Non-Photorealistic Real-Time Edge Rendering using Non-Duplicate Parallel Detection and Capping

BY
Dwight House
B.S. Computer Science, Louisiana State University Shreveport, 2007

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute of Technology
Redmond, Washington
United States of America

Summer
2010

Thesis Advisor: Dr. Xin Li

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS OF Dwight House

HAS BEEN SUCCESSFULLY COMPLETED ON May 14th, 2010

TITLE OF THESIS: Non-Photorealistic Real-Time Edge Rendering using Non-Duplicate Parallel Detection and Capping

MAJOR FILED OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

Dr. Xin Li, Chair	Dr. Gary Herron
Dr. Jason Hanson	Mr. Chris Peters

APPROVED:

Dr. Xin Li	date	Dr. Xin Li	date
Graduate Program Director		Dean of Faculty	
Mr. Samir Abou-Samra	date	Mr. Claude Comair	date
Department of Computer Science		President	

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

INSTITUTE OF DIGIPEN INSTITUTE OF TECHNOLOGY

PROGRAM OF MASTER'S DEGREE

THESIS APPROVAL

DATE: May 14th, 2010

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

Dwight House

ENTITLED

Non-Photorealistic Real-Time Edge Rendering using Non-Duplicate Parallel Detection and Capping

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE FROM THE PROGRAM OF MASTER'S DEGREE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

Dr. Xin Li
Thesis Advisory Committee Chair

Dr. Xin Li
Director of Graduate Study Program

Dr. Xin Li, Dean of Faculty

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

Acknowledgements

I would like to thank my family for their unending love, support, and encouragement. I thank also my friends who gave me moral support and assistance throughout the creation of this thesis. Thanks to Dr. Herron for his assistance in several key areas of my thesis study. Finally, I thank my thesis advisor Dr. Li for his guidance, patience, and kind words.

Table of Contents

1. Abstract

Edges are one of the most important illustrative constructions available in any media. These mere lines amplify structure, define boundaries, and provide style. Their utility has inspired researchers into real-time edge detection and rendering since the early days of computer graphics. There are dozens of diverse methods to achieve real-time edge detection and rendering.

In this thesis, the author first presents a survey of major edge detection methods, organized into four sub-categories. Then, the author proposes several contributions to an existing GPU edge detecting method. The final contribution is a complete restructuring of the original method to take advantage of a new technology. This method increases speed and accuracy while keeping the memory usage approximately the same.

2. Introduction

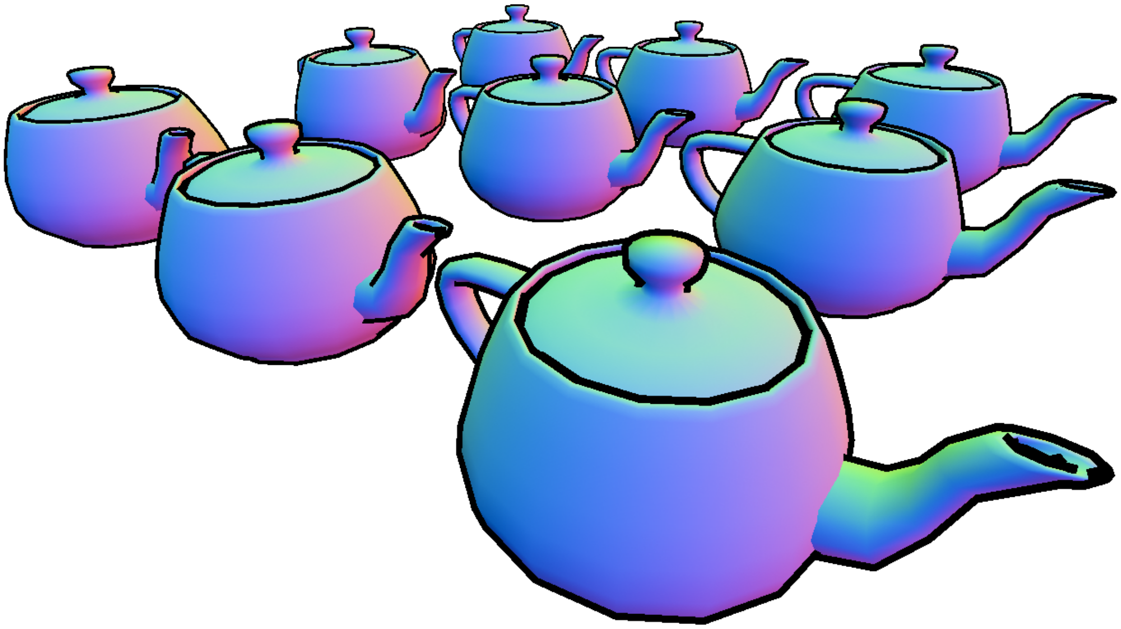


Figure 2-1: Several Utah Teapots rendered with depth-scaled object-space edge detection and a normal shader.

Non-photorealistic rendering (NPR) is any rendering with an intent other than the creation of photorealistic imagery. The power and customizability of more recent graphics card hardware has made NPR in real-time a reality in popular media and games. One structure that spans nearly all forms of non-photorealistic rendering is the humble edge. Edges have surprising expressive power for their relative simplicity. Using edges, inner object structure and the differentiation of nearby objects can be amplified and stylized. The following images are some examples of games that use edges and other effects in their non-photorealistic rendering.

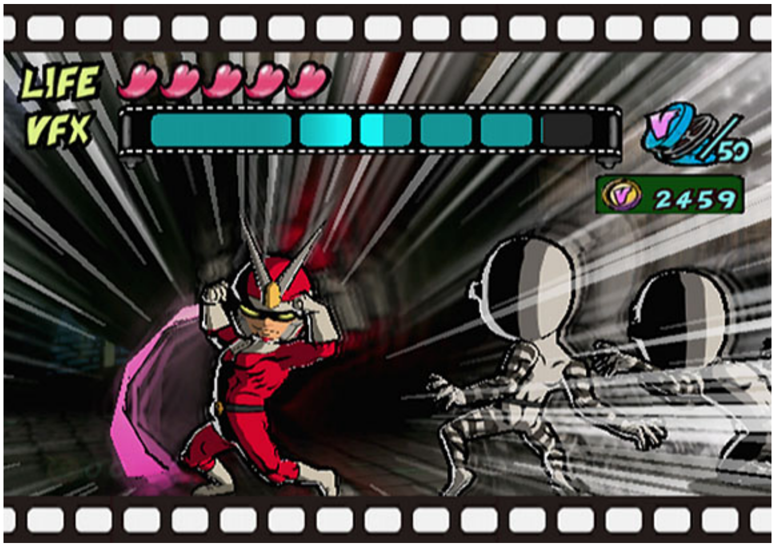


Figure 2-2: Viewtiful Joe (Capcom Production Studio 4) uses edge detection, cell shading, and various movie and anime-inspired animation effects to achieve an over-the-top graphical experience to match the chaotic nature of the gameplay. (Image courtesy of The Next Level)

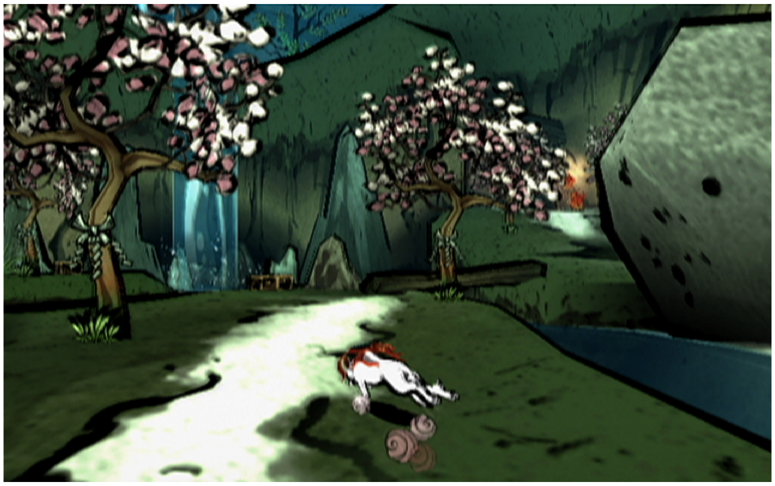


Figure 2-3: Ōkami (Ready at Dawn) makes use of a simpler, more sketchy edge detection along with screen-space canvas texturing and blurred cell-shading to create the visual appearance of a painting. (Image courtesy of Binge Gamer)

There are many methods of detecting and rendering edges. The author has chosen to limit the topic only to real-time edge detection on polygonal meshes. The author also focuses on the detection of edges over their subsequent rendering. Even with these restrictions, dozens of methods and considerations remain.

The practical applications of edges are detailed and the edge types are defined in the remainder of the introduction. Next, a survey of edge detection and rendering methods is presented. The dozens of methods are broken down into four sub-categories: hardware, image-space, object-space, and miscellaneous methods. Then, the details of a paper by Morgan McGuire and John F. Hughes on GPU accelerated edge detection are explained because they form the basis for the author's contributions. With that introduction, the author's provides his contributions, including a completely new edge detection and capping method. The thesis concludes with a comparative analysis of the two methods.

2.1. Applications of Edge Detection

Edge detection and rendering are used primarily for graphical styles and to provide structural and positional differentiation. A list of known edge applications is below.

- **Differentiate objects** - An outer border on an object can differentiate it from other objects and the rest of the scene in the absence of shading. With shading, it enhances the contrast between objects.
- **Differentiate sections of objects** - An edge can serve to divide up various sections of a single object. For example, edges outlining a character's clothes will greatly differentiate the clothes from the character's skin and face.
- **Enhance structural perception** - Within the object, the edges can give a greater weight to a specific part of the mesh, such as the nose or eyes of a face. Without shading, edges are fully capable of providing basic structure to the mesh.
- **Highlighting** - If one object needs to be in focus due to selection or other importance, edges on the object can help differentiate it from all other objects.
- **Anti-Aliasing** - Rendering systems typically blend colors and textures across the surface of the polygon meshes, but on the edges of these meshes, aliasing occurs because at any given time, the renderer does not know what objects will occupy a given pixel or the neighboring pixels. Edge detection allows a focused application of a blur, which significantly reduces the aliasing artifacts. This is a common method of anti-aliasing when using deferred shading.

- **Achieve specific graphical styles** - A wide variety of visual styles require edge detection as one of their major components. Such visual styles include: cell shading, technical and architectural illustration, pencil/pen styles, some forms of fur rendering, and several others.

2.2. Types of Edges

There are only a handful of edge types, but the terminology varies from book to book and paper to paper. For this thesis, the author will mention the variations in wording in the title of the types below, but will use the first mentioned name as the standard term for each type. Additionally, for the purposes of this thesis, all models and edges are described in terms of polygons. Edge detection and rendering specifically for non-polygon objects is beyond the scope of this thesis.

Contour/Silhouette/Outline Edge

Contour edges are the polygon edges for which one adjacent polygon is front-facing and the other is back-facing. Contours are view-dependent and animation dependent. They cannot be pre-calculated, though some methods can use preprocessing to increase the speed of the test.

There is often need to differentiate the outer contour that surrounds the object from internal contours. Some visual styles prefer to have a thicker edge on the outer contours in order to reinforce the difference between each object. In this paper, the author chooses to call this subtype “silhouette.” There is a lot of terminological conflict about this term. Many papers refer to all contours, internal and external, as silhouettes.

Crease/Hard/Feature Edge

Crease edges represent discontinuities in the surface of a mesh. A polygon edge is a crease edge if the angle between the adjacent polygons’ normals is greater than a user-defined “dihedral” angle [Gooch and Gooch 01]. Some implementations choose to split crease edges into two sub-categories, ridge and valley creases [McGuire and Hughes 04].

- **Ridge Crease Edge** - A crease edge where the internal angle between the two adjacent polygons is less than a user-defined ridge dihedral angle.

- **Valley Crease Edge** - A crease edge where the external angle between the two adjacent polygons is less than a user-defined valley dihedral angle.

Crease edges are not view dependent, but they can change based on animation.

Boundary/Border/Surface Boundary Edge

Boundary edges only occur on non-closed models [Gooch and Gooch 01]. It is a polygonal edge of only one polygon. These edges are not view-dependent and do not change with animation. Thus, they can be pre-calculated with no ill effects.

Intersection/Self-Intersection/Collision Edge

Intersection edges occur when two polygon surfaces pass through each other such that their intersection line does not correspond to a polygon edge. This is the most uncommonly mentioned edge type because of its relative rarity and difficulty to detect with most methods. Some papers ignore these edges and others define them as self-intersections, only applying them to meshes that pass through themselves. No paper the author has read has accounted for these types of edges formed by the intersection of polygons from two different meshes, though the concept is identical. Self-intersection edges may be pre-calculated as long as they do not animate. Intersection edges between different objects can only be pre-calculated if both objects do not animate or move relative to each other.

Marked/Material Edge

Marked edges are merely polygon edges that the user has defined to always be a drawn edge. These edges are not view-dependent or animation-dependent, because they are pre-calculated by definition. Often they are used to separate major sections of a mesh regardless of orientation.

Drawable Edge

This term is the author's. It is used to differentiate any polygon edge from polygon edges that should be drawn on a given frame. It can be used to refer to any other type of edge, except intersection edges. This differentiation is used to avoid ambiguity when discussing object-space edge detection.

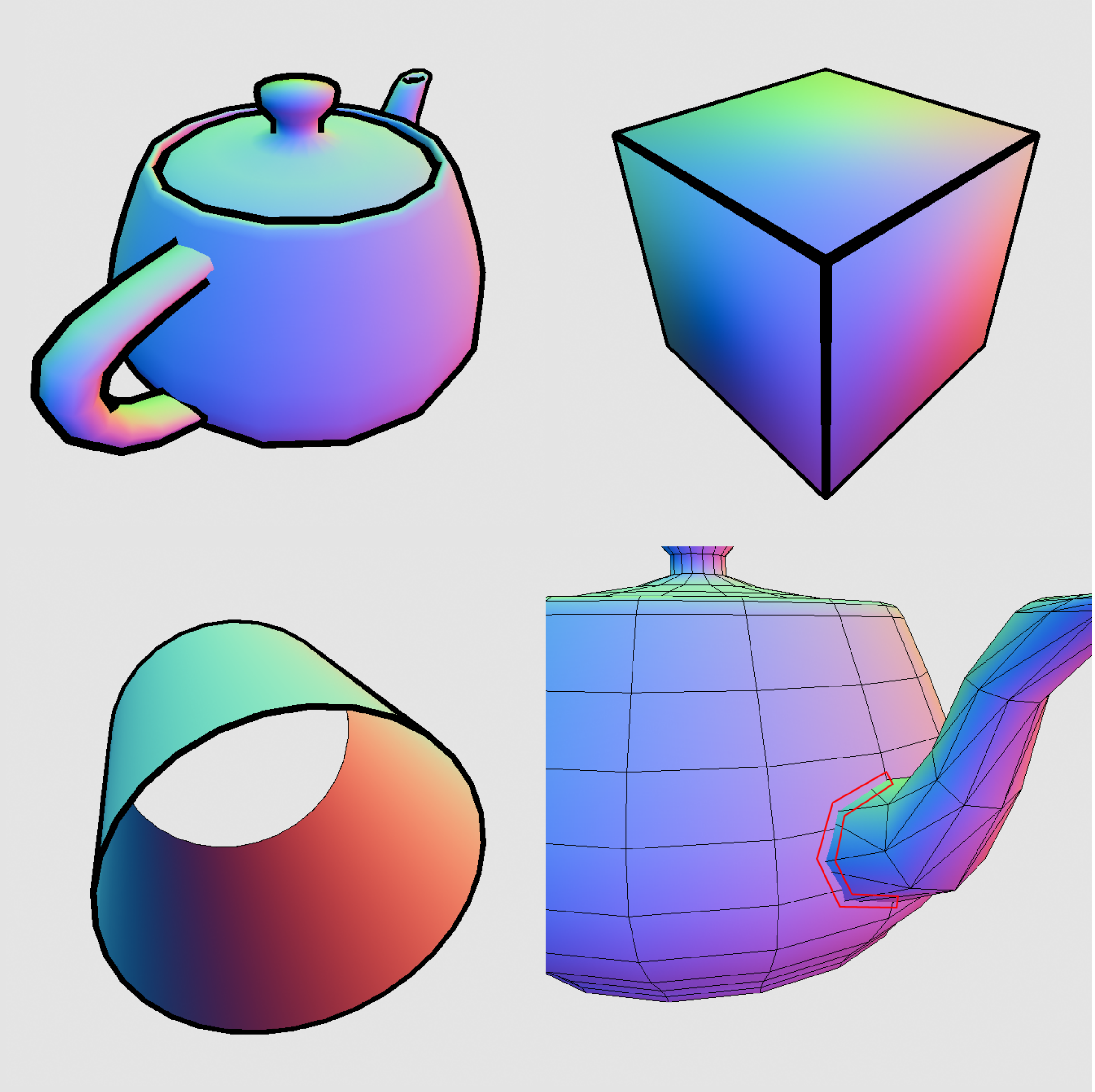


Figure 2-4: The teapot on the top-left shows contour edges. Notice how contour edges can be internal and external (silhouettes). The cube on the top-right is made up of creases (inner edges) and silhouettes (outer edges). The edges on the two ends of the tube on the bottom-left are boundary edges. The bottom-right image shows that the polygons of the teapot do not correspond to the intersection edge highlighted in red.

3. Edge Detection Method Overview

There are dozens of ways to detect and render edges in real-time. They all fit into roughly four categories: hardware, image-space, object-space, and miscellaneous methods. Hardware methods are quick and simple to implement, but with severe limitations. Image-space methods are a bit more involved and provide medium quality results. Object-space methods provide maximum quality and customizability, but often at the cost of speed. The miscellaneous category is a catch all, but these methods tend to be quick and limited like hardware methods. The following sections survey significant methods in each category.

4. Hardware Methods

Hardware-based edge detection and rendering covers a wide variety of different methods. They all lack the need for mesh pre-processing, allowing them to be highly adaptable to new scenes and mesh animation. Edge detection is achieved as a by-product of their rendering modes and order, rather than a specific edge detecting step. They represent the first and simplest available methods of interactive edge detection.

Unfortunately, they are very limited in terms of customizability. They typically generate single pixel edges for which only the color can be changed. Unless combined with other techniques, they generally only detect contour edges.

4.1. Back-Facing Wireframe Contours

Jarek R. Rossignac and Maarten van Emmerik [Rossignac and Emmerik 92] described how to generate contour edges in their description of their Contour Edges with Hidden-lines Suppressed (CEHS) method. It only displayed an mesh's edges, not the mesh itself.

The part of the algorithm that deals with contour edges is:

1. Render the object's polygon into the z-buffer (nothing drawn to screen)
2. Shift depth buffer backwards
3. Increase the thickness of line rendering
4. Render the polygon in wireframe mode
5. Return the depth offset to its original position
6. Return line rendering to normal thickness

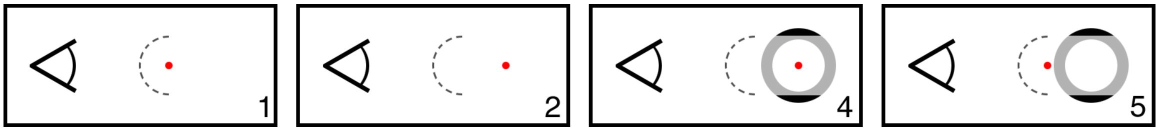


Figure 4-1: In step 1, the sphere mesh is rendered to the depth buffer (the dashed grey line) around the object origin (the red dot). In step 2, the origin is shifted back slightly. Step 4 renders the sphere's polygons in wireframe mode at the new origin using thick edges. The depth buffer's values prevent most of the edges from being rendered (grey areas). Step 5 shifts the origin back, preparing the scene for the next render.

