Language (AGAL) (Pharr & Fernando, 2005) to implement our shader. In AGAL syntax, it looks the following way:

m44 op, vt0, vc0 // project vertex coords (vt0) by MVP matrix (vc0)

m44 vt1, vt0, vc4 // project vertex (vt0) according to scene transform matrix (vc4)

sub v0, vt1, vc5 // compute a vector (v0) from vertex to camera position (vc5)

The "scene transform" matrix (vc4) transforms from model to world space. To represent the 32-bit depth buffer, we need to multiply interpolated squared distance between camera and vertex to powers of two. Let the distance from camera to the far plane of the view frustum is “f”. As the value of expression $\sqrt{f^2+f^2}$ is largest possible distance for any view frustum, we need to divide by it to normalize obtained distances. Finally, the AGAL pixel code looks like:

dp3 ft0.z, v0.xyz, v0.xyz // get a squared distance (ft0) by computing squared length of v0

mul oc, fc0, ft0.z // pack the depth (ft0) by multiplying to powers of two (fc0)

In this code the register fc0 has the following values:

$$fc0.xyzw = \left(\frac{1}{2f^2}, \frac{2^8}{2f^2}, \frac{2^{16}}{2f^2}, \frac{2^{24}}{2f^2}\right) \tag{1}$$

Here we took into account that the maximum depth value can be approximated this way for the simplicity of computations (no need to calculate square root):

$$\max depth = \frac{1}{\sqrt{f^2+f^2}} \approx \frac{1}{2f^2} \tag{2}$$

3.3. Neighbor pixels processing

Once the screenspace depth map is done, we can compute tangent vectors. To make this we simply sample depth values in adjacent pixels of the pixel in processing. Texture coordinate (UV) offsets for adjacent pixels are also known, that is why the resulting tangent vector is described by expression:

$$\vec{t} = (dx - 0.x,\ dy - 0.y,\ z - 0.z) \tag{3}$$

Here “0” is a position vector of the processed pixel; “dx” and “dy” are horizontal and vertical distances between processed pixel and its neighbor pixel. Both the “dx” and “dy” are specified in texture coordinates and corresponds to standard “u” and “v” components in range [0..1] for the whole screen. In our case, we use normalized “dx” and “dy” according to width and height of the screen texture:

$$dx = \frac{du*r}{width}; dy = \frac{dv*r}{height}; \tag{4}$$

Here “r” is a sampling step in pixels, because we can sample not only nearest neighbors, but skip them and make a leap sampling. In figure 1 the processed pixel is point “0” and its left and right neighbor pixels are “L” and “R” respectively. Surface tangent vectors are respectively: “0L” and “0R”.

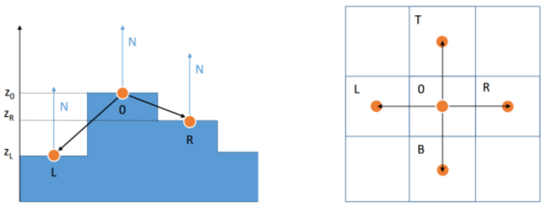

Figure 1. Idea of the shader. Left: side view of a depth map. Right: we use the same notation for top view of a depth map. “0” is a currently processing pixel. Then we compute tangent vectors from its neighbor pixels of a depth texture.

When the four tangents for the currently processed pixel of the screen had calculated, we compute a dot product of them and of the normal vector of the screen: “N” = (0, 0, 1). According to the figure 1: the lower the neighbor pixel is – the higher is the dot product. Differences between high and low depths finally will look like different intensity of a resulting color: the most covered from the incident light places of the 3d scene will be shady, especially if the sampling step is greater than one pixel (r > 1).

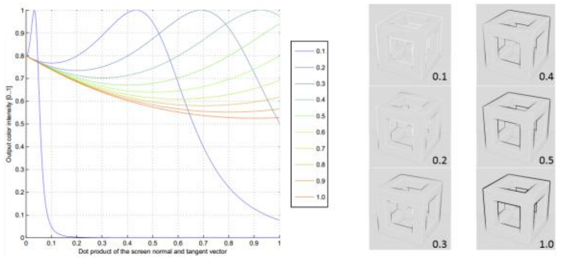

Figure 2. Left: Output color depending on the angle between surface tangents and screen normal. Angle is specified in terms of dot product [and is in range [0..1]. Different rainbow colors correspond to the bleach factor. Right: Result of the shader's work. Each number at the right bottom corners of each frame points to the bleach factor.

