A shader for simultaneous ambient occlusion and edge detection for reverse engineering

R R Gaisin\(^a\), K V Sharifullina\(^b\) and R N Kashapov\(^a,b\)

\(^a\)Engineering institute, Kazan (Volga region) Federal University, Kremlevskaya 18, 420008, Kazan, Russia
\(^b\)Kazan Physical-Technical Institute of RAS, Sibirsky Trakt 10/7, Kazan, Russian Federation

rusgaisin@gmail.com

Abstract. We have presented real-time post-processing shader, which 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). The algorithm is simple to implement and can be easily extended to other higher-level shader languages. It uses only depth texture without screenspace normal texture and without additional noise texture, which is key component of a rendering pipeline of most of known screenspace ambient occlusion algorithms.

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 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 shape. The question is how to draw lines that simultaneously convey several of these cues.

There are different types of bounding contours. Silhouettes represent boundaries between object and background. Occluding contour is a generalization of silhouettes: these mark any depth discontinuities, not just those against the background. This type of contour adds a lot of important detail to the drawing, but still does not convey shallow features, particularly those viewed head-on.

Sharp creases of the surface - works well for polyhedral objects, and is a frequent ingredient in technical drawings. The algorithm is quite simple: just look for a dihedral angle (that is, the angle between two faces connected along an edge) smaller than a threshold.

Many shape-conveying lines based on surface differentials. Occluding contours critically depend on the normal: they are zeros of the dot product between the normal and the view direction. Ridges and valleys (crest lines) are defined as local maxima of curvature [7].

Occluding contours (also known “interior and exterior silhouettes”) for any shape are locations of depth discontinuities. There are a few different ways of defining these, of which a very straightforward definition is simply those locations at which, from the current viewpoint, there is a depth discontinuity. Note that these are view-dependent lines. On the plus side, the view dependence makes it much more likely that these lines are interpreted as conveying shape, rather than as surface markings. On the other hand, this means that the lines will have to be recomputed for each frame.
On smooth surfaces, there is another definition of contours that is useful: contours are those surface locations, where the surface normal $n$ is perpendicular to the viewing direction $v$. That is, places where $n \cdot v$ is equal to zero.

Another important post-processing effect in computer graphics is screen space ambient occlusion (SSAO) [2]. SSAO is also useful for producing both interior and exterior silhouettes. However, primarily it was developed as an algorithm of shading. This technique emulates ambient lighting by computing shadows that nearby objects would casts on each other’s under the effect of an ambient shadowing. It is also can be performed in pixel shader, using the 3D scene depth buffer. For every pixel on the screen, the pixel shader samples the depth values around the current pixel and tries to compute the amount of occlusion from each of the sampled points. The occlusion factor depends on the depth difference between two sampled points.

Both this post-processing effects [3] have a deal with comparison of color of near-lying pixels. Basing on this, in present work we made universal screen post-processing shader, which can work as an edge-detection filter or as ambient-occlusion filter.

There are three basic sets of algorithms for extracting most kinds of feature lines from 3D objects of the scene. First, there are bitmap-space algorithms that render depth map (or Lambert-shaded surface), then extract contours by doing per-pixel processing (such as color thresholding) on the framebuffer. The advantage of this kind of algorithm is that it can be fast and easy to implement. A major drawback is that it makes it difficult to control the appearance and stylization of the resulting lines.

A second class represents multi-pass hybrid algorithms, which partially perform processing in object space, but the lines show up only in the framebuffer.

Final class of algorithm operates in object space – on the model directly. These algorithms are more complex, but they can provide good control over stylization technique.

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 [8]. 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. One drawback of this algorithm is that the thickness of the lines varies and becomes thicker in low-curvature zones. A way to get fixed a line thickness is based on curvature-dependent threshold.

For any constant-curvature region one can determine how thick the lines will be as a function of radial curvature of 3D surface.