By writing values to the depth buffer before shifting backwards, most edges are prevented from being drawn in step 4. Since the lines are rendered with additional thickness, lines on contour edges will poke out from behind the invisible "depth mask."

The color of these edges can be customized as desired. The thickness of the edge can also be controlled with the thickness value applied to the lines in step 3. Because line thickening does not round the ends of the lines it draws, sufficient thickness will result in gaps between adjacent edges. This can be alleviated by rendering a third pass where every vertex is rendered as a point, each thickened to the same degree as the edges.

Bruce and Amy Gooch [Gooch and Gooch 01] described a variation on Rossignac and van Emmerik's method.

The modified algorithm is:

1. Enable back-face culling
2. Set depth function to "Less Than"
3. Render the mesh in fill polygon mode (front-facing polygons only)
4. Enable front-face culling
5. Set depth function to "Less Than or Equal To"
6. Render the mesh in wireframe mode (back-facing polygons only)

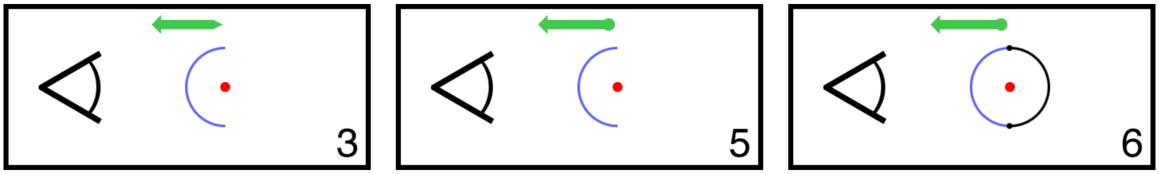


Figure 4-2: The green arrow represents the state of the depth function. The blue line represents the front-facing polygons and the black line represents the back-facing polygons in wireframe mode. The dots in step 6 represent points on the model where the depth function will overwrite front-facing pixels with back-facing pixels, creating the contour.

This variation favors changing how the depth buffer deals with values at the same depth, rather than offsetting the whole mesh. As a result, even though the back-facing pixels could be considered behind the previously rendered front-facing pixels, by default an edge with a maximum width of one pixel will still render. Gooch and Gooch’s modified method is faster than Rossignac and van Emmerik’s method, because only half of the polygons are rendered in each pass.

One final variation combines the ideas of both above variations. Steps 2 and 5 of Bruce and Amy Gooch’s method can be ignored if one desires contour edges that do not overlap the mesh at all. In this variation, the line thickness must be set greater than one to have any visible effect on the final image.

4.2. Back-Facing Filled Polygon Contours

Ramesh Raskar and Michael Cohen [Raskar and Cohen 99] discussed several hardware-based methods of edge detection. Their first attempt was a step backward from the previously mentioned technique, but it was needed to reach their final, advanced technique.

The algorithm is virtually identical to Bruce and Amy Gooch's modifications above, except that wireframe rendering is replaced with filled polygon rendering. However, Raskar and Cohen’s method was published first.

The algorithm is:

1. Enable back-face culling
2. Set depth function to "Less Than"
3. Render the mesh in fill polygon mode (front-facing polygons only)
4. Enable front-face culling
5. Set depth function to "Less Than or Equal To"
6. Render the mesh in fill polygon mode (back-facing polygons only)

Because the polygons are not scaled and cannot be thickened the way that lines can, rendered contour edges using this technique will be a maximum thickness of one pixel. Contour pixels may be missed or flicker due to z-buffer quantization and pixel sampling.

4.3. Offset Back-Facing Filled Polygon Contours

Raskar and Cohen mentioned another technique that would allow for thicker edges. It uses depth offsets to force the back-facing polygons toward the camera, pushing through the surface of the front-facing polygons at contour edges. More of the back-facing polygons' area will be rendered on top of the front-facing polygons than in previous methods.

The algorithm is:

1. Enable back-face culling
2. Set depth function to "Less Than"
3. Render the mesh in fill polygon mode (front-facing polygons only)
4. Enable front-face culling
5. Set depth function to "Less Than or Equal To"
6. Offset render depth towards camera a defined distance
7. Render the mesh in fill polygon mode (back-facing polygons only)
8. Return depth offset to normal

The thick edges this technique allows come at a dramatic cost. Since the back-faces are pulled through the front-faces, the orientation of the front-facing surface will determine the thickness of the edge. As a result, the overall thickness of the edge on the mesh will be uncontrollable and inconsistent for many meshes. Unlike some previous methods, any thickness gained is at the expense of rendering the edge on top of the front-facing surface of the mesh. This effect may be desired, however.

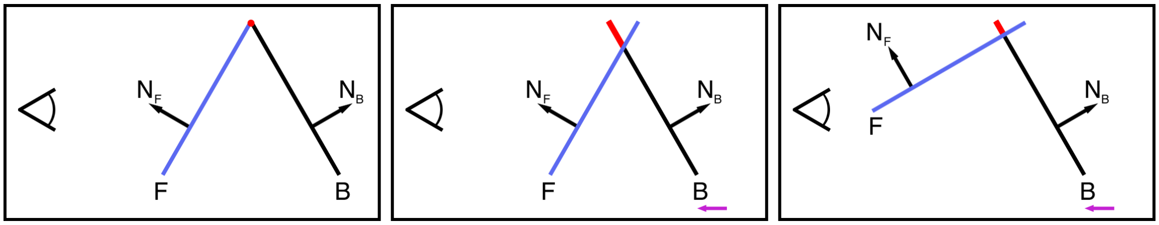


Figure 4-3: The leftmost image shows the actual layout of the front (F) and back (B) facing polygons, and their respective normals. The red dot shows the location of the actual contour edge. In the second image, shifting the back-facing edges forward a distance equivalent to the purple arrow, they become the contour edge. However, the inconsistency with this method is demonstrated on the rightmost image: different front-facing polygon orientations will generate different thicknesses.

4.4. Thickness-Normalized Back-Facing Filled Polygon Contours

As one final step, Raskar and Cohen suggested another method to solve the inconsistent thickness problem while reducing the gapping problem of Rossignac and van Emmerik's wireframe thickening technique.

Assuming all mesh polygons are convex, the back-facing polygons' edges will be expanded outward a distance determined by their angle and distance from the camera. The result is that the actual edges are shifted out from behind the front-facing polygons the exact distance necessary to create a thickened border that is equally thick on all sides in screen-space. They call this process “polygon fattening.”

Each edge should be shifted by:

```
(z * sin(α)) / dot(E, N)
```

In the direction:

```
cross(E, N)
```

Where z is the distance from the edge midpoint to the camera, α is the angle between the edge vector and the view vector, V is the view vector, N is the polygon face normal, and E is the edge vector.

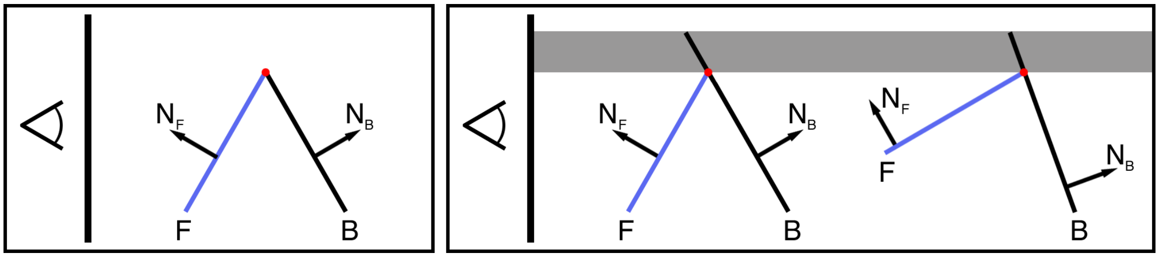


Figure 4-4: The shifted polygon edge maintains the same screen-space thickness despite the orientation of the polygons making up the edge.

However, this action of shifting the edges outward removes their connectivity. Without reconnecting them, they will not render as polygons. This is handled by creating new polygons that fit the shape of the outer edges as a triangle fan. This process causes the "capping" effect. Every back-facing polygon should be replaced with a set of 2N polygons, where N is the number of edges in the original polygon. The initial point to which all fan triangles connect should be the center point of the polygon.

The algorithm could be implemented:

1. Enable back-face culling
2. Render the mesh in fill polygon mode (front-facing polygons only)
3. Enable front-face culling
4. Loop through each back-facing polygon
    a. Begin rendering triangle strip, using the polygon's center point as the initial point
    b. Loop through each edge in order
        i. Shift the edge outward
        ii. Render both points to the triangle strip

It should be noted here that the gaps, while dealt with, are typically not completely filled as one might expect. How well the capping works is dependent on the desired thickness of the edge and the angle between each set of two connecting contour edges.

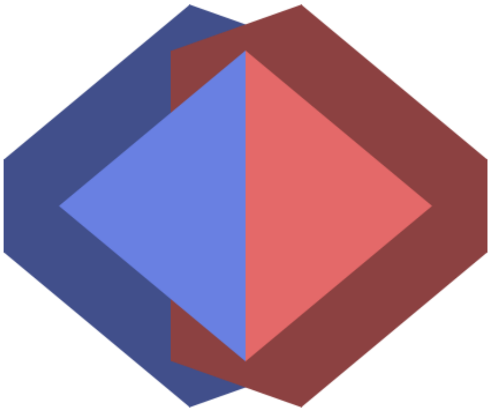


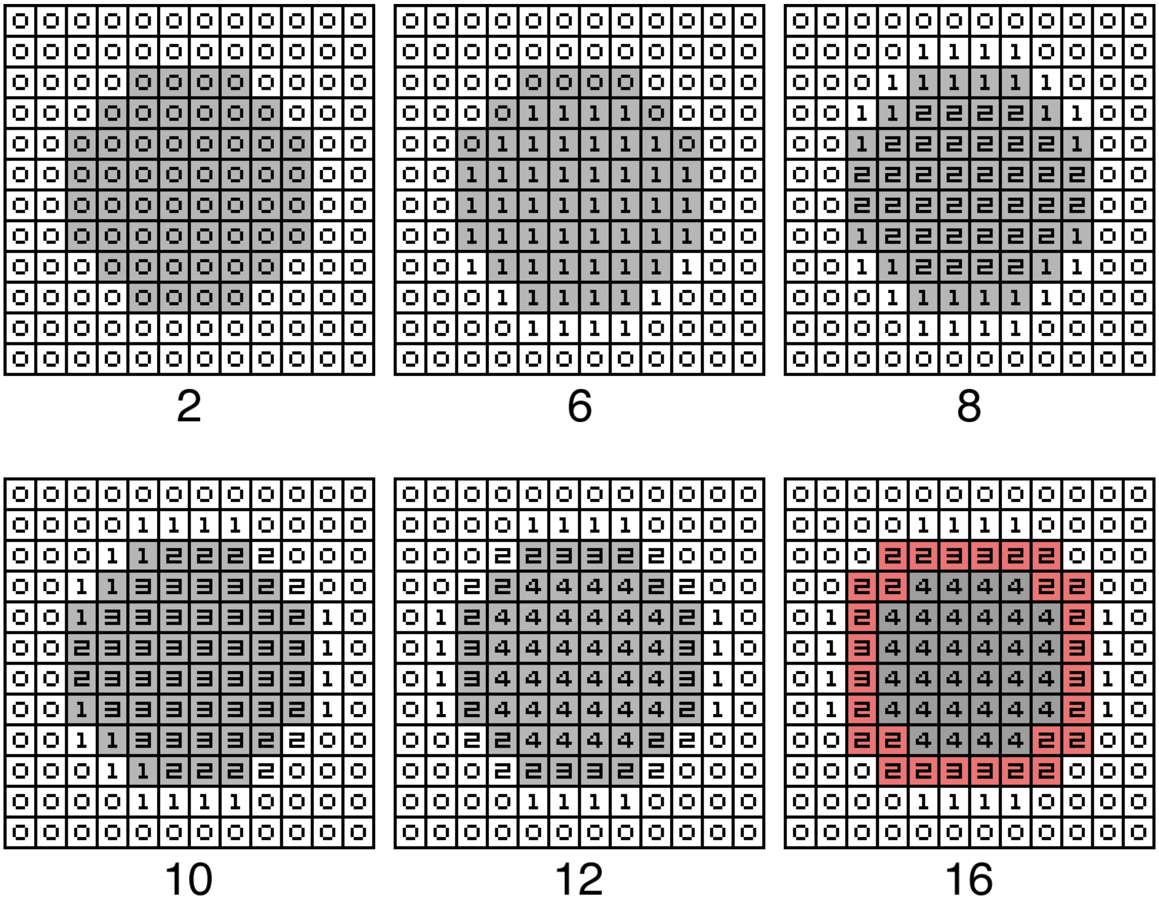
Figure 4-5: Demonstrating edge fattening and it's effects on capping. The brighter areas represent actual polygons from which the darker areas are calculated. Along contour lines, the fattened back-facing polygons will partially stick out from behind the front-facing polygons.

4.5. Stencil-Based Pixel Shifted Silhouettes

Tom McReynolds and David Blythe [McReynolds and Blythe 99] described a method that renders the object multiple times to the stencil buffer, each time at a slight offset on the x and y axes. By rendering an edge-colored quad over the object with certain stencil pass rules, the silhouette edges of the previously rendered object will appear.

The algorithm is:

1. Render the shaded mesh, if desired
2. Clear stencil buffer to zero
3. Disable color and depth buffers
4. Set stencil buffer to always pass, the operation to always increment
5. Shift the viewport by one in the positive y direction
6. Render mesh
7. Shift the viewport by two in the negative y direction
8. Render mesh
9. Shift the viewport by one in the positive x direction and one in the positive y direction
10. Render mesh
11. Shift the viewport by two in the negative x direction
12. Render mesh
13. Shift the viewport by one in the positive x direction (original position)
14. Enable color and depth buffers
15. Set stencil buffer to pass at two or three (the second bit equal to 1)
16. Render edge-colored quad that covers the mesh



2

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

6

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	0	0
0	0	0	1	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

8

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0
0	0	1	1	2	2	2	2	1	1	0	0
0	0	1	2	2	2	2	2	2	1	0	0
0	0	2	2	2	2	2	2	2	2	0	0
0	0	2	2	2	2	2	2	2	2	0	0
0	0	1	2	2	2	2	2	2	1	0	0
0	0	1	1	2	2	2	2	1	1	0	0
0	0	0	1	1	1	1	1	1	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

10

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	1	1	2	2	2	2	0	0	0
0	0	1	1	3	3	3	3	2	2	0	0
0	0	1	3	3	3	3	3	3	2	1	0
0	0	2	3	3	3	3	3	3	3	1	0
0	0	2	3	3	3	3	3	3	3	1	0
0	0	1	3	3	3	3	3	3	2	1	0
0	0	1	1	3	3	3	3	2	2	0	0
0	0	0	1	1	2	2	2	2	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

12

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	2	2	3	3	2	2	0	0	0
0	0	2	2	4	4	4	4	2	2	0	0
0	1	2	4	4	4	4	4	4	2	1	0
0	1	3	4	4	4	4	4	4	3	1	0
0	1	3	4	4	4	4	4	4	3	1	0
0	1	2	4	4	4	4	4	4	2	1	0
0	0	2	2	4	4	4	4	2	2	0	0
0	0	0	2	2	3	3	2	2	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

16

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	2	2	3	3	2	2	0	0	0
0	0	2	2	4	4	4	4	2	2	0	0
0	1	2	4	4	4	4	4	4	2	1	0
0	1	3	4	4	4	4	4	4	3	1	0
0	1	3	4	4	4	4	4	4	3	1	0
0	1	2	4	4	4	4	4	4	2	1	0
0	0	2	2	4	4	4	4	2	2	0	0
0	0	0	2	2	3	3	2	2	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Figure 4-6: The above illustrations show the state of the stencil buffer after each corresponding step. The grey area represents the actual drawable mesh pixels. In the last frame (16), the red indicates which pixels would form the edge.

This method is relatively slow because the scene must be rendered at least four times per frame.

4.6. Inverted Stencil Contours

A faster, less accurate method of stencil-based edges by K. Akeley was mentioned in McReynolds and Blythe’s class. It only requires one render to the stencil buffer and uses wireframes rather than filled polygons.

The algorithm is:

1. Clear stencil buffer to 0
2. Set stencil function to always pass and the operation to invert
3. Enable back-face culling
4. Render mesh in wireframe mode (front-facing polygons only)

5. Set stencil function to pass if the value is 1
6. Render edge-colored quad that covers the mesh

This works because wireframe edges that are part of two adjacent polygons will be drawn twice, inverting their stencil pixels back to zero. However, the pixels on the polygon edges between front and back-facing polygons will only be drawn once, and will have a stencil buffer value of 1.

This technique will also generate front-facing boundary edges, but not back-facing boundary edges. McReynolds and Blythe suggested that the technique be combined with the drawing of all visible boundary edges to complete the edges. They blocked the inclusion of invisible boundary edges by first rendering an offset depth buffer that would prevent covered edges from rendering. This method is also susceptible to stencil rendering interference if the whole scene is processed for edges at once.

4.7. Hardware Creases and Contours

Ramesh Raskar [Raskar 01] released a paper describing a hardware method to render crease edges in addition to contours. Though the technique takes advantage of the geometry shader and a secondary depth-buffer, it does not use a specific edge-detecting step.

The contours are rendered by expanding all back-facing polygons a user-defined distance, based on the screen-space projection width, similar to Raskar's previous method. After shifting all of these polygon's edges outward, the expanded edges are then reconnected into solid polygons and rendered black to create the contours.

For creases, the ridge and valley varieties are created differently. The ridge creases are created by generating a new quad with a length and location equivalent to each front-facing polygon edge in the geometry shader. When it is first created, this new quad should be flush with the polygon surface. Before sending it to the remainder of the pipeline, it should be rotated the user-defined dihedral angle about the edge vector, and thickened the same way as the contour edges so that all edges have a consistent thickness.

Since the quad is rotated the dihedral angular distance from the originating polygon, these edges will only appear if the adjacent polygon connects with an angle sharper than the one defined by the user, creating the crease edge. This can create z-fighting if the user's dihedral angle is equivalent to the angular distance of two connected polygons.

Valley creases are far more complicated to create. They, like the ridge creases, use a user-defined dihedral angle to determine where to create expanded polygon edges as quads. However, the visibility requirements for valley edges are the reverse of those needed for the ridge creases. To solve this, Raskar created a dual depth buffer. With two depth buffers, the created valley crease quad will only be allowed to modify the final image if it renders in-between the two depth values.

Finally, Raskar's method defined a way to detect self-intersection edges using the dual depth buffer technique again.

5. Image-Space Methods

Image-space edge detection and rendering utilizes image processing techniques to detect discontinuities in one or more rendered images. This operation may have variations in what image(s) is used as input, what type of image processing is used, and how the output information is utilized.