3.4. Fixing too dark or too bright zones

In case of using this algorithm as the edge detection filter, it will look okay, but for effect of ambient occlusion it produces improper results. To overcome this unnecessary dark halo we add so-called bleach factor that will make too dark color values lighter. Then output color looks the following way:

$$oc = dp * \left(2 - \frac{dp}{L}\right) \tag{5}$$

Here "dp" is a dot product of tangent and screen normal; "L" is the bleach factor which is in range [0..1]. For low values of "L", initially too dark zones will be lighter, and for high values of "L" dark zones stay dark. Then we simply accumulate these four samples of corrected depth and divide by total count of samples (=4) to set the output color in correct range (see "Appendix A" of this paper). Pixel Bender do not support loops, that's why all the four depths of neighbor points were sampled step-by-step by repeating key parts of a code. Note that in the "Appendix A" we use depth map as an input image. For clarity, this depth map is encoded as grayscale texture, and its red, green and blue channels have the same color values in range [0..127] or [0..1] in Pixel Bender's notation. Pixel Bender's code illustrates the algorithm well, but is not applicable for real-time rendering, due to lack of performance. For practical use, we also have translated the code to Adobe Graphics Assembly Language. We used almost the same approach to rendering of 3d meshes as in previous work of our colleagues (Gaisin, Nikiforova, Kashapov & Kashapov, 2014). But in this case we should render the scene twice:

- Render the whole scene (mesh textures, lights, etc.) to the BitmapData. BitmapData is a standard class of the Action Script 3.0 programming language. It stores color data of pixels inside rectangular area.

- Render the packed 32-bit depth texture of the scene (no textures for meshes, no lights) to the BitmapData.

- Render the screen's rectangle, sample previously stored BitmapData objects of the scene and depth map and perform the final post processing: a function creates two triangles in GPU's memory to render screen's rectangle, then draws its triangles with our post processing shader. This shader takes screen texture and depth texture (fs0, fs1) as input sources.

The AGAL version of the shader is supplemented by new parameter "range". For most cases the range value 0.8 * camera.far produces the best result (where camera.far is a far clipping plane of a camera's frustum). One can change this parameter to make near-laying objects neater.

4. RESULTS

First, we have checked the output of our shader written in Pixel Bender. We use grayscale depth map representing the geometry of a hollow cube (figure 3). On the left side is the depth map that is used as the input texture of a shader. Central part of the Fig. 3 corresponds to the effect of ambient occlusion. The most occluded parts of the cube are darker. This image is produced with the following parameters: contrast=64 (of 100), r=7 (of 10), bleach=0.05 (of 1.0), brightness=0.1 (of 2.0). Right side of the Fig. 3 shows the result of the same shader's work, but with modified parameters: contrast=26, r=1, bleach=1, brightness=0. Now it works like the edge detection filter. Here are some notes about the meaning of these parameters:

- In physical world the contrast is the difference in color that makes an object distinguishable. In our case, we adjust the resulting color by the following formula:
  $color = color^{contrast}$.

- "r" is the distance (in pixels) between currently processed pixel and its neighbors, where we sample the depth. Reasonable range of "r" is [1..10]. Low values of "r" produce narrow lines. High values of "r" produce thick lines.

- Bleach factor makes too dark areas lighter and is well described in previous section of this paper.

- Brightness corresponds to uniform brightness of the resulting picture. In our case its inverted brightness, so zero value points to bright picture and high values points to dark picture.