Object-space approach provides two types of contour extraction. There are many implementations of object-space algorithm. Contours along the mesh edges produced via a loop over all edges to check whether each has one adjacent back face and one adjacent front face [9]. To extract contours within faces of a 3D mesh, one can use interpolated values of $n \cdot v$ ($n$ is a face normal vector, $v$ is view direction). Unfortunately, this technique depends on geometrical complexity of the 3D scene and can be relatively slow for high-poly meshes. There is a large body of work on acceleration of these algorithms, that try to reduce time. For the contours-within-faces case, one popular technique is to construct a hierarchical data structure, where each node stores a quiver of the normals below it. The most popular is hybrid approach. It consists of three main steps: Draw front faces in white, offset towards viewer. Draw enlarged back faces. As a result, the second rendering pass will “look out” from behind the geometry rendered previously on the first pass. Algorithm is fast and easy to implement, but it is applicable for silhouettes only.

Another useful image-space algorithm uses data from the depthbuffer (a color that depends on depth) instead of rendering $n \cdot v$. This approach makes the rendering simpler, but the image processing more complex. "Main work" of this technique lays on shoulders of pixel shader rather than vertex shader. That's why it can be treated as a screen space postprocessing effect. Our shader is a bit similar.

Main part
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 edge detection technique with the Laplacian operator [1]. 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. The basic algorithm is the follows:

- 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 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 we 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.

To encode z-values as rgb colors of the depth texture, we use well-known approach described in [4]. For integer RGBA textures, we need to pack the depth value and write it into the color channel. There are plenty of ways to pack depth into texture. We use the follows approach:

In the fragment shader: just to project 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) [5] to implement our shader. In AGAL syntax, it looks the following way:

\[
\begin{align*}
\text{m44 op, vt0, vc0} & \quad // \text{project vertex coords (vt0) by MVP matrix (vc0)} \\
\text{m44 vt1, vt0, vc4} & \quad // \text{project vertex (vt0) according to scene transform matrix (vc4)} \\
\text{sub v0, vt1, vc5} & \quad // \text{compute a vector (v0) from vertex to camera position (vc5)} \\
\end{align*}
\]

The "scene transform" matrix (vc4) is the transformation matrix that transforms from model to world space.

In the pixel shader we need to use the far clipping plane of a camera. To represent the 32-bit depth buffer, we need to multiply interpolated squared distance between camera and vertex to powers of two: 20, 28, 216, 224. 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 will look like:

\[
\text{dp3 ft0.z, v0.xyz, v0.xyz // get a squared distance (ft0) by computing squared length of v0} \\
\text{mul oc, fc0, ft0.z // pack the depth (ft0) by multiplying to powers of two (fc0)}
\]

In this code the register fc0 has the following values:

\[
\begin{align*}
\text{fc0.xyzw} = \left( \frac{1}{2^{16}}, \frac{1}{2^{16}}, \frac{1}{2^{16}}, \frac{1}{2^{16}} \right)
\end{align*}
\]

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):

\[
\text{max depth} = \frac{1}{\sqrt{2^2 + 2^2}} \approx \frac{1}{2}\text{f}^2
\]

Now we have the depth texture that is used for further computations. 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:

\[
\tilde{t} = (dx - 0.x, dy - 0.y, z - 0.z)
\]
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 which are in range [0..1] fo the whole screen. In our case, we use normalized “dx” and “dy” according to width and height of the screen texture:

\[
dx = \frac{du \times r}{\text{width}}; \quad dy = \frac{dv \times r}{\text{height}};
\]  

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”.

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 be look like different intensity of a resulting color. As a result, the most covered from the incident light places of the 3d scene will be shady, especially if the sampling step is greater then one pixel (r > 1). In this case, significant differences of depth will have dark halo. Despite of simplicity, the algorithm produces improper results in case if a single object of the scene is situated near from the camera, and all other objects are relatively far from the camera, one will be overshadowed too.

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, which will make too dark color values lighter. Then output color looks the following way:

\[
oC = dp \times (2 - \frac{dp}{L})
\]

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 (which equals 4) to set the output color in correct range. For clarity of algorithm, we have wrote the source code of the fragment shader in Adobe Pixel Bender language (see the “Appendix A” section 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. This depth map in less precise (8-bit), than the one, described above (our 32-
bit depth buffer). 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 (AGAL). This shader language is simple but yet powerful and consist of series of opcodes acting on registers, stored in GPU’s memory. AGAL code is less human-readable, but times faster, than the same codebase written in Pixel Bender. We used almost the same approach to rendering of 3d meshes, as in previous work of our colleagues [6]. But in this case, we should render the scene twice:

- Render the scene with its whole appearance (mesh textures, lights, etc.) to the BitmapData. BitmapData is standard class of the Action Script 3.0 programming language. It stores color data of a 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. Function “initBuffers” creates two triangles in GPU’s memory to render screen’s rectangle. Function “draw” draws its triangles with our post processing shader. This shader takes screen texture and depth texture (fs0, fs1) as input sources.