5.1. Image Processing with the Sobel Operator

For edge detection, image processing techniques typically apply convolutions to various properties of images or specialized images to generate gradients, a vector representing the local maximum change. In image processing, convolutions take the form of component-wise matrices called kernels or operators. These operators store numbers in each cell which are multiplied by some numerical property of the equivalent pixel in the input image. This operation is applied to every pixel in the input image, with optional special cases at the image borders where input data may not be available. The resulting output image data can be used in further calculations or displayed directly.

For each pixel (x) there are surrounding pixels (A, B, C, D, E, F, G, and H) except at the extreme borders of the viewport. The layout is:

$$\begin{bmatrix} A & B & C \\ D & x & E \\ F & G & H \end{bmatrix} = L$$

L represents the input pixels that will be used in conjunction with a kernel. Since a pixel is typically made up of color and alpha transparency information, the user must decide what property of the input image pixels he will use. The two most common types of information used will be described later.

The Sobel operator [HIPR 00] will be used to illustrate how operators are applied to input images. The Sobel operator detects gradients in the horizontal and vertical axes separately. The typical Sobel operators are:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = Gx \qquad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = Gy$$

When one applies the Sobel operator to a given pixel, a component-wise multiplication is applied and then the inner terms are summed:

```
LGx = (-A) + C + (-2D) + (2E) + (-F) + H;
LGy = A + (2B) + C + (-F) + (-2G) + (-H);
```

The LGx and LGy terms can be used to determine the direction of the gradient, but for the purposes of edge detection, only the magnitude of the gradient is needed.

The gradient magnitude is calculated:

$$|G| = \sqrt{LGx^2 + LGx^2}$$

However, for most purposes, the gradient magnitude can be approximated by:

$$|G| = |Gx| + |Gy|$$

The gradient magnitude can be used directly in edge detection. By drawing black pixels where the gradient magnitude is greater than or less than a user-defined value, edges will be rendered. The process of choosing which pixels to draw based on the gradient (or any value) is called thresholding.

5.2. Other Operators

The kernel most often used by real-time graphics applications is the Sobel operator. However, in image processing literature, there are several other edge-detecting operators. Below is a list of other operators from Wikipedia and Hypermedia Image Processing Reference [HIPR 00] that might be used.

Prewitt Operator

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = Gx$$

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = Gy$$

Scharr Operator

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} = Gx$$

$$\begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} = Gy$$

Robert's Cross Operator

Robert's Cross Operator works with a smaller data set and is therefore less accurate. The unusual size of the kernel means that the "central" pixel is on the upper left-hand side. Otherwise, it operates under the same principle as the other operators.

$$\begin{bmatrix} x & A \\ B & C \end{bmatrix} = L$$

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = Gx$$

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = Gy$$

Operator Variations

There are several other versions of the Sobel operator with different numbers, dimensions, and calculation schemes. These variations are meant to gain various accuracy or detection abilities not available with the version of the Sobel operator described above [Kroon 09]. Variations exist for the other operators as well. These variations are beyond the scope of this thesis.

5.3. Input Imagery

In traditional image processing, the input image is typically a grayscale version of the image for which edges are desired. A grayscale image provides a single value from zero to one that represents the intensity at each pixel. Dealing with image noise and non-edge areas of high gradient magnitude are very difficult to overcome with only the original image available. Real-time graphics provide more flexibility. While the scene could be rendered in grayscale form and then used for edge detection on a second pass, edges can be detected with more accuracy using visualizations of scene information.

Depth Buffer Usage

Takafumi Saito and Tokiichiro Takahashi [Saito and Takahashi 90] used the scene depth information as their input image. On the first pass, the scene's depth information is rendered and retrieved. Variations of the Sobel operator are used to obtain first and second-order differential data. After thresholding and other error adjustment, the resulting information is rendered to create an edge image. They also explored methods of using the edge information to enhance the fully shaded image beyond merely overlaying the edges.

The difficulty with using only the depth information for the edge test is that while depth buffers can be used to detect contour edges, they are not very good for use in detecting crease and intersection edges, as surrounding pixels have very similar depth information. Additionally, depth buffer resolution limitations can prevent the adequate detection of edges on objects at similar depths.

Normal Buffer Usage

Philippe Decaudin [Decaudin 96] suggested using the scene's normal information to supplement the depth buffer information. Normals rendered as colors very distinctly show crease and intersection edge discontinuities. Most edges can be found using both gradient magnitudes together.

Since a normal is a 3D vector, the user may choose to do the edge detection on all three dimensions of the normal and then sum the resulting modified values. Otherwise, a one dimensional component of the normal buffer at each pixel is required. One simple possibility is the dot product of the view and normal vector. Another is the grayscale version of the normal buffer, which can be found by summing of the normal's axes and then normalizing the result. As with the depth buffer, the resulting gradient image must be thresholded to get only the desired edges.

5.4. Modern Implementation

When Saito & Takahashi and Decaudin wrote their respective papers the graphics hardware available required them to transmit the rendered images back to the CPU each frame. While the depth information could be retrieved directly, the normal color information had to be created by rendering the scene twice with special lighting conditions before the image was transmitted back to the CPU for image processing. Once the image was processed, it had to be sent back to the graphics card where the edge information and other shading were combined to create the final image.

Modern graphics cards provide programmable shaders and framebuffer objects. The shaders allow for the more specific and easier creation of depth and normal imagery. The frame buffer objects allow the graphics card to create offscreen renders that can be held over into subsequent frames without interfering with imagery rendered to the screen. With these two improvements, image-space edge detection can be accomplished with two rendering passes without any significant data transfer to or from the graphics card. [Card and Mitchell 02]

In the first pass, all the scene's geometry should be rendered with a special shader that does not send its data to the screen. Instead, the fragment (pixel) shader calculates normal information at each pixel from the vertex normals. The 3D normal information is stored in the RGB components of a framebuffer object created prior. The depth information, whether calculated manually or not can be stored in the alpha channel of the same framebuffer object that contains the normal information. Optionally, additional framebuffer objects can be created to hold other information such as the shade color at each pixel.

In the second pass, a screen-aligned quad with UV coordinates that prevent wrapping is rendered instead of the scene. In the shader, the UV coordinates of the quad are used as lookups into the framebuffer object rendered in the previous frame. At this point, the edge detection methods are performed at each pixel for both the normal and depth information. Generally, the sum of the edges found in both sets of information are used together to represent all edges, though there is plenty of room for customization. Finally, the edge information is combined with any other shading information rendered in the previous pass to create the final output color for each pixel. The output of these operations will be an edge or edge-enhanced image.

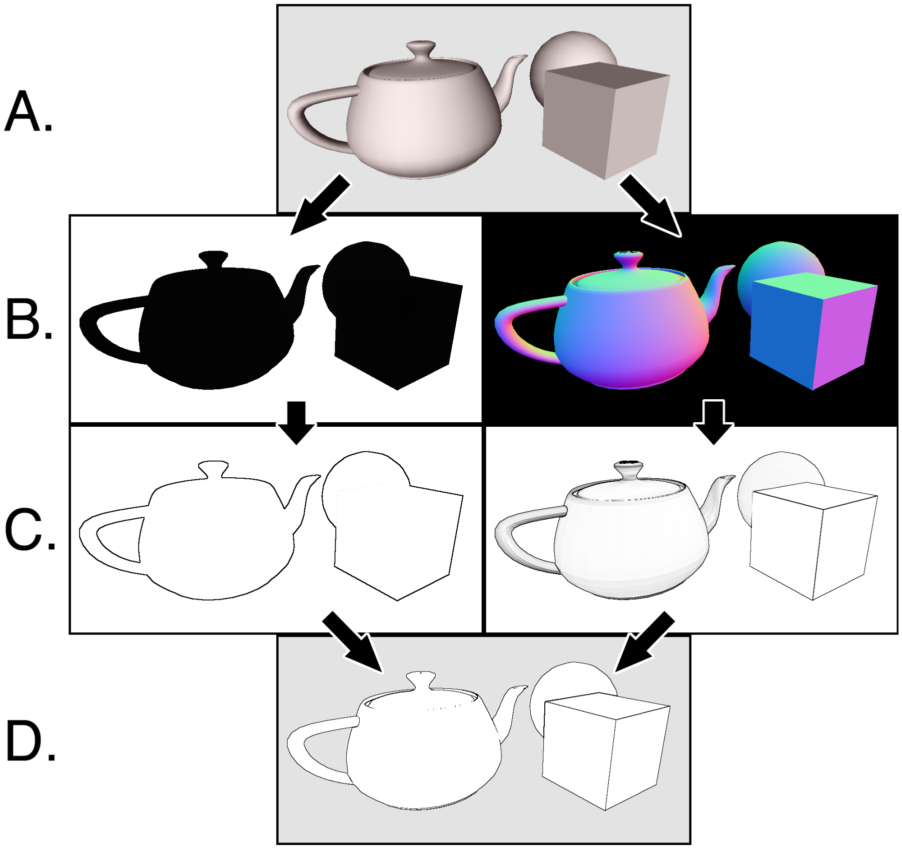


Figure 5-1: A shaded scene (A) may be rendered using the image-space edge detection by doing a pass where depth (left B) and normal (right B) information are rendered to framebuffers. In the next pass, while rendering a screen-sized quad, the two framebuffers are used by an image processing operator to get their respective gradient magnitudes (C). The gradient magnitudes are thresholded and used to create the final output (D).

5.5. Technique Variations

Image-space edge detection is often the method of choice when detecting edges for the purpose of anti-aliasing because they overlap both sides of the edges detected. For this use, areas determined to be edges are used as a mask for a blur operation. This creates an inexpensive anti-aliasing effect that is especially well-suited to deferred shading.

Additional scene information could be used to further augment the edge detection. For example, providing every object a unique color would allow silhouette edges to be detected even of the objects shared nearly identical depths and surface normals in the edge areas. Unique colors could also be given to distinct object faces to prevent missed edge cases where an object is folded over itself, as described in Aaron Hertzmann's paper [Hertzmann 99].

5.6. Advantages and Disadvantages

Image-space edge detection has two distinct advantages over other edge detection techniques:

- Constant edge detection speed
- Natively detects intersection edges

Because the number of edge detecting operations is directly dependent on the screen size, rather than the number of objects, the edge detecting step of the second pass will perform at the same speed no matter how simple or complex the scene is. The first pass is still limited by the hardware and how complicated the scene is, however.

Intersection edges are discontinuities that don't correspond to polygon edges. However, in image-space, intersections edges are merely another variation of a surface discontinuity.

The disadvantages are:

- Lack of customizability
- Missed edges

- False positive edges
- Relatively high tweaking required to get accurate results

The thickness of the edges can’t be directly controlled. The only way to get thicker edges is to apply further image processing or do the edge detection of buffers that are larger than the output resolution. Though these edges may be colored, only screen-space texturing is possible.

Depending on the threshold settings, legitimate edges can be missed entirely. The thresholding might also create false positive edges, where the rate of change, while smooth, was too great to be ignored by the image processing.

The thresholding values must be tweaked by the user to achieve the desired results. These threshold values are sensitive to the object, the rendering purpose, and the user’s own preferences. More trial and error is required for image-space edge detection than for other techniques.

6. Object-Space Methods

The object-space method of edge detection seeks to identify drawable edges in three dimensional space. At its most basic, all unique polygon edges are checked for drawability given a camera position and other factors. All variations on the basic idea are implementation details, supplemental features, and attempts to significantly reduce the number of tests necessary.

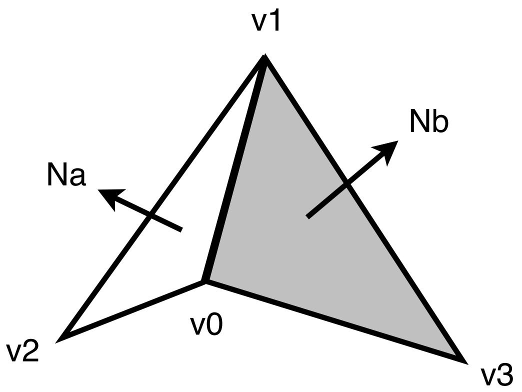


Figure 6-1: A single polygon edge connecting two adjacent polygons. Na and Nb are the face normals for the two adjacent polygons. v0 and v1 vertices define the edge. v2 and v3 complete the two adjacent polygon's vertices. The right polygon, and therefore the v3 vertex, may not exist. If so, the edge is a boundary edge.

6.1. Drawable Edge Tests

The definition of each edge type defines a simple formula for determining if a polygon edge is drawable from a given perspective. Refer to figure 6-1 for a graphical reference of the terms used below.

Contour Edge

```
(dot(Na, Nb) * dot(Nb, V)) <= 0
```

Where Na and Nb represent the face normals for the two polygons adjacent to the polygon edge and V represents the view vector, pointing from one of the edge vertices toward the camera. All vectors should be normalized. [Marshall 01]

Na and Nb can be calculated:

```
Na = normalize(cross((v1 - v0), (v2 - v0)));

Nb = normalize(cross((v3 - v0), (v1 - v0)));
```

V can be calculated:

```
V = normalize(CameraPosition - v0);
```

The dot product of two normalized vectors is the cosine of the angle between them. For vectors greater than 90 degrees apart, the result of a dot product is negative. Two back-facing or two front-facing polygons will result in the multiplication being positive. Therefore, by multiplying the dot product results together, only cases where one polygon or the other is back-facing, but not both, will render the contour test true.

Crease Edge

```
dot(Na, Nb) < -cos(θ)
```

Where θ is the user-defined dihedral angle. All vectors should be normalized.

The dihedral angle measures the angle between two planes, which are the two adjacent polygons in this case. However, the dot product is cosine of the angle between the face normals. The dihedral angle and the angle between the normals form an inverse relationship, hence the negating of the cosine operation on the right. The crease test will render true if the angle between the polygons is smaller than the dihedral angle.

To differentiate between ridge and valley crease edges, the following modifications [McGuire and Hughes 04] can be made to the test:

Ridge Crease Edge

```
(dot(Na, Nb) < -cos(θr)) && (dot((v3 - v2), Na) <= 0)
```

Valley Crease Edge

```
(dot(Na, Nb) < -cos(θv)) && (dot((v3 - v2), Na) > 0)
```

Where θr and θv are the ridge and valley dihedral angles, respectively. All vectors should be normalized.

The right side of the test differentiates ridges and valleys by comparing the vector pointing from v2 to v3 to the face normal Na. This is because the Na points away from v3 when the edge is a ridge, and it points towards v3 when the edge is a valley.

The differentiation of crease edges into ridge and valley varieties was included for completeness and will not be mentioned further.

Boundary Edge

Since boundary edges are based on the structure of the mesh and do not cease to be boundaries when the mesh animates, these edges should always be pre-calculated and flagged as drawable.

Marked Edge

Similar to the boundary edge, these edges are always pre-calculated. They should be flagged as drawable.

Intersection Edge

These do not correspond to polygon edges, so they are not handled by this technique, if at all. They will not be discussed further in this section.

6.2. Edge Detection

The most basic object-space method requires that the mesh be preprocessed into an edge mesh, which stores adjacency data for the mesh's polygon edges. It should contain the two edge vertices and the one or two other vertices that define the one or two triangles attached to the edge. Every polygon edge should get one and only one entry, unless modifications or optimizations require more.

How this is implemented is determined by the needs of the architecture. If possible, it is beneficial to store the edge mesh in terms of indices into the vertex array, rather than making duplicates of the vertices. Doing so will significantly reduce the amount of memory needed to store the edge mesh.

Detecting the edges simply involves looping through each entry in the edge mesh to determine if it fits the criteria for being drawable. Boundary, marked, and pre-calculated crease edges can be flagged as always drawable by setting the v3 entry equal to the v0 entry. One could also specifically differentiate the various pre-calculated drawable edges by setting v3 equal to v1, or v2 equal to v0 or v1. This edge mesh setup closely resembles McGuire's [McGuire and Hughes 04]. Others exist and a few of them are mentioned below.

Optimizations and Variations

Markosian et al. [Markosian et al. 97] used complete adjacency information, storing references to all connecting vertices at each end of the edge. By doing so, drawable edges could be traversed along their entire length. This traversal ability was used to remove the need to test every edge every frame. They randomly checked a portion of the edges for drawability.

When a drawable edge was found, adjacent edges were also checked for drawability until no more drawable edges could be found. Doing this allowed them to detect the longest, and therefore the most relevant, edges with a minimum number of random tests. Because edges tend to stay the same from frame to frame, a portion of drawable edges from the previous frame were also tested.

Markosian et al. found a fivefold increase in speed using these optimizations over testing all edges. Of course, edges can be missed entirely from frame to frame, so some accuracy is lost. They used and improved hidden edge removal algorithms which prevent occluded edges from being drawn. Such algorithms are beyond the scope of this thesis.

Gooch et al. [Gooch et al. 99] described a method where they stored edges' normal arc on a sphere surrounding the object. Groups of similar arcs, in gauss map format, were stored hierarchically so that groups of edges could quickly be deemed all back-facing or all front-facing. A plane was placed at the origin of the sphere and then aligned perpendicular with the view vector. Edges whose arc intersect the plane are contour edges. This technique allowed contour edge detection to be sped up by 1.3 times for their S. Crank mesh and 5.1 times for a sphere. Unfortunately, this technique only works well under orthographic projection.

In a similar idea, Sander et al. [Sander et al. 00] created a hierarchical search tree of polygons. At each node, they created anchored cones that represented the maximum range of the normals possessed by vertices in the node. This information can be used to quickly determine that no contour edges are possible for whole sets of nodes without testing individual edges.

John W. Buchanan and Mario C. Sousa [Buchanan and Sousa 00] defined a different kind of edge mesh that helped in the testing for contour, boundary, and marked edges. It combined the edge detection into the polygon rendering, significantly reducing the impact of edge detection on rendering time. However, it assumed the processing of individual polygons during the rendering process, which is no longer common thanks to vertex buffer arrays.

Jeff Lander [Lander 01] documented the optimization of ignoring edges that have co-planar adjacent polygons. Flat planes only generate drawable contour edges on their outside edges, not their internal edges, and they lack the angular difference between adjacent polygons to generate crease edges. During the preprocessing step, an additional test checks for co-planar adjacent polygons. If found, the edge is not added to the edge mesh.

Depending on how the mesh is created and how many flat areas it has, the savings in the number of needed edges can be significant. For example, the author found that 22.3% of the edges in the Utah Teapot can be safely ignored, and an amazing 50% of a cylinder's edges can be ignored. Low-polygon meshes, typically created using triangles instead of quads have a much lower number of ignorable edges. Only 1.6% of Blender's Suzanne Monkey model can be ignored.

Morgan McGuire and John F. Hughes [McGuire and Hughes 04] detailed a method to shift the entirety of the edge detection and rendering to the graphics card via programmable shaders. Copies of adjacency, vertex, and normal information were stored in vertex buffer arrays and accessed within the vertex shader. If the edge was found renderable, the degenerate duplicate vertices were turned into screen-aligned quads. Otherwise, they were shifted behind the camera, where they would be clipped during the rendering process. McGuire and Hughes also took a critical look at how the thick edge gaps could be filled effectively. Since McGuire and Hughes' paper had a major influence on this thesis, its algorithms and features will be more specifically addressed later.

Still other variations exist. Aaron Hertzmann and Dennis Zorin [Hertzmann and Zorin 00] described a method of using 4D dual surfaces to determine the contour edges with curve-plane intersections. Tom Hall [Hall 03] created a modification of Markosian et al.'s technique by focusing almost exclusively on tracking contour changes from frame to frame. By looking at adjacent edges to previously found edges and noticing the relative camera change, he was able to significantly reduce the number of edges tested. His method worked especially well with highly tessellated meshes.