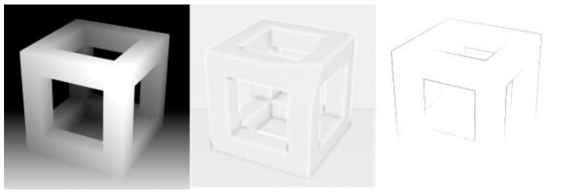
Figure 3. From left to right: initial 8-bit depth map; Pixel Bender shader output working as ambient occlusion effect; Pixel Bender shader output working as edge detection filter. The same codebase for all cases.

Now, it is time to see the results of our shader written in AGAL. Let us start with the edge detection (Fig. 4). The AGAL (Scabia, 2011) version of our shader has the following parameters: contrast=30 (of 1000), range=2000 (of 10000) for left part of Fig. 4 and range=1000 (of 10000) for central part of Fig. 4, radius=1.0 (of 5), bleach=0.2 (of 1.0), brightness=0.5 (of 2.0). The "range" parameter controls the contrast of small differences of a depth of parts of three-dimensional model. If the parameter is within the distance from the camera to the object, then small details are better visible. To display only the outer contour of the object, it is sufficient to set this parameter times greater: the shader can display both the outline of a model and its inner edges.

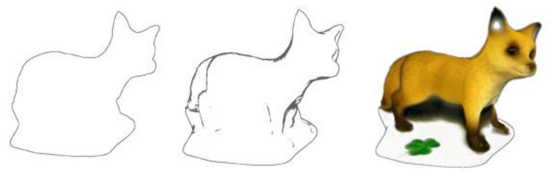
Figure 4. AGAL shader output for purposes of the edge detection. Left: show just the outline of the model, range=2000; Center: show the outline and other inner features of the model, range=1000; Right: result of the shader work – the scene texture is mixed with contours.

The shader can produce the effect of ambient occlusion (Figure 5). Using the same code base and the same scene, we increase the "r" parameter to smooth inner features of the model. Then we slightly increase the "bleach" and the "brightness" parameters. Final parameters are the following: contrast=30 (of 1000), range=1000 (of 100000), radius=5 (of 5), bleach=0.3 (of 1.0), brightness=1.0 (of 2.0).

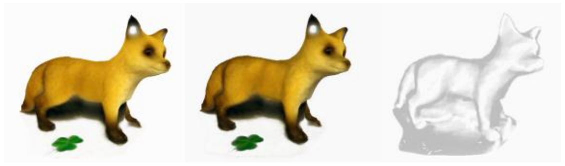
Figure 5. AGAL shader output for ambient occlusion. Render of an individual mesh. Left: scene texture; Center: scene texture and our filter with ambient occlusion shading; Right: raw output of our filter (without mixing with the scene texture).

For architectural and industrial rendering it is often good to display not only the scene with ambient occlusion and lighting, but also to make accent on the edges. The shader can do these tasks simultaneously. Fig. 5 shows how both the effect of ambient occlusion and the edge detection filter look together. As in previous cases, we use the scene without any lighting (and no shadows), but final image looks realistic. This effect can be achieved with the same setting as for Fig. 5, but with low "r" (r=1.5) and low overall brightness. It is also one of possible applications of our algorithm: to emulate direct lighting.

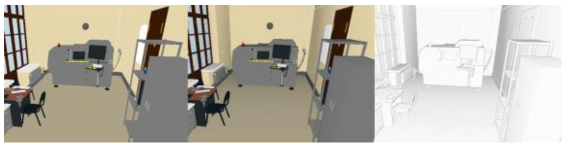
Figure 6. AGAL shader output for ambient occlusion and for edge detection. Rendering of interior of a room. Left: scene texture (no lights!); Center: scene texture and our filter (AO+edges); Right: raw output of our filter.

5. CONCLUSIONS AND FUTURE WORK

We have presented real-time post-processing shader that produces the effect of ambient occlusion and edge detection at the same time. The shader is written in two versions: in Pixel Bender 2.5 (for test purposes) and in AGAL (for real-time rendering). For screen resolution 1280x1024 we achieve 45-60 frames per second for scenes of one million polygons on desktop system equipped with Intel HD graphics 3000. For resolution 1024x720 we achieve 25-30 fps for the same 3d scene on reference mobile system with Mali-400MP2 on Android OS.