Final AGAL code has three modes:

- Render just a scene.
- Render the scene mixed with post-processing filter.
- Render just the filter.

In contrary of the “Appendix A”, the AGAL version of 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-lying objects neater.

In contrary of the “Appendix A”, the AGAL version of 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-lying objects neater.

**Fig. 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.

**Conclusions**
First, we checked the output of our shader written in Pixel Bender. We use a simple 8-bit grayscale depth map representing geometry of a cube without the inner central part. The result of a shader work is on the figure 3. On the left side is the depth map, which 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 is 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 work of the same shader, 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:

1) In physical world, the contrast is the difference in color that makes an object (or its representation in an image or display) distinguishable. In our case, we adjust the resulting color by the following formula: color=\[\text{color} \times \text{contrast}\].

“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.

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

Fig. 3. From left to right: test 8-bit depth map; Pixel Bender’s shader output working as ambient occlusion effect; Pixel Bender’s 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. For edge detection, AGAL version of our shader has slightly different parameters. The “range” parameter now 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. If you want to display only the outer contour of the object, it is sufficient to set this parameter times greater. This is the one of advantages of our shader filter: it makes possible to display both the outline of a model and its “inner” edges.

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 edges. 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 edges. Our shader can do these tasks simultaneously. Fig. 4 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). Nevertheless, final image looks good and has almost realistic lighting. This effect can be achieved with the same setting as for Fig. 4, but with low “r” (r=1.5) and low overall brightness.
Fig. 4. AGAL shader output for ambient occlusion and for edge detection. Rendering of architectural scene (interior of a room). Left: scene texture; Center: scene texture and our filter; Right: raw output of our filter (without mixing with the scene texture).

Being fully screen space post-processing effect, the shader is almost independent on geometric complexity of a scene, so its performance is mostly specified by screen resolution. For screen resolution 1280x1024 we achieve 45-60 frames per second for scenes of two million polygons.

Brightness corresponds to uniform brightness of the resulting picture. In our case it’s inverted brightness, so zero value points to bright picture and high values points to dark picture. We have presented real-time post-processing shader, which 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). This effect is flexible, adjustable and useful for rendering of industrial design, architectural works and engineering. The algorithm is simple to implement and can be easily extended to other higher-level shader languages. It uses only depth texture without screenspace normal texture and without additional noise texture, which is key component of a rendering pipeline of most of known screenspace ambient occlusion algorithms. Our algorithm forces to render the scene twice: standard render pass + contours extraction pass. The shader is fast enough, but to increase the speed of processing, we plan to use new feature of AGAL: multiple render targets, so we can directly construct a real G-Buffer, rather than rendering the scene to separate textures.

References
[1] Umbaugh S 2010 Digital image processing and analysis: human and computer vision applications with CVIPtools (2nd ed. ed.) Boca Raton, FL: CRC Press
[2] Langer M, Buelthoff H 2000 Depth discrimination from shading under diffuse lighting Perception 29 (6) pp 649–60
[3] Miller G 1994 Efficient algorithms for local and global accessibility shading Proceedings of the 21st annual conference on Computer graphics and interactive techniques pp 319–26
[4] Pharr M, Fernando R 2005 GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems) Addison-Wesley Professional (Chapter 12)
[5] Scabia M Vertex and Fragment Shaders http://www.adobe.com/devnet/flashplayer/articles/vertex-fragment-shaders.html
[6] 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. Eng. 69 012007
[7] Rusinkiewicz S 2004 Estimating Curvatures and Their Derivatives on Triangle Meshes Second International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT)
[8] Michael Burns, Janek Klawe, Szymon Rusinkiewicz, Adam Finkelstein, Doug DeCarlo 2005 Line Drawings from Volume Data In SIGGRAPH
[9] McGuire, Morgan and John F Hughes 2004 Hardware-Determined Feature Edges http://graphics.cs.williams.edu/papers/EdgesNPAR04/edges-NPAR04.pdf