6.3. Edge Rendering

In object-space edge detection, the rendering is a distinctly different process from the detection. After a polygon edge has been determined drawable, the user has the choice of what to do with that information. The possible edge effect is highly dependent on the technology available and speed requirements. In this section, several methods of edge rendering are discussed.

Line Edge

The most simple method of drawing the edges is to create a rasterized line between the two edge points: v0 and v1. If the hardware supports it, thick edges can be created by specifying these rasterized lines be greater than one pixel in width. This method is very fast and requires no additional processing of the edges. One advantage rasterized lines have over a single pixel-width quad is that no matter their orientation, they always render a single pixel on the edge. A single pixel-width quad can get rendered to less than a single pixel at some orientations, creating artifacts. However, these lines can only be customized in terms of their color. They cannot be textured.

Quad Edge

By expanding edges into screen-space quads, they can be made as thick as the user desires, textured, and/or anti-aliased. McGuire and Hughes [McGuire and

Hughes 04] worked almost exclusively with this method. They constructed two types of edges from quads: full and half quad edges.

Full quad edges were made by generating the screen-space perpendicular vector, scaling it, and using it to offset the points of the edge on both sides. This generated a thick quad around every drawable edge.

Half quad edges were used mainly for contours. Since each edge is on the surface of the mesh, contour edges would have half of their quad rendered below the surface. This can lead to artifacts. Half quad edges only render on the "outside" of the mesh, pointing in the same direction as v0's vertex normal n0, solving the artifact problem.

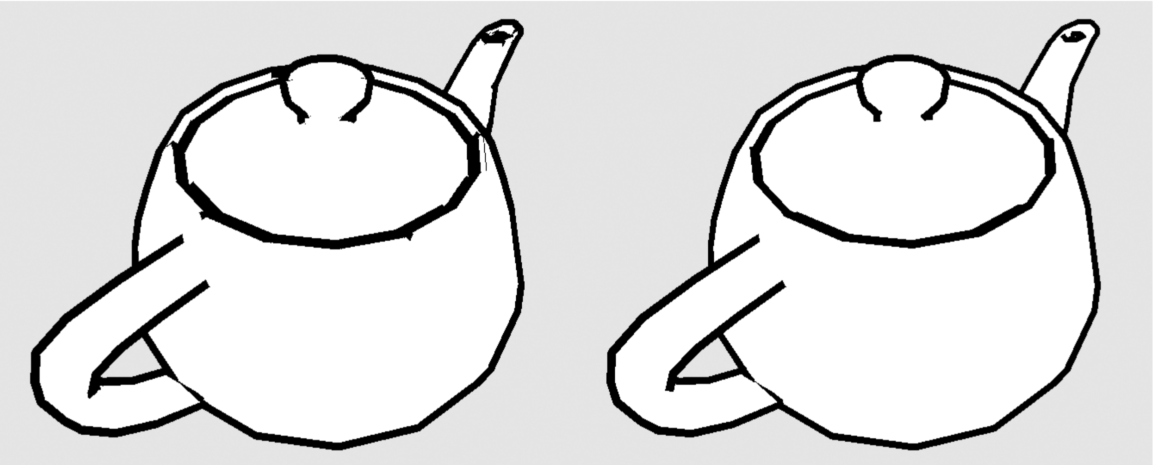


Figure 6-2: The full quad method (left) has artifacts on the spout and lid where the bottom half of an edge pushes through the surface. Using half quad edges for the contours (right), these artifacts are eliminated.

Like any quad, these edges can be textured, anti-aliased, or shaded as the user desires. Adjusting the scaling factor for the perpendicular vector allows thickness control.

Other Artistic Edges

J.D. Northrup and Lee Markosian [Northrup and Markosian 00] connected related edges together to form a few sets of long, connected edges. They then expanded the edges into connected screen-space quads, which were easily textured. Using this connection step, textured strokes could easily extend beyond the bounds of a single polygon edge's screen-space length. Such realistic stroke parameterization is difficult or impossible when edges are treated independently.

Jeff Lander [Lander 01] used multiple rasterized lines rendered at slight distance and rotational offsets to get a sketchy look. Marc ten Bosch [Bosch 06] created a method of partial edge detection that faded edges in as they approached visibility, and faded them out as they approached invisibility. This reduces the sometimes problematic popping effect that occurs when the contour edge from the previous frame shifts to another polygon edge.

6.4. Thick Edge Gaps

When rendering thick edges as thick lines or screen-space quads, non-parallel connecting edges will have visible gaps. These gaps get worse as the edges are made thicker. There have been a few attempts to address this problem.

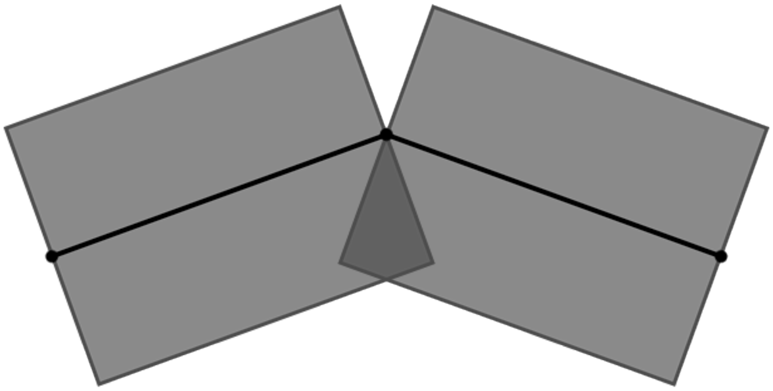


Figure 6-3: Two connected drawable edges. Since they are not parallel, they create a gap above their connection point.

Bruce Gooch et al. [Gooch et al. 99] suggested that a single, thick point be rendered at both ends of the of any drawable edge. While this will smoothly solve the gap problem, thick rasterized points are not available on all hardware. Additionally, these points can generate the same artifacts as full quad edges on contours, and they interrupt edge texturing.

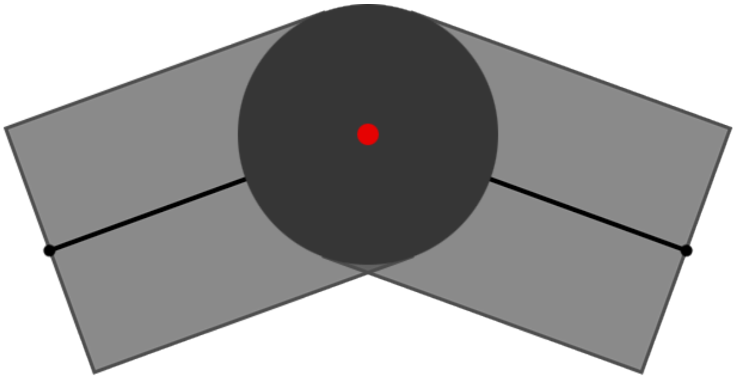


Figure 6-4: The red dot represents the screen-space vertex where the two edges connect. A single dark grey point is drawn there with a thickness equivalent to the edges' thickness.

Morgan McGuire and John F. Hughes [McGuire and Hughes 04] solved the gap problem with the limited resources of the GPU shader environment by rendering two extra cap-passes, where they generated triangle caps on both ends of the edge. The caps of connecting edges' triangle caps shared a vertex along the vertex normal projected into screen-space, which allowed the caps to line up in most cases. They also found a way to implement texture parameterization on these caps such that they wouldn't interfere with the texturing of the edges they connected to. There were a few problematic cases, however, when the vertex normals' screen-space projections did not reflect the curvature of the edge, resulting in caps on the wrong side. They suggested generating caps on both sides of the edge in cases where such failure was likely. To make smoother caps, they also suggested that caps could be generated out of a triangle fan, rather than a single triangle, but at the expense of many additional render passes.

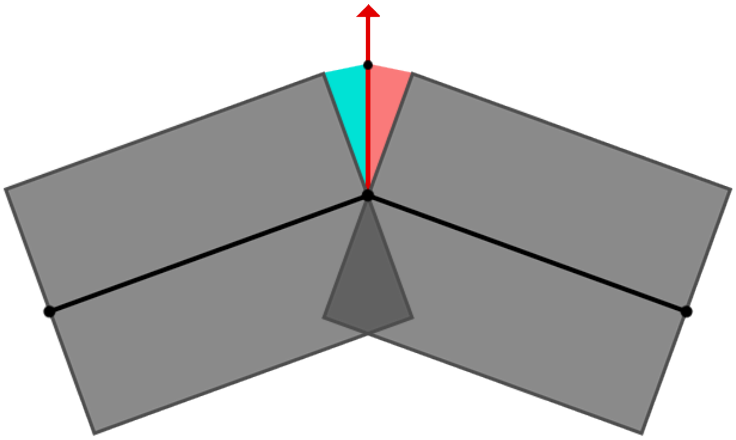


Figure 6-5: Using the red screen-space normal, both the left and right edges will generate half of the cap that fills the gap.

7. Miscellaneous Methods

This category is a catch-all for those methods that don't fit well in other categories. Thus, there are no real similarities among the methods listed.

7.1 Surface Angle Contours

Surface angle contours take advantage of the fact that as the surface of a smooth mesh approaches a contour, the normal becomes closer and closer to perpendicular to the view vector. The contour occurs on the surface at exactly the points on the surface where the view and normal vector are perpendicular.

Theoretically, the surface could be shaded black when the normal is exactly perpendicular to the view vector. However, meshes are made of flat polygons and their edges can be completely missed if using only the exact perpendicular pixels. Thus, the surface angle contour method always uses a range of values close to the perpendicular to indicate edges, regardless of implementation.

Gooch et al. [Gooch et al. 99] briefly mentioned the use of an environment map containing black color on the outer rim, forming a dark circle. Since the outer rim of the environmental map is where the calculated reflection direction of the near-perpendicular surface areas will access, the surface near the contours get rendered black.

Carl Marshall [Marshall 01] implemented the idea using a pixel shader. At each surface pixel, the dot product of the view and normal vector was used to index into a 1D texture that contained a dark color at the lower end. Additionally, one could simply threshold the dot product value to render black below a certain value, as displayed in Figure 7-1.

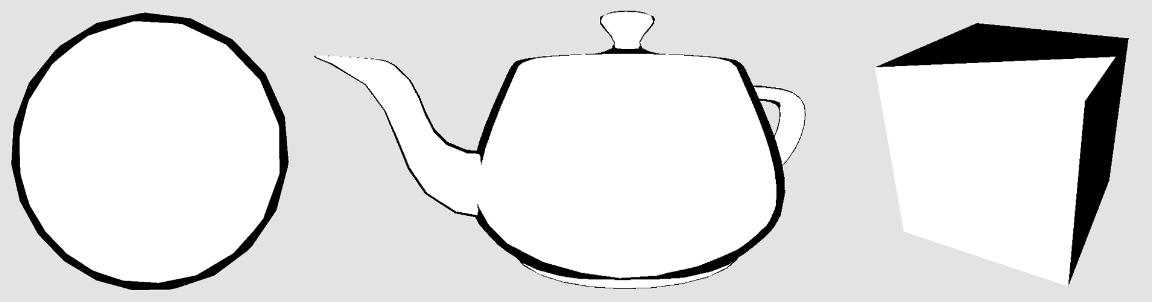


Figure 7-1: This variation uses a thresholding dot product check in a pixel shader. Some objects work better than others with this method, but the edge width is almost always inconsistent. The cube illustrates the main failure case for this method.

Though this method is extremely simple and inexpensive, it yields inconsistent results for some meshes. Flat surfaces in meshes cause a complete failure of this method.

7.2 Expanded Inverted Geometry Contours

Christopher Evans [Evans 03] described this method in a tutorial. The technique creates contour edges by rendering the mesh twice. The first pass renders the object normally, with whatever shading desired. The second pass, however, renders the same object scaled up slightly, with no shading, colored black, with its normals inverted, and back-face culling enabled. The combination of back-face culling and inverted normals only allows the back side of the scaled object to render, forming outlines for the object.

This method is extremely simple to implement and has a different visual effect than similar hardware methods. However, it also shares the multiple renders and relative lack of customizability of hardware methods.

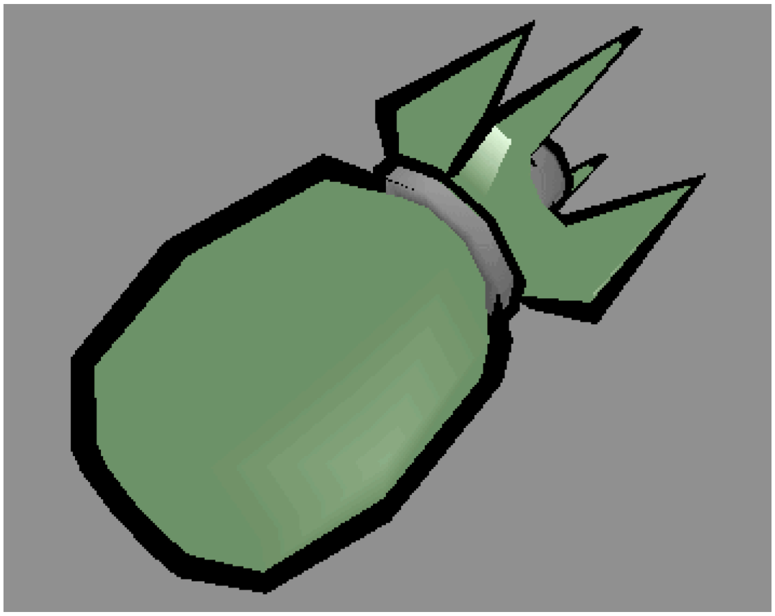


Figure7-2: The black edges are actually a scaled version of the bomb, rendered with back-face culling, inverted normals, and no shading. Some artifacts are generated, but it generally gives pleasing results. (Image courtesy of ChrisEvans3D.com)

8. Morgan McGuire and John F. Hughes’ Hardware-Determined Feature Edges

The author focused heavily on improving the object-space method described by Morgan McGuire and John F. Hughes [McGuire and Hughes 04]. To provide context and ease understanding, the author here details McGuire and Hughes’s work as it relates to the author’s contributions.

McGuire and Hughes’ goal was to transfer all detection and rendering of edges to the graphics card, gaining the speed of parallel calculation and removing the bottleneck of geometry data transfer from the CPU to the GPU. Parallel computation and the technology of the time forced them to test every polygon edge independently, unlike some of the object-space optimizations. As a result, a large amount of GPU memory is required.

The method used four render passes:

- Mesh pass - Render mesh normally at a slight backward offset
- Edge pass - Render drawable edges as expanded quads or lines
- Cap pass (first side) - Render the first side cap of each drawable edge
- Cap pass (second side) - Render the second side cap of each drawable edge

8.1. Data Structure

McGuire and Hughes created an edge mesh data structure that contained four edge vertex for each unique polygon edge. The edge vertex contained values required for edge detection and generation. Those that can be represented geometrically are shown in Figure 8-1.

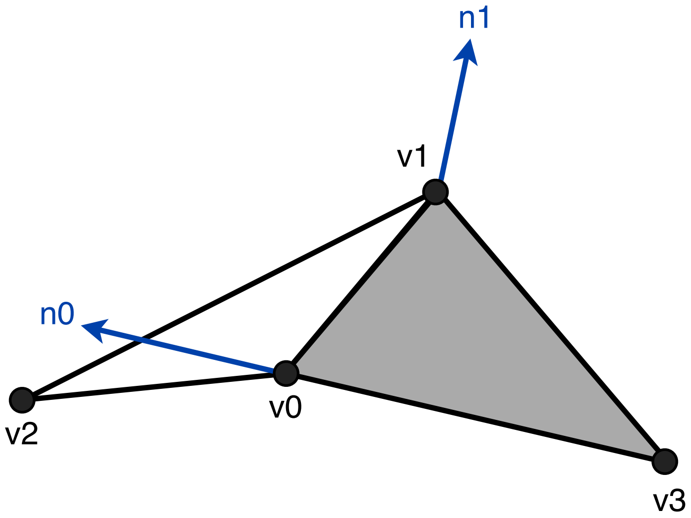


Figure 8-1: The important geometric values for each edge in McGuire and Hughes' method: v0, v1, v2, and v3 are vertices on two polygons that share an edge. n0 and n1 are the vertex normals for v0 and v1, respectively.

The values in each edge vertex are:

- v0 - first vertex in the edge
- v1 - second vertex in the edge
- v2 - final vertex that, with v0 and v1, makes the first polygon
- v3 - final vertex that, with v0 and v1, makes the second polygon
- n0 - v0's normal vector
- n1 - v1's normal vector
- r - random scalar used in texture parameterization
- i - scalar from 0 to 3 that differentiates duplicates in the edge mesh

v0, v1, v2, v3, n0, and n1 are all 3D vectors. r is used only for object-space texture parameterization. i is used to pick different output vertices in the vertex shader so that the edge will become non-degenerate. If the edge is a boundary edge, there is no v3 vertex. In that case, v3 should be set equal to v0.

For each polygon edge, all of the above values are obtained from the mesh, except for i. The newly formed edge vertex is duplicated three times. Each of the now four edge vertices with the same data are given a different, ordered i value from 0 to 3. All four of these edge vertices for each polygon edge is stored with all other edge vertices in an edge mesh. Finally, the edge mesh is copied into vertex buffers on the GPU for later use. This preprocessing step is quite expensive, but once complete, all edge detection and rendering can be accomplished by the GPU with no special data transfer whatsoever.

The user needs some way of referencing the edge mesh buffers on the GPU. Generating an index array buffer containing sequential numbers from 0 to (4E - 1), where E is the number of unique polygon edges, allows the data to be referenced and rendered with a single render call. Furthermore, other index array buffers can be created to reference only the first three, or first two, i values of each set of four edge vertices. Such index array buffers are useful for rendering caps and thin edges.

8.2. Edge Detection

McGuire and Hughes' method of edge detection uses a special vertex shader to access the edge mesh data from the vertex buffers prior to doing calculations. Once the data has been obtained, typical object-space edge detection is performed. As mentioned in Section 6.1, the edge tests are:

Contour	`(dot(Na, Nb) * dot(Nb, V)) <= 0`
Crease	`dot(Na, Nb) < -cos(θ)`
Boundary	`v3 == v0`
Marked	`v3 == v0`

Where Na and Nb are calculated:

```
Na = normalize(cross((v1 - v0), (v2 - v0)));
Nb = normalize(cross((v3 - v0), (v1 - v0)));
```

And V is calculated:

```
V = normalize(CameraPosition - v0);
```

In the event that an edge is determined drawable, the vertex shader must calculate an output vertex. McGuire and Hughes differentiated ridge and valley creases. However, the author does not differentiate them in his tests.

8.3. Edge Creation

The user has several choices for what type of edge to generate, including a rasterized line, full quad, or half quad. For each type of output, the i value is used to determine which of several output vertices should be generated.

Rasterized Line Edge

For rasterized lines, only two duplicates per edge vertices are needed in the edge mesh. If available, it's most efficient to use an index buffer that contains only the indices of the first two edge vertices per set in the edge mesh. When the i value is 0, the vertex shader should output MVP * v0. Otherwise, it should output MVP * v1. MVP is the ModelViewProjection matrix.