In comparison with modern techniques, the edge detection quality is more like the one obtained by reviewed screen-space algorithms of non-stylized line extraction based on Laplacian operator.

ACKNOWLEDGMENTS

This work was funded by the subsidy of the Russian Government to support the Program of competitive growth of Kazan Federal University among world class academic centers and universities.

REFERENCES

Bavoil, L. & Sainz, M. (2008). *Image-Space Horizon-Based Ambient Occlusion*. SIGGRAPH.

Burns, M., Klawe. J. & Rusinkiewicz S., Finkelstein, A., DeCarlo, D. (2005). *Line Drawings from Volume Data*. SIGGRAPH 2005.

Fox, M. & Compton, S. (2008). Ambient Occlusive Crease Shading. *Game Developer Magazine.*

Gaisin, R., Nikiforova, A., Kashapov, N. & Kashapov, R. (2014). *Development of the method of creation of the 3d model of topology of the surface by means of the optical microscope*. IOP Conf. Ser. Mater. Sci. 69.

Langer, M. & Buelthoff H. (2000). Depth discrimination from shading under diffuse lighting. Perception, 29(6),649–660.

McGuire, M., & Hughes, J. F. (2004). Hardware-determined Feature Edges. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering* (pp. 35–47). New York, NY, USA: ACM.

Miller, G. (1994). Efficient algorithms for local and global accessibility shading. *Proceedings of the 21st annual conference on Computer graphics and interactive technique*, pp. 319–326.

Pharr, M. & Fernando, R. (2005). GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. NY: Addison-Wesley Professional; 2005

Ritschel, T., Grosch, T. & Seidel, H.P. (2009). Approximating Dynamic Global Illumination in Screen Space. *Proceedings ACM SIGRAPH Symposium on Interactive 3D Graphics and Games, Boston*.

Rusinkiewicz, S. (2004). *Estimating curvatures and their derivatives on triangle meshes. Proceedings - 2nd International Symposium on 3D Data Processing, Visualization, and Transmission. 3DPVT 2004*.

Scabia, M. (2011). Vertex and Fragment Shaders. Retrieved from: http://www.adobe.com/devnet/flashplayer/articles/vertex-fragment-shaders.html

Shanmugam, P. & Arikan, O. (2007). Hardware accelerated ambient occlusion techniques on GPUs. *Proceedings of the 2007 symposium on Interactive 3D graphics and games*.

Umbaugh, S. (2010). Digital image processing and analysis: human and computer vision applications with CVIP tools. *2nd ed. Boca Raton. FL: CRC Press.*

APPENDIX

Appendix A - Implementation of fragment shader code (pseudocode)

Input parameters: contrast = [0..100], r = [1..10], bleach = [0..1], brightness = [0..2]

Input data: depth map stored in (src).
Output data: color of a processed pixel (dst).

```
// get UV coordinates of current pixel (oc)

// get depth value (z0) of currently processed pixel
of the depth map
pixel4 pix0 = sampleNearest(src, oc)
z0 = (pix0.r + pix0.g + pix0.b) / 3
```

***For each 4 neigbour pixels*:**

```
// sample depth in neighbour pixel, sample color
with specific offset (dx, dy)

dx = [-r,r], dy = [-r,r]

// get depth of a pixel
tpix = sampleNearest(src, float2(oc.x + dx, oc.y +
dy))
tz = (tpix.r + tpix.g + tpix.b) / 3
```

```
// compute vector from a central pixel to neigbour pixel (z0tz)
z0tz = float3(dx - oc.x, dy - oc.y, tz - z0);

// compute angle between the screen normal and previous vector = (N * z0tz)

tdp = dot(float3(0, 0, 1), z0tz)

// make too dark zones bright and adjust overall brightness, excluding negative values
tdp = tdp * (2 - (tdp / bleach))
tdp = 0.5 * max(tdp + brightness, 0)

// accumulate color
o = o + 1 / (1 + $tdp^2$)

end

// normalize 4 passes
o = o / 4

// add contrast
o = pow(o, contrast)

// return output color
dst = pixel4(o, o, o, 1.0)
```