Required Screen-Space Values

For both quad edge types, McGuire and Hughes create them in screen-space to ensure a consistent thickness. The screen-space versions of some edge components are needed to accomplish the creation process. Those values are:

- s0 - v0 in screen-space
- s1 - v1 in screen-space
- m0 - n0 in screen-space
- p - normalized perpendicular vector to the screen-space edge vector (s1 - s0)

The calculation of each value is described below. Note that the uncommon vector-vector multiplication and division operations are used in these and following operations. They should simply be treated as component-wise multiplication and division. Also note that McGuire and Hughes did not fully convert vertices to screen-space, doing their calculations in projection-space instead. The author fully converts all edge generation vertices and vectors to screen-space in his calculations below.

```
vec4 s0 = MVP * vec4(v0.xyz, 1.0);

s0.xy = (s0.xy / s0.w) * vec2(Width, Height);

vec4 s1 = MVP * vec4(v1.xyz, 1.0);

s1.xy = (s1.xy / s1.w) * vec2(Width, Height);
```

s0 and s1 are calculated storing v0 and v1 multiplied by the combined model, view, and projection matrix (MVP). Then, their x and y components are sent to screen-space by doing the homogeneous division and multiplying by the width and height of the viewport (Width and Height), respectively.

```
vec4 temp = MVP * vec4(v0.xyz + n0.xyz, 1.0);

temp.xy = (temp.xy / temp.w) * vec2(Width, Height);

m0 = normalize(temp.xy - s0.xy);
```

A point on n0 is converted to screen-space, then that value is subtracted from the s0 point. This normalized vector is m0.

```
p = normalize(vec2(s0.y - s1.y, s1.x - s0.x));
```

The p vector can optionally be lengthened or shortened to vary the screen-space thickness of the final quad. Using it and the other screen-space values in conjunction with each vertex's i value, the full and half quads can be created as described below.

Full Quad Edge

By default, an edge should be created as a full quad. This always applies to crease, border, and marked edges. Contour edges can optionally be rendered as half quads. The creation of each output vertex is described by the table below.

ID	Output Vertex
i = 0	`vec4((s0.xy - p.xy) / vec2(Width, Height) * s0.w, s0.zw)`
i = 1	`vec4((s1.xy - p.xy) / vec2(Width, Height) * s1.w, s1.zw)`
i = 2	`vec4((s1.xy + p.xy) / vec2(Width, Height) * s1.w, s1.zw)`
i = 3	`vec4((s0.xy + p.xy) / vec2(Width, Height) * s0.w, s0.zw)`

This output is described graphically in Figure 8-2. Note that the projection to screen-space operation is inverted after offsetting the edge points by the p vector, so that the vertex shader exports valid data.

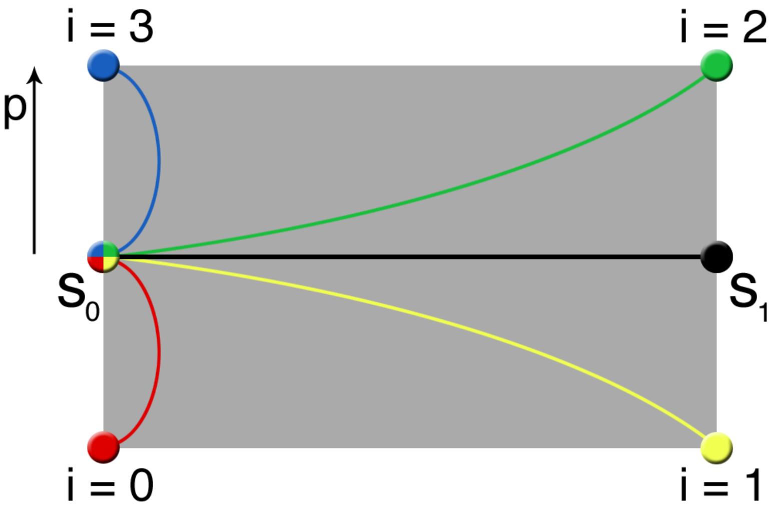


Figure 8-2: For each point in the degenerate set, the output is created based on the i value.

Half Quad Edge

The half quad method is intended for use with contour edges only. As mentioned in section 6.3, they prevent the bottom half of the quad from sticking through nearby geometry. To ensure that the edge is rendered to the outside of the mesh, each use of the p vector is modified by a sign operation. It reverses the direction of p if it is pointed away from m0 (that is, the angle between them is greater than 90 degrees). The creation of each output vertex is described by the table below.

ID	Output Vertex
i = 0	`s0`
i = 1	`s1`
i = 2	`vec4((s1.xy + p.xy * sign(dot(m0, p))) / vec2(Width, Height) * s1.w, s1.zw)`
i = 3	`vec4((s0.xy + p.xy * sign(dot(m0, p))) / vec2(Width, Height) * s0.w, s0.zw)`

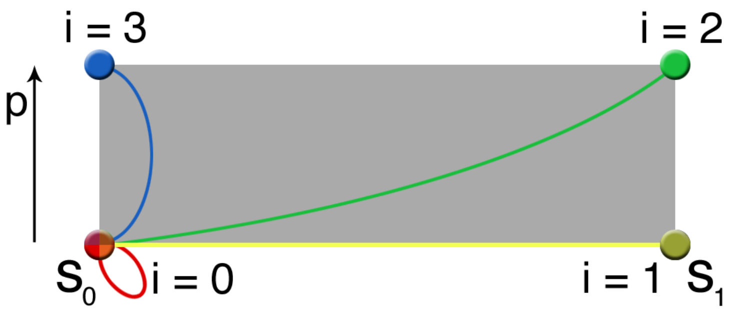


Figure 8-3: The p vector is always pointing outward in this method, so the edge quad will only render on the outside of the mesh.

Non-Drawable Edges

If the edge is not drawable, the vertex (0, 0, -1, 1) should be generated as output. Such a projection-space point will always render behind the near plane, and the pipeline will clip it before attempting to render it. This is how McGuire and Hughes accounted for the vertex shader's inability to delete undesired vertices from the pipeline.

8.3. Cap Creation

As mentioned in sections 6.4, when edges are rendered thicker than a few pixels visible gaps occur. To fill them, McGuire and Hughes created caps that render on both sides of each drawn edge. These two half caps are triangle polygons that meet up along the screen-space projection of the shared vertex normal. Two extra render passes are required to create the caps: one for the v0 side and one for the v1 side. The caps can be generated from the same edge mesh data as the edges, but they need a different index array buffer. The cap index array buffer needs index values that reference only the first three duplicates of each set of edge vertices.

Unlike the half quad edge method, the caps need to be generated relative to the normal of the vertex to which they are attached. Thus, the screen-space projection of the n1 vector, m1, is also needed to generate the v1 cap.

It is created:

```
vec4 temp = MVP * vec4(v1.xyz + n1.xyz, 1.0);

temp.xy = (temp.xy / temp.w) * vec2(Width, Height);

m1 = normalize(temp.xy - s1.xy);
```

With m1 and previous screen-space values, the v0 and v1 side half caps are generated as described by the tables below.

ID	Output Vertex (v0 side cap)
i = 0	`s0`
i = 1	`vec4((s0 + p * sign(dot(m0, p))) / vec2(Width, Height) * s0.w, s0.zw)`
i = 2	`s0 + m0`

ID	Output Vertex (v1 side cap)
i = 0	`s1`
i = 1	`vec4((s1 + p * sign(dot(m1, p))) / vec2(Width, Height) * s1.w, s1.zw)`
i = 2	`s1 + m1`

If the user decides to change the edge thickness by scaling the p vector, the m0 and m1 vectors must be scaled by the same amount. McGuire and Hughes were careful to generate the caps in an order that causes only front-facing polygons to occur. The author does not use back-face culling, so this detail was ignored.

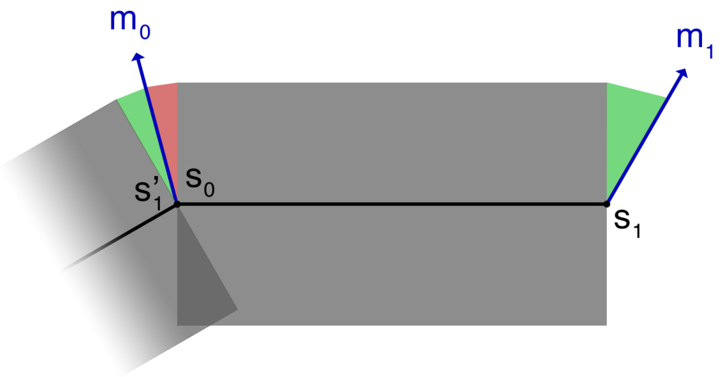


Figure 8-4: Depiction of how caps are formed. The red cap is created for the v0 point. The green caps are created for the v1 point of each drawable edge. Note that two half caps are created for each drawable edge, even if there is no connecting edge. Caps connect along the normal, making them seamless.

8.4. Problems

McGuire and Hughes' technique has a few major problems, two of which were caused by the limitations of the hardware available at the time of writing. The author's contributions solve or partially solve these problems.

Screen-Space Thickened Edges

While it is trivial to scale the thickness of the drawn edges and caps, the use of screen-space scaling makes all edges have the same thickness no matter their distance from the camera. This can cause artifacts and confusion about the distance of the object. Though screen-space thickness may be desired, having a depth-based scaling factor is beneficial for other graphical requirements.

Caps Sometimes Appear on the Wrong Side

When the screen-space projection of the vertex normals does not correspond well to the curvature of the edge, caps can be generated on the wrong side of the edge under certain perspectives. McGuire and Hughes suggested rendering caps on both sides of the edge when the error is likely to occur.

High Duplication of Work

The edge detection must occur ten times for all polygon edges: four for the edge, and two sets of three for the half caps. After an edge/cap is determined drawable, the screen-space values must also be generated ten times. Geometry shader usage was proposed as a solution to this problem, but the technology wasn't yet available.

High GPU Memory Usage

In order to render the object normally, all of the vertex and normal information is already on the GPU in the form of vertex buffer arrays. Not only does the edge mesh contain four duplicates, but all of the data for each edge vertex is a duplicate of information already on the GPU. This was done because of the limitations of vertex shaders at the time. Shader access to data textures was proposed as a solution for this problem, but the technology wasn't yet available.

9. Non-Duplicate Parallel Edge Detection and Capping

After researching and implementing McGuire and Hughes' method, the author focused on improving it. The most important resulting contribution is called non-duplicate parallel edge detection and capping. It provides improvements to McGuire and Hughes' method by completely reworking the implementation to take advantage of a new technology.

OpenCL (Open Computing Language) is a framework for doing general computation on a wide variety of hardware. It's primary focus is to unlock the massively parallel computational power of modern GPUs for non-graphical applications. It also offers buffer interoperability with OpenGL (Open Graphics Library). The author discovered that these features lend themselves well to object-space edge detection. It single-handedly replaces the two future technologies mentioned in McGuire and Hughes' paper as ways to improve their method: geometry shaders and data texture lookup. Simultaneously, it provides a way to create more accurate caps and eliminate the problem of inappropriate normal projection.

9.1. Method Overview

OpenCL allows the reading and writing of OpenGL vertex buffers. This means that the geometric and connectivity data need not be sent to the OpenCL context every frame. It can be generated during a preprocessing step and then transferred to the GPU's buffers, just as McGuire and Hughes did. Unfortunately, OpenCL does not support rendering commands, so detection and rendering must be separate steps.

Since all vertex buffers on the GPU are theoretically available to an OpenCL program (called a kernel), duplicate vertex information is not needed. The kernel can make use of the mesh's vertices already on the GPU. The only additional buffer needed by the GPU is the vertex connectivity information in the form of vertex indices. By not repeating 3D vertex information, the amount of extra data required to render edges is significantly lower than in McGuire and Hughes' method.

Once the data is retrieved, the edge detecting step is largely the same as in McGuire and Hughes' method, with two exceptions: edge detection and edge value generation occurs once per edge, and the output vertices are stored to an output buffer rather than being rendered directly.

The caps, on the other hand, are created completely differently from McGuire and Hughes' method. The cap creation kernel uses output data from the edge detecting step to skip the edge detection step entirely. Normals are not used in the creation process, so caps no longer appear on the wrong side of the edge under any circumstance.

Two data passes are used to calculate the edges and caps. Then three very simple rendering passes render the mesh, edges, and caps. The details of the implementation are below.

9.2. Data Structure

A large amount of preprocessing is required to arrange a mesh's polygon edges into an edge and cap mesh. The edge mesh is similar to the one generated for McGuire and Hughes' method, except that only the connectivity information is needed in the form of indices. Duplicate vertices and normals are not needed. The cap mesh, however, indexes into both the vertices and the edge mesh so that it can use the output of the edge detection step as input for the cap detection step.

First, all triangles in the mesh are processed to generate a list of unique edges with associated adjacent points. Since indices are being used instead of the actual data, each edge vertex contains:

- Edge vertex index 0 (indexes the vertex buffer)
- Edge vertex index 1 (indexes the vertex buffer)
- Adjacent vertex index 0 (indexes the vertex buffer)
- Adjacent vertex index 1 (indexes the vertex buffer)

All of these edge vertices together represent the edge buffer. These indices refer to the vertex buffer for the mesh, and could also be used to access the normals. However, unless half quads are being used, normals are unnecessary. As with McGuire and Hughes' method, if the adjacent vertex index 1 does not exist because the edge is a boundary edge, it is set equal to edge vertex index 0.

Another buffer, the edge out buffer, needs to be created with enough space to contain the set of four potential projection-space output vertices for each edge in the edge buffer. The buffer must have room for 4E four dimensional vertices, where E is the number of edges in the edge buffer. This buffer will hold the quad representations of the drawable edges for the dual purpose of rendering edges and defining which caps should be drawn.

Then the cap buffer should be created, using the edge and vertex buffers' indices. Each cap vertex contains:

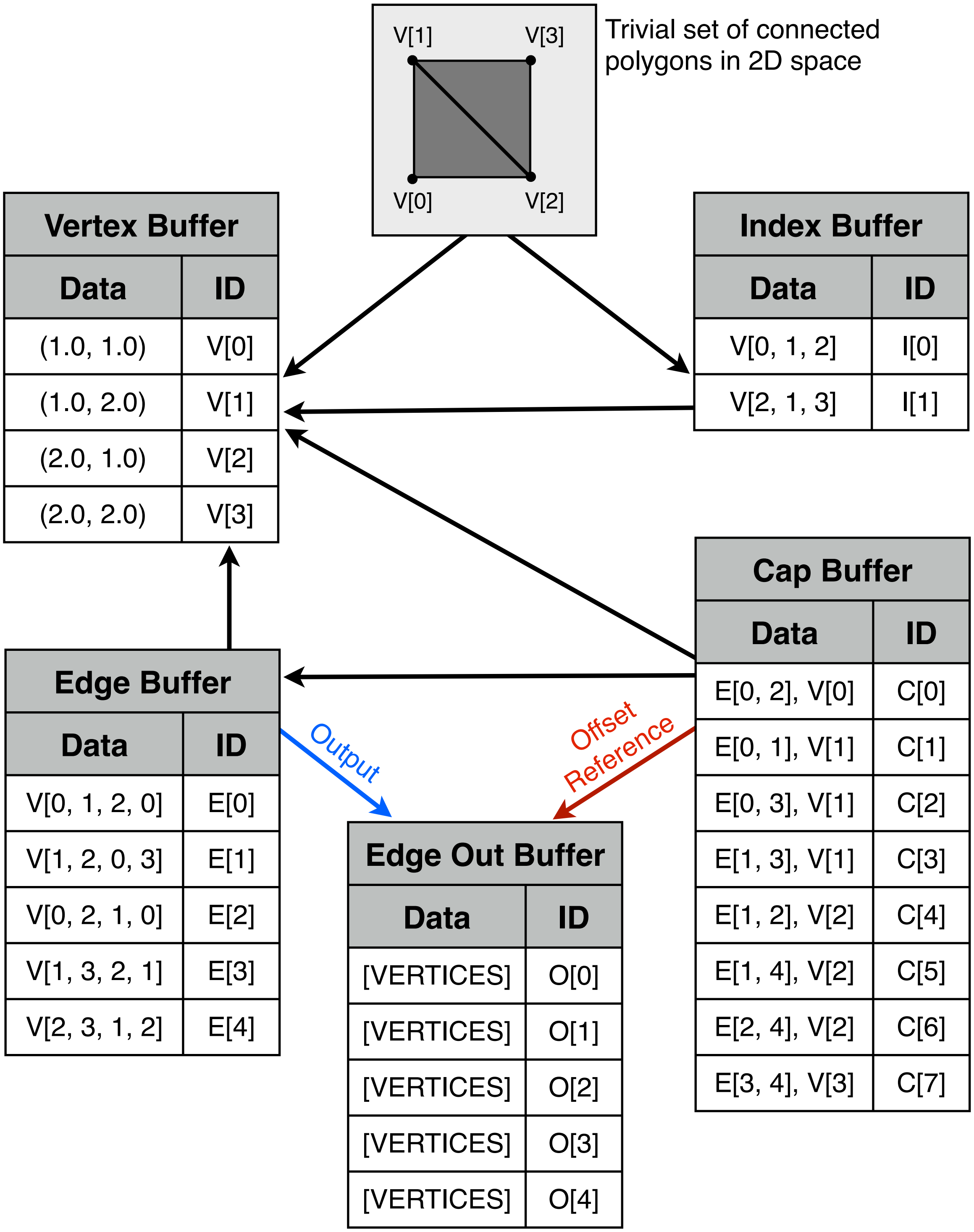
- Edge index 0 (indexes the edge buffer)
- Edge index 1 (indexes the edge buffer)
- Connecting vertex index (indexes the vertex buffer)

The two edge indices serve a dual purpose:

- Used with known offset to index into the edge out buffer to determine cap drawability
- Used with the connecting vertex index to determine how to retrieve data necessary to generate the cap vertices

As with the edge buffer, the cap buffer needs a corresponding cap out buffer to store its sets of projection-space vertices for later rendering. Since these caps will be made of quads, the buffer must be able to store 4C four dimensional vertices, where C is the number of caps in the cap buffer.

Once all these buffers have been created, and the two input buffers populated, they should be sent to the graphics card in preparation for edge detection and rendering. Refer to Figure 9-12 for a depiction of the buffers' relationships.



Vertex Buffer

Data	ID
(1.0, 1.0)	V[0]
(1.0, 2.0)	V[1]
(2.0, 1.0)	V[2]
(2.0, 2.0)	V[3]

Index Buffer

Data	ID
V[0, 1, 2]	I[0]
V[2, 1, 3]	I[1]

Edge Buffer

Data	ID
V[0, 1, 2, 0]	E[0]
V[1, 2, 0, 3]	E[1]
V[0, 2, 1, 0]	E[2]
V[1, 3, 2, 1]	E[3]
V[2, 3, 1, 2]	E[4]

Edge Out Buffer

Data	ID
[VERTICES]	O[0]
[VERTICES]	O[1]
[VERTICES]	O[2]
[VERTICES]	O[3]
[VERTICES]	O[4]

Cap Buffer

Data	ID
E[0, 2], V[0]	C[0]
E[0, 1], V[1]	C[1]
E[0, 3], V[1]	C[2]
E[1, 3], V[1]	C[3]
E[1, 2], V[2]	C[4]
E[1, 4], V[2]	C[5]
E[2, 4], V[2]	C[6]
E[3, 4], V[3]	C[7]

Figure 9-12: For a trivial set of polygon edges (top), the vertex and index buffers are used to draw the mesh normally. The edge vertices (edge out buffer) are calculated using the edge buffer which contains vertex connectivity information referring to indices of the vertex buffer. To determine if a given cap should be drawn, the edge connectivity information (the two edge buffer indices in the cap buffer) are used to offset index into the edge out buffer where drawability is easily determined. If the cap should be drawn, the cap out buffer (not displayed) is filled with cap vertices by using the cap buffer's two edge buffer references and a vertex buffer reference, which corresponds to the vertex where the two edges connect.

9.3. OpenCL Notes

OpenCL can only access OpenGL buffers if the context of OpenCL is initialized with OpenGL's context. Obtaining the OpenGL context is implementation-specific. Once obtained, however, all OpenGL buffers can be freely read from, but in order to write to them, OpenCL must acquire or lock those buffers prior to operating on them. In order for OpenGL to regain the ability to render from those buffers, they must be released from OpenCL.

Additionally, non-texture buffer access of OpenGL buffers by OpenCL use the non-cached global GPU memory. This memory access is very slow, and optimization is very important to achieving good results. Other OpenCL-specific information and examples are available in the specification and any of the numerous tutorial websites about it.

9.4. Edge Detection and Creation

When the user creates the OpenCL edge detection kernel, the user supplies the number of executions it will perform. That quantity is equal to the number of edges in the edge buffer. Then, when the kernel is activated and all edges are being checked for drawability at the same time, each instance of the kernel knows what its own ID is. There is one ID per kernel execution, so this value is used as the index for the edge buffer.

In addition to the kernel ID, the edge detection kernel requires references to the vertex, edge, and edge out buffers and copies of the ModelViewProjection matrix, the camera position, and the viewport width and height. The kernel ID is used to index into the edge buffer. The four vertex buffer indices obtained there are then used to obtain the vertices necessary for edge detection: the two vertices defining the edge, and the vertices defining the adjacent points on the two connected polygons. This information is exactly the information available to McGuire and Hughes' shaders at the time of edge drawability testing.

The actual edge detection steps are exactly the same as those in McGuire and Hughes' methods, with the minor exception that boundary and marked edges can be checked for drawability using index comparisons instead of vertex comparisons. Check section 8.2 for the details.

For edges that are non-drawable, a degenerate quad made of the vertex (0, 0, -1, 1) should be sent to the edge out buffer, starting with the index (4 * ID). This assumes that the user is submitting the vertices to buffers as float4 values, rather than individual float values. Otherwise, the buffer index values here and elsewhere will need to be calculated appropriately and carefully so that the index access lines up to the appropriate items in the buffer.

For edges that are drawable, the kernel should generate the screen-space values necessary to generate the quad edge and then send those four vertices to the edge out buffer, starting with the index (4 * ID). These values should be exported in projection-space, just as the vertices were output from McGuire and Hughes' vertex shaders. At this point, all four vertices can be generated in one step, preventing a lot of duplicate computations.

With the edge detection step complete, the edge out buffer contains sequential quad vertices that define the drawable edges. Non-drawable edges have degenerate quad entries that will be clipped during the rendering process.

9.5. Cap Detection and Creation

Cap detection is a completely new operation. In McGuire and Hughes' method, two half caps were created on each side of every drawable edge. Since the cap buffer provides the connectivity information of every possible combination of connecting edges, the caps can be generated more accurately and efficiently.

Like the edge detection step, the cap kernel will need some of the same resources, including: references to the vertex, edge, edge out, cap, and cap out buffers and copies of the ModelViewProjection matrix, the camera position, and the viewport width and height. The number of cap kernel operations should be equal to the number of entries in the cap buffer. The cap kernel will supply a kernel ID that can be used to index into the input and output buffers, as was done in the edge detection kernel.

The edge buffer contains indexes to the two connecting edges of a cap and an index to the vertex that connects them. By using the cap data indirectly, through the edge indices, the original edge vertices can be obtained. Then a typical edge test could be performed to determine the drawability of both of the edges. However, it's more efficient to use the recently populated edge out buffer to determine the drawability state of both of the connecting edges.

All non-drawable edges generate four entries of the vertex (0, 0, -1, 1) in the edge out buffer. Using the indices of the connecting edges offset by four, the kernel can obtain the first output vertex of each connecting edge. If both of these output vertices are not equal to the degenerate vertex, they are both drawable and the cap should be created at the connecting vertex. If only one or neither of them are drawable, then no cap should be created. Instead, a degenerate quad made of the vertex (0, 0, -1, 1) should be stored in the cap out buffer.

If the cap should be drawn, three vertices are needed:

- vL - the vertex on the “left” edge that is not the connecting vertex
- vM - the connecting vertex
- vR - the vertex on the “right” edge that is not the connecting vertex

The index of the vM vertex is part of the cap buffer item. vL and vR can easily be obtained by first checking the vertex indices of both edges. Then, after determining which indices correspond to the vL and vR using the known connecting index, the vertices can be retrieved from the vertex buffer.

Using the same operations as previously mentioned in section 8.3, these three vertices need to be transformed to screen-space vertices sL, sM, and sR. Then, using vectors along the edges, the “middle vector” needs to be determined. This vector lies on the same plane as, and points away from, the two edges. The middle vector will be used to generate the connecting point that was previously obtained from vertex normals.

It is calculated:

```
middleVector.xy = -normalize(normalize(sL.xy - sM.xy) + normalize(sR.xy
   - sM.xy));
```

The perpendicular vector for both edges are needed for the rest of the calculation. The “left” edge’s perpendicular vector will be referred to as p0, and the other, p1. They are obtained:

```
p0 = normalize((float2)(sL.y - sM.y, sM.x - sL.x));

p1 = normalize((float2)(sR.y - sM.y, sM.x - sR.x));
```

They need to be modified to point in the same direction as the middle vector (less than 90 degrees of angular separation). This modification is accomplished via the operations below:

```
p0 = p0 * sign(dot(p0, middleVector));

p1 = p1 * sign(dot(p1, middleVector));
```

After all these values have been generated, the output vertices for the drawable caps can be sent to the cap out buffer. The vertices of the cap are listed in the table below.

Vertex	Calculation
0	`(float4)((sM.xy + middleVector) / sM.w * (float2)(ScreenWidth, ScreenHeight), sM.zw)`
1	`(float4)((sM.xy + p0) / sM.w * (float2)(ScreenWidth, ScreenHeight), sM.zw)`
2	`(float4)(sM.xy / sM.w * (float2)(ScreenWidth, ScreenHeight), sM.zw)`
3	`(float4)((sM.xy + p1) / sM.w * (float2)(ScreenWidth, ScreenHeight), sM.zw)`

Since the cap’s creation is based on the connected edges rather than the normal, the normal’s orientation is irrelevant to the position of the cap. The cap also always perfectly fills the gap created between the two edges, no matter how the two edges are oriented.

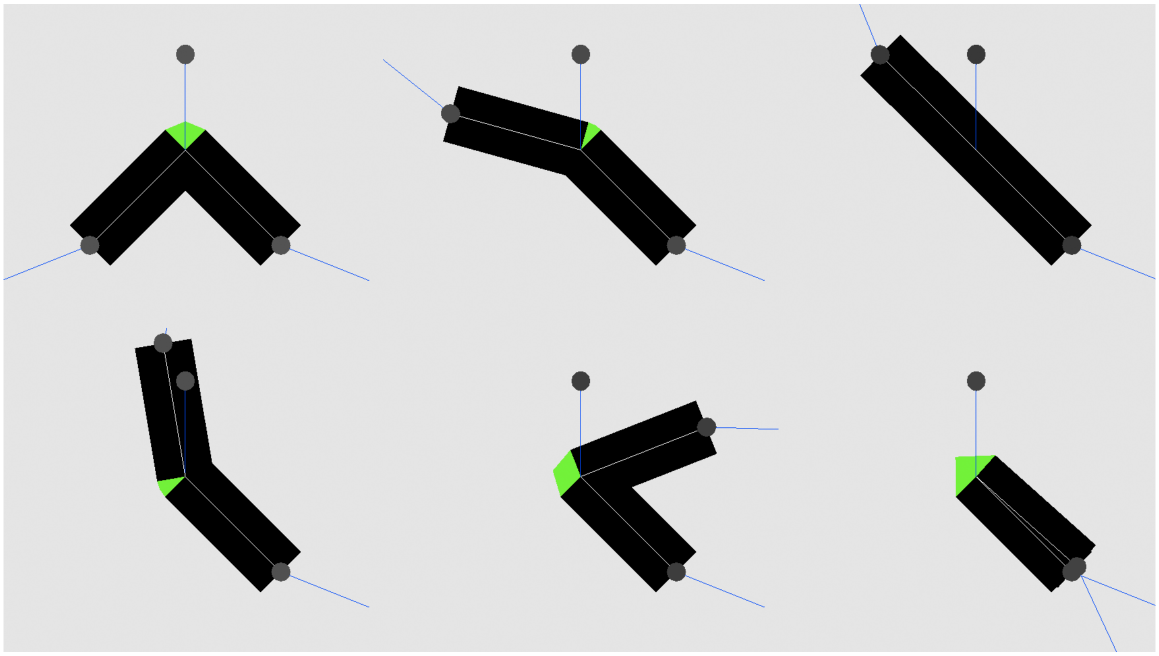


Figure 9-13: The green caps are generated without the use of the normal, and always perfectly fill the gap between the two edges because they are created utilizing the vertices of both edges.

9.6. Rendering

All edge and cap vertices were output from their respective kernels in projection-space, the output format of vertex shaders. Thus, the vertex shader for the edges and caps must only pass the vertex through directly, with no additional processing. The vertex shader is displayed below.

```
void main() { gl_Position = gl_Vertex; }
```

The complete render process consists of rendering the mesh normally, with a slight backwards offset, then rendering the edges and caps as ordered quad arrays. Because OpenCL can interoperate with OpenGL's buffers, no edge, cap, or vertex data is sent to or retrieved from the GPU during each frame.

10. Comparative Analysis

The author’s method of edge detection and rendering does vastly fewer operations because it can export multiple pieces of data at once. It also prevents caps from appearing on the wrong side because normals are no longer used. Unfortunately, due to the current state of OpenCL’s memory access technology and a lack of optimization, the actual speed of the edge detection is slower for the author’s method when compared to McGuire and Hughes’ method. While these speeds will improve with further optimization, in order to compare the two methods on an even playing field, the author implemented both techniques on the CPU.

The CPU implementation differs from the GPU implementation in only two ways: memory access speed is equivalent, and operations are not done in parallel. All further metrics are based on the CPU implementations. McGuire and Hughes’ method is referred to in charts as “GLSL” and the author’s method is referred to as “CL.”

10.1. Operation Quantities

In this section, charts and tables are presented that compare the number of various types of operations in both methods, for both edges and caps, under the best and worse-case conditions.

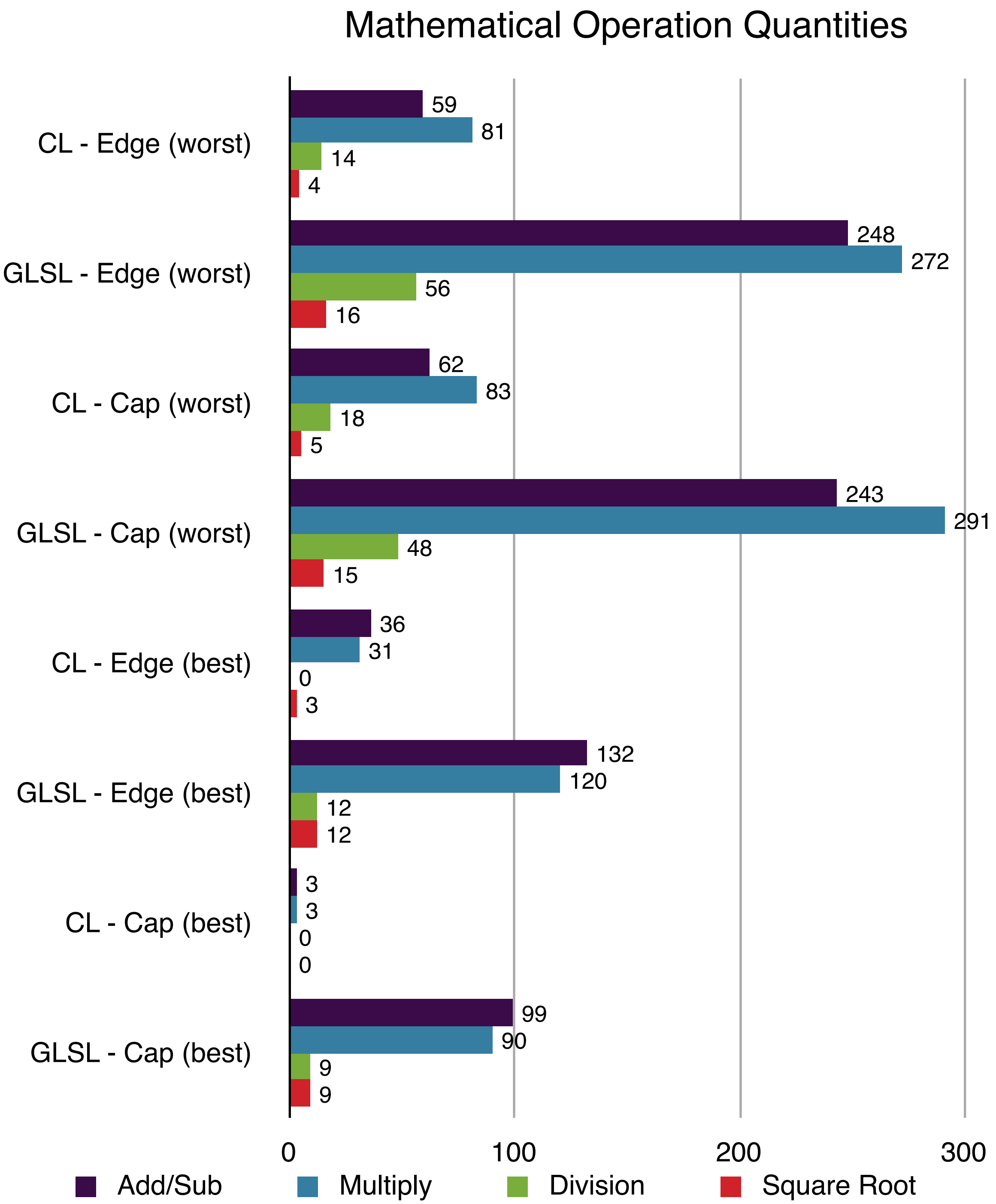


Figure 9-13: This chart shows the relative quantity of different types of mathematical operations used by both methods under different conditions. The author's method requires vastly fewer operations to its lack of duplication of work and test optimizations. Smaller is better.

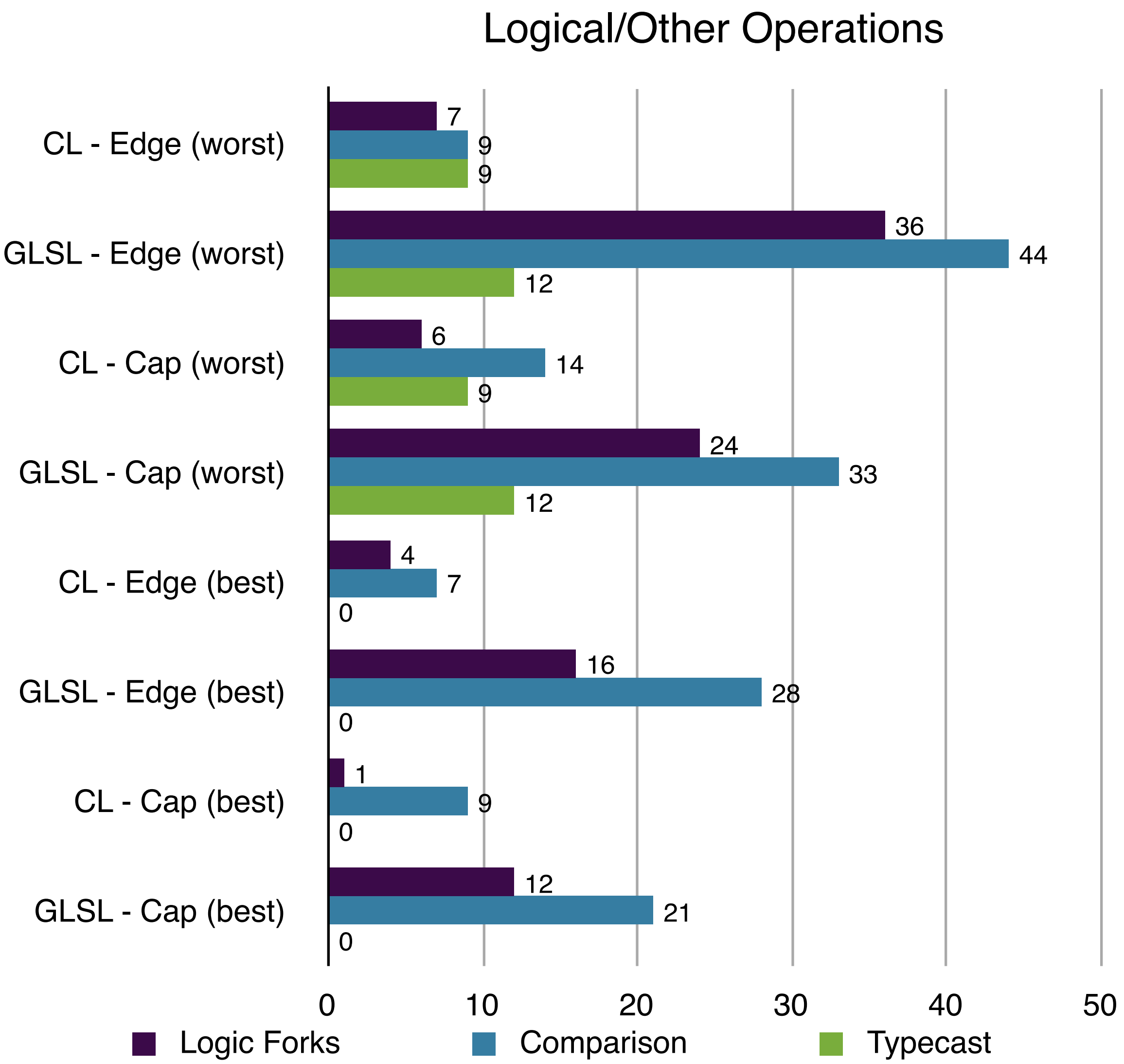


Figure 9-14: This is a comparison of the number of logic forks (if statements), comparison operations, and typecasts (vector types) used by each method under different conditions. The author's method does significantly fewer of these operations. Smaller is better.

The following table shows the ratio comparisons of the two methods. While the quantities of all operations were reduced in the author's method, many of them are from the simple elimination of duplication. Thus, those ratios that represent an operation reduction of greater than four times are displayed with a green cell. Smaller is better.

	Edge Ratio CL:GLSL Worst Case	Cap Ratio CL:GLSL Worst Case	Edge Ratio CL:GLSL Best Case	Cap Ratio CL:GLSL Best Case
Add/Sub	0.238	0.255	0.272	0.03
Multiply	0.298	0.285	0.258	0.033
Division	0.25	0.375	0	0
Square Root	0.25	0.333	0.25	0
Logic Forks	0.194	0.25	0.25	0.083
Comparisons	0.205	0.424	0.25	0.429
Typecasts	0.75	0.75	N/A	N/A

10.2. Memory

In this section, the memory access, memory usage, and local memory usage is compared for the two techniques for edges and caps under different conditions. 32-bit float and integer types were assumed.

Memory Access

As previously mentioned, memory access speeds are significantly different for shaders and OpenCL. Shaders get their memory access in a cached form and are approximately ten times faster than access to global values in OpenCL. This non-equivalence is why the global label is in quotes.

	“Global” Read	“Global” Write
CL - Edge (Worst)	16	16
GLSL - Edge (Worst)	48	N/A
CL - Cap (Worst)	31	16
GLSL - Cap (Worst)	48	N/A
CL - Edge (Best)	16	16
GLSL - Edge (Best)	48	N/A
CL - Cap (Best)	10	16
GLSL - Cap (Best)	36	N/A

In McGuire and Hughes' method, output data is sent directly into the pipeline for rendering. The author's method, on the other hand, must store those values to a buffer for later rendering.

Memory Usage

Calculating the total memory footprint of both edges and caps is dependent on the number of possible edges and caps, the view direction, and the type of mesh in question. So, several tables are needed to illustrate the conclusions found.

The first table describes the number of bits used by edges and caps in both methods. The caps of McGuire and Hughes' method uses only a small amount of memory because it makes use of the edge data. There is also some memory used by the GLSL edges and caps not mentioned here, because it is transparently created and destroyed by the graphics pipeline.

	Memory Usage Per Item
CL Edge	640 bits
GLSL Edge	2688 bits
CL Cap	608 bits
GLSL Cap	96 bits

Then, for a sample of objects, the next table compares the number of edges and caps used by both methods.

	Edges (CL)	Edges (GLSL)	Caps (CL)	Caps (GLSL)	Cap:Edge Ratio (CL)	Cap:Edge Ratio (GLSL)
Simple Cube	12	12	24	24	2	2
Normal Cube	24	24	24	48	1	2
Cylinder	96	96	192	192	2	2
Cone	64	64	592	128	9.25	2
Quad Sphere	2016	2016	6944	4032	3.44	2
Ico Sphere	1920	1920	9570	3840	4.98	2
Teapot	1180	1180	4420	2360	3.75	2
Monkey	1449	1449	7188	2898	4.96	2
Bunny	20812	20812	107290	41624	5.16	2

The number of caps to edges in McGuire and Hughes' method is always two because there are two half caps drawn for each edge, one on each side. For most meshes, the author's method will have far more caps because it handles every possible case. However, the significant calculation reduction on each cap in the author's method mitigates the impact of the additional caps.

Combining the information of the previous two tables, the number of bits needed by each type of object's edges and caps can be determined. They are displayed in the next table. Values are in bits.

	Edges (CL)	Edges (GLSL)	Caps (CL)	Caps (GLSL)
Simple Cube	7680	32256	14592	2304
Normal Cube	15360	64512	14592	4608
Cylinder	61440	258048	116736	18432
Cone	40960	172032	359936	12288
Quad Sphere	1290240	5419008	4221952	387072
Ico Sphere	1228800	5160960	5818560	368640
Teapot	755200	3171840	2687360	226560
Monkey	927360	3894912	4370304	278208
Bunny	13319680	55942656	65232320	3995904

Totaling all memory used by the edges and caps for each type of object under each method, the memory ratios illustrate that under the current system, most meshes use about the same memory in both methods. There are a few that have better memory usage (green), and one, the cone, with terrible memory usage (red). The cone's high memory usage comes from the high numbers of caps that occur at the tip where most of the edges come together. Values are in bits.

	Total CL Memory	Total GLSL Memory	Memory Ratio (CL:GLSL)
Simple Cube	22272	34560	0.644
Normal Cube	29952	69120	0.433
Cylinder	178176	276480	0.644
Cone	400896	184320	2.175
Quad Sphere	5512192	5806080	0.949
Ico Sphere	7047360	5529600	1.274
Teapot	3442560	3398400	1.012
Monkey	5297664	4173120	1.269
Bunny	78552000	59938560	1.31

Local Memory Usage

Within the kernel and shader, a certain amount of local memory is needed to perform the tests and other operations. The following table compares each method for edges and caps under different conditions. Values are in bits.

	Local Memory Usage
CL - Edge (Worst)	1216
GLSL - Edge (Worst)	1376
CL - Cap (Worst)	1312
GLSL - Cap (Worst)	1600
CL - Edge (Best)	704
GLSL - Edge (Best)	800
CL - Cap (Best)	32
GLSL - Cap (Best)	800

The amount of local memory needed for the author’s method is less in every case than McGuire and Hughes’ method.

10.3. Rendered Output

Regardless of the detection and creation speed, the number of edges and caps drawn will also influence the framerate. At a given view direction, the same number of edges will be generated by both methods. However, the number of drawn caps is significantly different. In McGuire and Hughes' method, the number of caps drawn is always two times the number of edges. The author's method is more varied, though consistently generates fewer caps than McGuire and Hughes' method (green). These numbers were calculated with the camera pointing at the origin and positioned at (3, 3, 3) in world-space. The meshes were at the origin, with a scale size of approximately one.

	Edges (CL & GLSL)	Caps (CL)	Caps (GLSL)	Cap:Edge Ratio (CL)	Cap:Edge Ratio (GLSL)
Simple Cube	12	24	24	2	2
Normal Cube	24	24	48	1	2
Cylinder	66	72	132	1.09	2
Cone	34	37	68	1.09	2
Quad Sphere	72	72	144	1	2
Ico Sphere	55	55	110	1	2
Teapot	205	228	410	1.11	2
Monkey	345	488	690	1.41	2
Bunny	1175	1397	2350	1.19	2

10.4. Speed

All meshes were tested with both methods for total rendering speed using the framerate as the metric. The table below shows the final results. The author's method is faster for complex meshes. For meshes that are very simple or have single vertices with many connecting edges, the OpenCL method is slower. Again, speed is based on the CPU implementation. When implemented in

OpenCL itself, the author’s method is still slower due to lack of optimization and memory access speeds.

	Framerate (CL)	Framerate (GLSL)	CL:GLSL Ratio
Simple Cube	1155	1180	0.979
Normal Cube	1135	1156	0.982
Cylinder	1139	1182	0.964
Cone	1276	1338	0.954
Quad Sphere	679	144	4.715
Ico Sphere	631	155	4.071
Teapot	906	226	4.009
Monkey	602	176	3.42
Bunny	54	14	3.857

11. Other Contributions

This section details the author's other modifications and improvements to McGuire and Hughes' paper. These contributions can be applied to McGuire and Hughes' method without using any new technology unlike non-duplicate parallel edge detection and capping.

9.1. Depth-Based Edge Thickness

In McGuire and Hughes' method, all edges and caps were drawn scaled along the screen-space perpendicular vector or one of the screen-space vertex normal vectors. Screen-space scaling gives all drawn edges the same thickness, regardless of distance from the viewer. Additionally, edges extending into the distance will appear to get larger the further away they reach, rather than smaller as one might expect from a 3D scene.

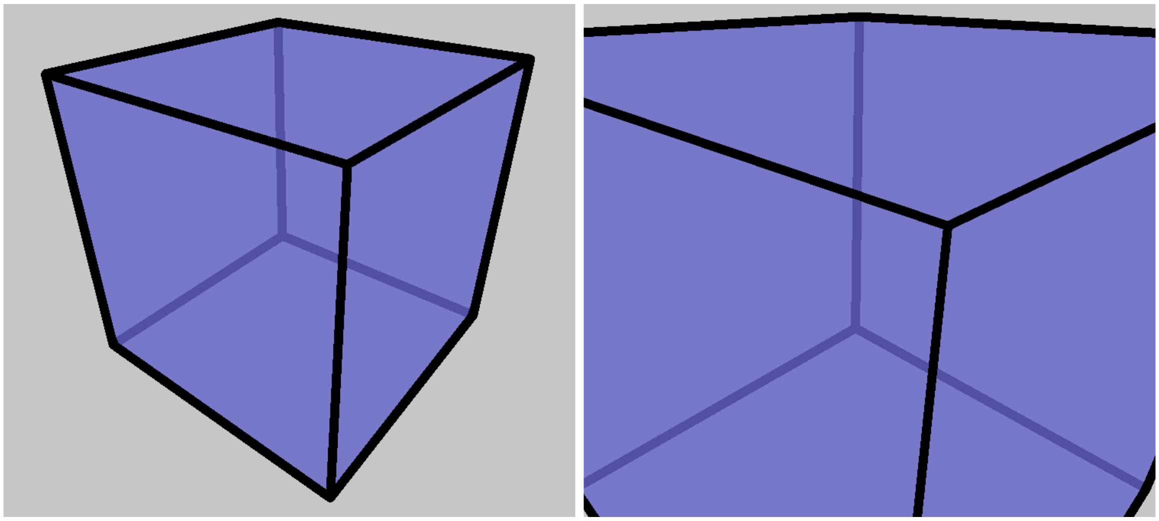


Figure 9-1: The cube on the left is a few units away from the camera. The right cube's front-most edges are directly in front of the camera, but have the exact same thickness as the cube's back edges.

Even if these effects are desirable, there is one major disadvantage to this form of scaling. As objects get further from the viewer, the object's geometry will eventually become overwhelmed by the edges.

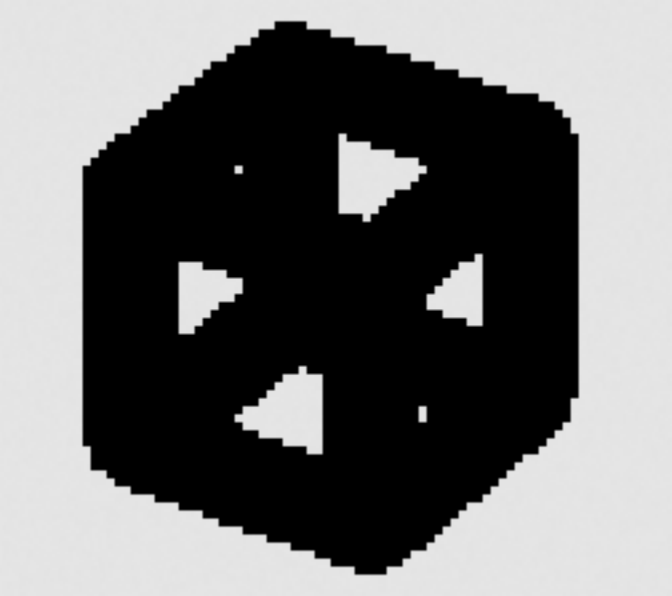


Figure 9-2: This cube is far away from the camera, yet the edges remain the same pixel width, overpowering the mesh.

To allow for more realistically scaled edges and prevent the overpowering problem, the author proposes a depth-based scaling factor be applied to the screen-space perpendicular and normal vectors of each edge vertex.

Naïve Method

The view-space z-coordinate represents the depth value needed to scale the screen-space vectors realistically. With each vertex in local-space, the naïve method to find their view-space representations uses a matrix multiply. Then the z-coordinates can be inverted and negated to form the depth-based scaling factor:

```
depthScalingFactor = 1.0 / -(ModelViewMatrix * localSpaceVertex).z;
```

This depth-based scaling factor must be multiplied into each use of the screen-space vectors during the vertex output process, multiplied into any other arbitrary scaling factors used.

Optimized Method

There is a way to avoid the matrix multiplication entirely. In the process of creating these edges, the projection-space version of v0 and v1’s z-coordinates are created. The depth value at each vertex in the edge can be calculated by multiplying the projection-space z-coordinates by the portions of the inverse projection matrix that apply to z-coordinates.

The inverse projection matrix is:

$$\begin{bmatrix} \frac{r}{n} & 0 & 0 & 0 \\ 0 & \frac{t}{n} & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{n-f}{2nf} & \frac{n+f}{2nf} \end{bmatrix}$$

In relation to the z-coordinate, the matrix multiplication operation of the inverse projection matrix and a projection-space point becomes:

```
depth = 0 * (projected.x + projected.y + projected.z) + (-1 *
   projected.w);
```

```
depth = -projected.w;
```

Thus the depth scaling factor calculation becomes:

```
depthScalingFactor = 1.0 / projected.w;
```

Where projected represents either v0 or v1 in projection-space.

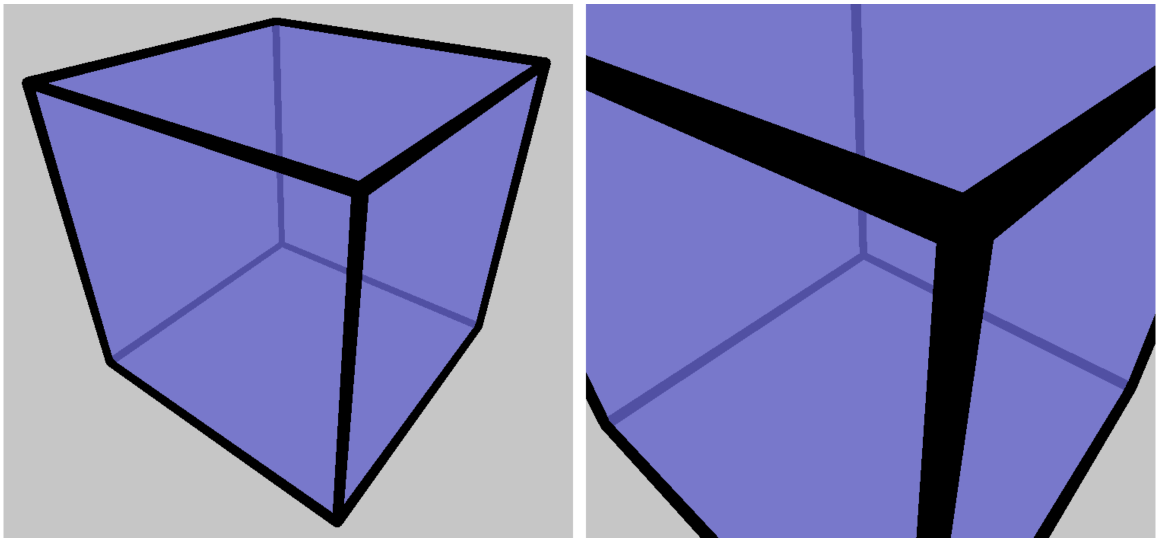


Figure 9-3: The same cube as above with the depth-based scaled thick edges. Notice how each edge's thickness will scale along its length.

Orthographic Projection Considerations

Such a compact optimization is not available when using orthographic projection. After applying the same rules to the inverted orthographic projection matrix, the calculation becomes:

```
depthScalingFactor = -2.0 / (projected.z * (n - f) - projected.w * (n +
   f));
```

The amount of calculation is not significantly less than accessing the value through world-space vector multiplication with the ModelView matrix, and requires two additional pieces of data: the near and far plane distances (n and f).

Minimum Thickness

One final improvement to the thickness modification is to cap the bottom of the scale factor through the use of a max function. This will assure a minimum edge thickness no matter the distance of the object.

```
depthScalingFactor = max(1.0, 1.0 / projected.w);
```

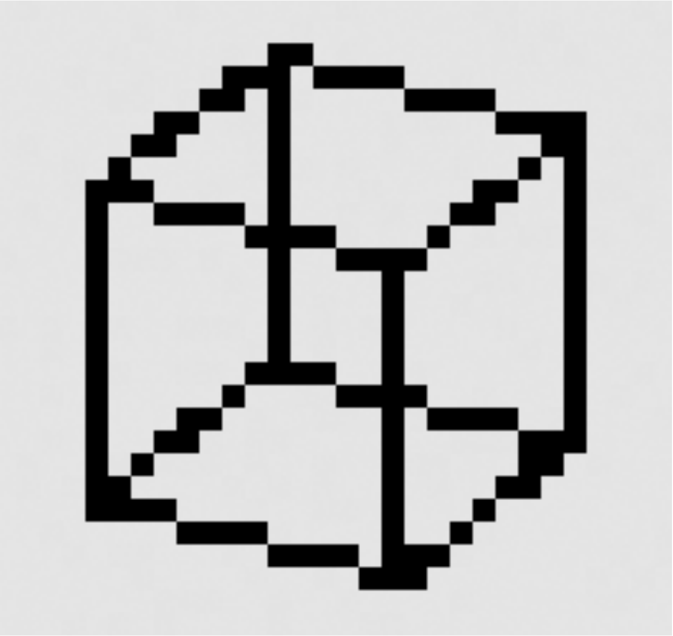


Figure 9-4: A distant cube with edges that remain visible yet don't overpower the mesh.

Edges rendered with a minimum thickness of less than one can cause flickering and missing edges if the distance is great enough.

9.2. Reduce Effects of Incorrectly Projected Normals

The screen-space projection of each vertex's normal is used to choose the side on which to render the caps. For some meshes viewed from certain view angles, the projected normals will point in the wrong direction, causing caps to appear on the wrong side of the edge. McGuire and Hughes suggested that the caps be rendered on both sides of the edge in cases where this might occur.

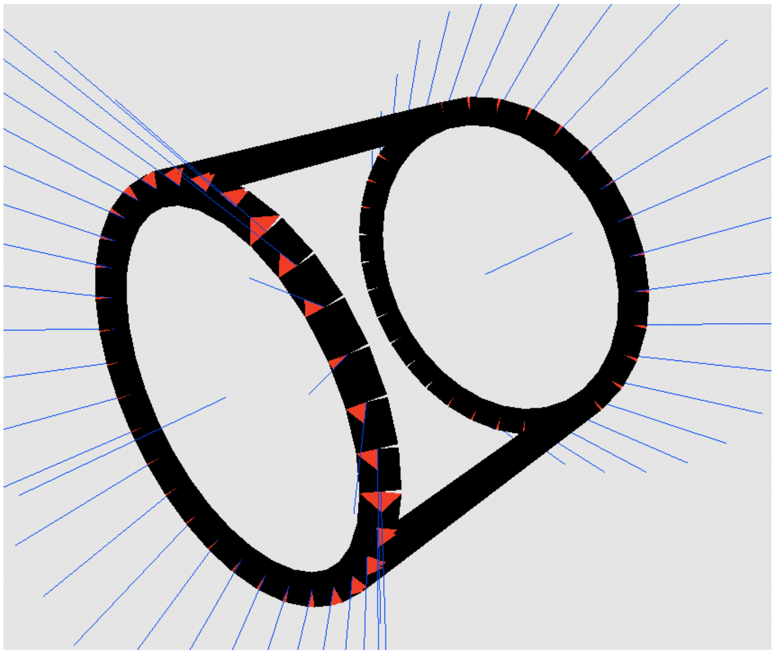


Figure 9-5: The shared vertex normals (blue) of the cylinder crease edges, when viewed from this angle, demonstrate the failure case of normal-based capping (red).

The author found that for every potential gap created at the intersection of two drawable edges, there is exactly one "normal" vector that, when projected, would yield a correctly placed cap. Each vertex is limited to a single normal vector because of the restriction to only include unique edges. However, if edges are defined not only by their location, but also by their normals, the problem can be solved. Crease edges tend to have more than one appropriate normal, as indicated in McGuire and Hughes' paper. By using a common crease splitting operation in a 3D application, and allowing edges to be distinguished by their normals in addition to their position, some edges are duplicated, but with appropriate normals. This is a simple application of the proposed solution in McGuire and Hughes' paper

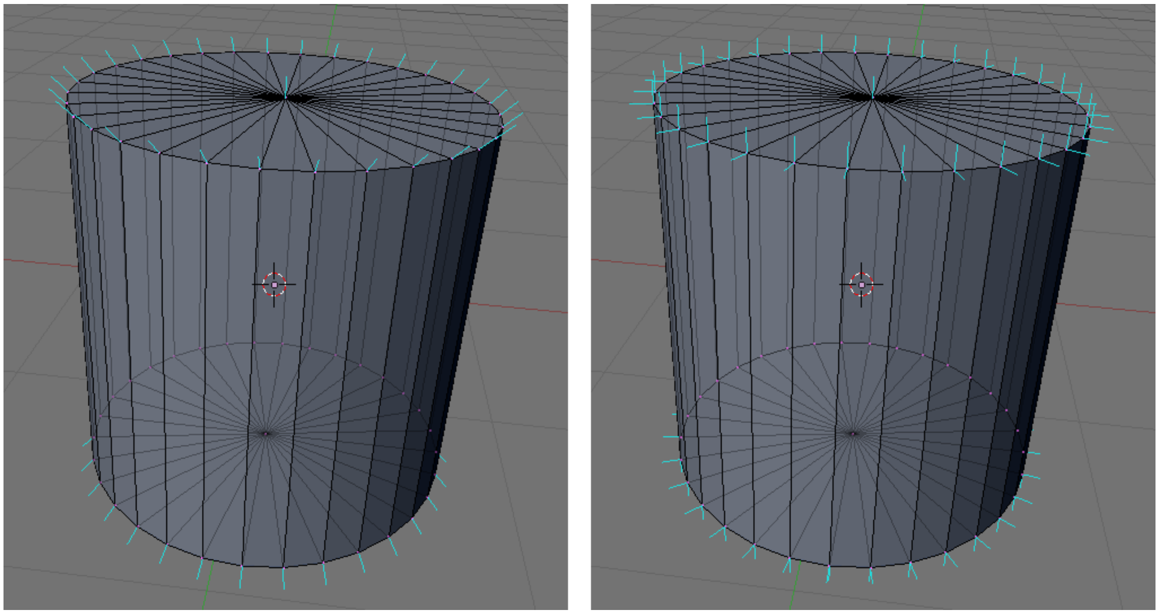


Figure 9-6: The shared vertex normals of the left cylinder generate caps that end up on the wrong sides from some view angles. The right cylinder, which has appropriate normals for each major section of the mesh at the cost of duplicate edges, will generate correct caps no matter the view angle.

If the right cylinder from figure 9-6 is rendered, the output looks like Figure 9-7. Since the edges are created without the normals, any duplicate edges could be marked to output a degenerate edge quad. The cap creation would remain the same. Doing this would reduce the impact of having the additional edges in the edge mesh. Using double caps on all crease edges would yield similar results, but at the cost of additional render passes for all edges.

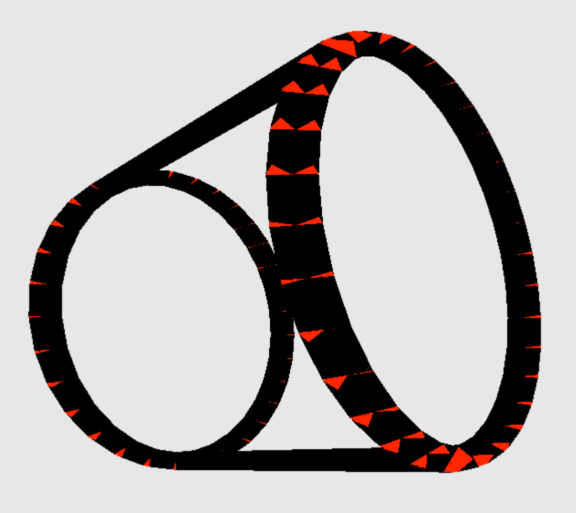


Figure 9-7: A second set of normals for problematic edges generates the correct output.

For some meshes, the problematic normals originate from a curved portion of the mesh abutting a flat portion. When these objects are created, the vertices that connect the curved and flat portions get blended normals, which are incorrect for both portions (see Figure 9-6). Flat portions of meshes, as long as they are connected at their extreme edges to a non-flat surface, should not generate edges at all. Therefore, their normals should not influence the edge detection of other edges. For a cylinder, the correct normals on the ends are the same as the normals for a tube.

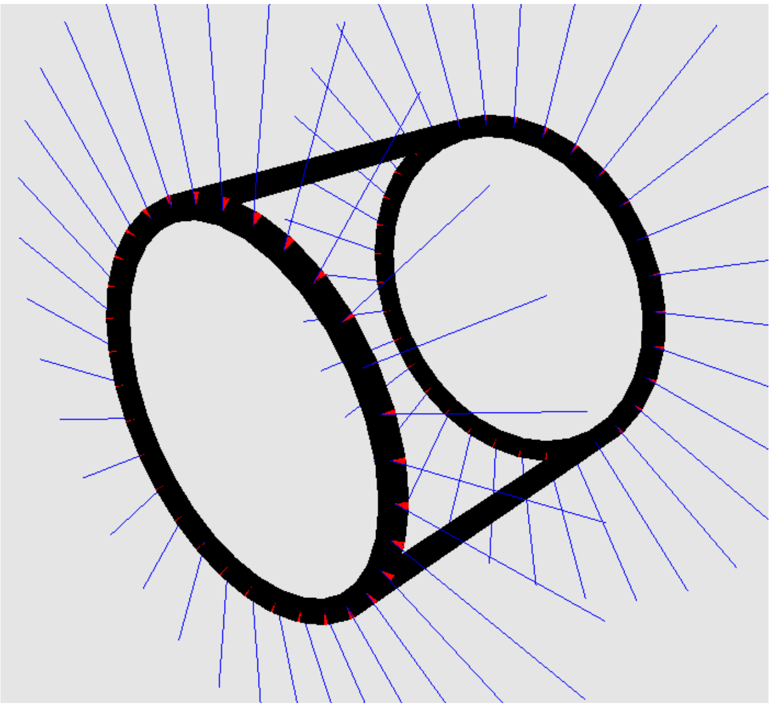


Figure 9-8: The tube's normals create the perfect normals for a cylinder. No matter the view angle, the normals of the flat surface were not needed to generate the correct caps.

Unfortunately, the flat to curved abutment is not the only case that will cause problems when the normals are projected. For example, when a plane is created, many 3D applications create the normals so that they all point upward out of the plane. However, to generate the appropriate caps for the boundary edges of the plane, the normals should lie on the same plane as the mesh itself, pointing diagonally outward.

Although the author was unable to find an algorithmic solution to generate appropriate normals for every kind of mesh at every view angle, he did find that the correct cap could always be generated with enough edge duplicates or a better normal chosen by a discerning eye.

9.3. Alternate Drawable Edge Types

The author explored several methods to reduce the amount of duplicate edge tests and other calculations by combining multiple passes together. The resulting new edge structures have various advantages and disadvantages over McGuire and Hughes' edge/cap drawing method. All of these new edge types render slightly faster than McGuire and Hughes' edges.

Half Hex Edges

Half hex edges combine the area of a half quad edge and the two half caps. They form a hexagon shaped edge on the "outside" of the mesh out of two quads. This combination eliminates one whole render pass, and reduces the number of output vertices need by two.

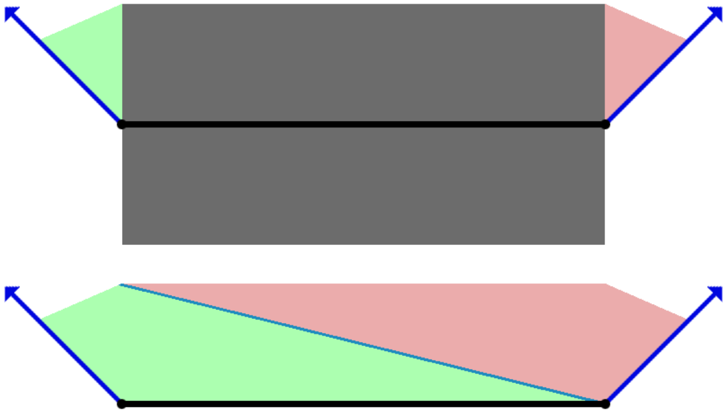


Figure 9-9: McGuire and Hughes' edge and two half caps setup (top) compared to a half hex edge (bottom). The blue arrows represent the vertex normals for the black edge. The half hex edge combines the caps into the edge, but is only rendered on the "outside" of the mesh.

The area in half hex edges, as with other other alternate edges, can be split into quads in multiple ways. For the sake of simplicity, only the split method displayed in the figure above will be used to describe the output vertices.

ID	Output Vertex (green side)
i = 0	`s0`
i = 1	`s1`
i = 2	`vec4((s0.xy + p.xy * sign(dot(m0, p))) / vec2(Width, Height) * s0.w, s0.zw)`
i = 3	`vec4((s0.xy + m0.xy) / vec2(Width, Height) * s0.w, s0.zw)`

ID	Output Vertex (red side)
i = 0	`vec4((s0.xy + p.xy * sign(dot(m0, p))) / vec2(Width, Height) * s0.w, s0.zw)`
i = 1	`s1`
i = 2	`vec4((s1.xy + m1.xy) / vec2(Width, Height) * s1.w, s1.zw)`
i = 3	`vec4((s1.xy + p.xy * sign(dot(m0, p))) / vec2(Width, Height) * s1.w, s1.zw)`

Half hex edges only render on the side of the m0 vector. Thus they are only suitable for use with contour edges. Noticeable edge flipping occurs if they are used for other types of drawable edges.

House Edges

House edges are named for their silhouette. They have a large base that extends above and below the polygon edge, and a “roof” that sits on top of the base. House edges require two new vertices not seen in other drawn edge types. They are the intersections of a line and two vectors. A special perpendicular vector is required for these calculations:

```
pAlt = p * sign(dot(p, m0));
```

The line is defined by two points:

```
innerS0 = s0 + pAlt;

innerS1 = s1 + pAlt;
```

The two vectors are two instances of the altered perpendicular vector, with origins at each of screen-space normal offset vertices:

```
pOriginPoint0 = s0 + m0;
```

```
pOriginPoint1 = s1 + m1;
```

The two additional vertices are calculated using the standard 2D intersection formula between vectors and a line. The other four vertices are calculated as in other techniques. They include: s0 offset by m0, s1 offset by m1, s1 offset by the “outward” p vector, and s0 offset by the “outward” p vector. See Figure 9-10 for a visual depiction of a house edge.

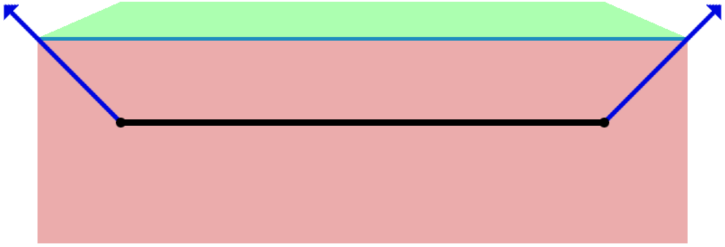


Figure 9-10: The house edge extends downward from the normal-offset points until the bottom of the inside edge is reached. This type of drawn edge covers much more area than other types.

The house edge generates very bad artifacts at even remotely sharp connecting edges. The extra area on the underside makes the problem of thick edges passing through nearby geometry even worse. This type of edge is not recommended for use in any circumstance, despite its slightly higher speed compared to McGuire and Hughes’ method.

Plug Edges

Plug edges, also named for their silhouette, combine the good qualities of half hex and the house methods. It uses only two passes, generates area on both sides of the edge, and doesn’t create too much extra area. It also creates only output vertices used in McGuire and Hughes’ edges, so the terrible artifacts of the house method are non-existent.

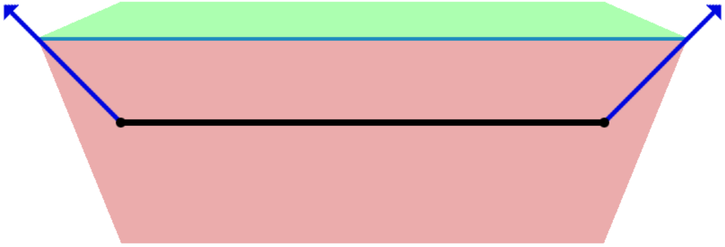


Figure 9-11: The plug edge takes the idea of the house edge, but simplifies it so that the points at the bottom match up with the actual extended edge point (vertex - perpendicular). This edge method generates a lot less area than the house edge, but retains the edge/cap combination.

Plug edges can be used for crease and boundary edges, while half hex edges can handle the contours. This combination generates quality roughly equivalent to McGuire and Hughes' edge/caps, but with the speed advantage of only two passes per mesh.

12. Conclusion

The depth-based edge thickness scaling factor provides more realistic edge shapes and reduces artifacts at almost no additional cost. Badly oriented normals can be prevented using duplicate edges for crease edges. For some meshes, with observation and testing, duplicate edges can be avoided by determining which normals to use and which to ignore. The plug and half hex methods of edge creation can be used to reduce the number of render passes by one, while retaining approximately the same quality as McGuire and Hughes' method of edge and cap creation.

The author's method of edge detection and capping is theoretically superior to McGuire and Hughes' method. It uses about the same amount of memory, yet produces higher accuracy caps at a faster overall speed. Unfortunately, implementation details make its real speed slower. Future optimizations and improvements in OpenCL memory management should remove that speed difference.

13. Future Work

McGuire and Hughes' paper mentions the use of geometry shaders and data texture access as a method to reduce the amount of duplicate information and calculations. These technologies are now available, and fully compatible with the new cap method developed for the author's method. Until the OpenCL memory issues are solved, geometry shaders and data textures can be used to implement the author's method. Texture caching and direct output to the pipeline would simplify and speed up the technique. However, it would also require that the mesh vertices be stored twice, once in the form of data textures.

Implementing a half quad-like version of the author's method will provide more clean results when dealing with contour edges. Without the normals, however, determining the appropriate drawing side for the edges is a more difficult problem, since normals are not used. The cap creation process will also need to change to deal with half thickness edges.

In relation to OpenCL, various memory management optimizations in the design of the OpenCL kernels should increase the speed significantly. Preprocessing the mesh into patches of related vertices that can be cached to a local work group of kernels is one such optimization. OpenCL is a new technology. Its automatic memory management will improve with time, further increasing the speed.

Microsoft's DirectCompute shaders have many of the same abilities as OpenCL and the author's method could be implemented using that technology. One advantage it has over OpenCL is the ability to export calculations to the graphics pipeline. This will allow the edge and cap rendering passes to be integrated into the computation passes. That ability to skip two render passes should speed up the framerate slightly.

Bibliography

[Aila and Miettinen 04] Aila, Timo and Ville Miettinen. “dPVS: An Occlusion Culling System for Massive Dynamic Environments.” IEEE Computer Graphics and Applications, vol. 24, no. 2. 86-97. March.

[Akenine-Möller et al. 08] Akenine-Möller, Tomas, Eric Haines, and Naty Hoffman. 2008. “Non-Photorealistic Rendering.” In Real-Time Rendering, Third Edition. 507-530. A K Peters, Ltd.

[Bosch 06] Bosch, Marc ten. 2006. “Real-Time Hardware-Determined Feature Edges.” http://www.marctenbosch.com/npr_edges/.

[Buchanan and Sousa 00] Buchanan, John W., and Mario C. Sousa. 2000. “The edge buffer: A data structure for easy silhouette rendering.” ftp://ftp.cs.ualberta.ca/pub/juancho/edge.pdf.

[Card and Mitchell 02] Card, Drew and Jason L. Mitchell. 2002. "Non-Photorealistic Rendering with Pixel and Vertex Shaders." http://developer.amd.com/media/gpu_assets/ShaderX_NPR.pdf.

[Decaudin 96] Decaudin, Philippe. 1996. "Cartoon-Looking Rendering of 3D-Scenes." http://www.antisphere.com/Research/Publis/RR-2919-en.pdf.

[Evans 03] Evans, Christopher. 2003. “Thick Lines for Real-Time Cel Shading.” http://www.chrisevans3d.com/tutorials/cel_lines/.

[Everitt 00] Everitt, Cass. 2000. “One-Pass Silhouette Rendering with GeForce and GeForce2.” NVIDIA Corporation White Paper.

[Gooch and Gooch 01] Gooch, Bruce and Amy Gooch. 2001. “Feature Edges: Silhouettes, Boundaries, and Creases” and “Automatic Systems: Illustration (Artistic Line Drawing).” In Non-Photorealistic Rendering. 117-159. A K Peters, Ltd.

[Gooch et al. 99] Gooch, Bruce, Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Richard Riesenfeld. 1999. “Interactive Technical Illustration.” http://www.ppsloan.org/publications/iti99.pdf.

[Hall 03] Hall, Tom. 2003. “Silhouette Tracking.” http://www.bytegeistsoftware.com/various/SilhouetteTracking.pdf.

[Hertzmann 99] Hertzmann, Aaron. 1999. "Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines." http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.9731&rep=rep1&type=pdf.

[Hertzmann and Zorin 00] Hertzmann, Aaron, and Denis Zorin. 2000. “Illustrating smooth surfaces.” http://mrl.nyu.edu/~dzorin/papers/hertzmann2000iss.pdf.

[HIPR 00] Hypermedia Image Processing Reference (HIPR). Robert Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. 2000. http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm, http://homepages.inf.ed.ac.uk/rbf/HIPR2/roberts.htm.

[Isenberg et al. 03] Isenberg, Tobias, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. 2003. "A Developer's Guide to Silhouette Algorithms for Polygonal Models." IEEE Computer Graphics and Applications, vol. 23, no. 4. 28-37. July.

[Kroon 09] Kroon, Dirk-Jan. 2009. "Numerical Optimization Of Kernel Based Image Derivatives." http://www.k-zone.nl/Kroon_DerivativePaper.pdf.

[Lander 01] Lander, Jeff. 2001. “Images from deep in the programmer's cave.” Game Developer. May. 23-28.

[Markosian et al. 97] Markosian, Lee, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. 1997. “Real-Time Nonphotorealistic Rendering.” ftp://ftp.cs.brown.edu/pub/papers/graphics/research/sig97-npr.pdf.

[Marshall 01] Marshall, Carl. 2001. “Cartoon Rendering: Real-time Silhouette Edge Detection and Rendering.” In Game Programming Gems 2, ed. Mark DeLoura. 436-443. Charles River Media, Inc.

[McReynolds and Blythe 99] McReynolds, Tom and David Blythe. 1999. "Advanced Graphics Programming Techniques Using OpenGL." SIGGRAPH `99 Course. http://www.opengl.org/resources/code/samples/sig99/advanced99/notes/node108.html.

[Mitchell 02] Mitchell, Jason L. 2002. “Image Processing with 1.4 Pixel Shaders in Direct3D.” http://developer.amd.com/media/gpu_assets/ShaderX_ImageProcessing.pdf.

[Northrup and Markosian 00] Northrup, J.D., Lee Markosian. 2000. “Artistic Silhouettes: A Hybrid Approach.” http://graphics.cs.brown.edu/research/art/artistic-sils/artistic-sils-300dpi.pdf.

[McGuire and Hughes 04] McGuire, Morgan and John F. Hughes. 2004. “Hardware-Determined Feature Edges.” http://graphics.cs.williams.edu/papers/EdgesNPAR04/edges-NPAR04.pdf.

[Raskar 01] Raskar, Ramesh. 2001. "Hardware Support for Non-photorealistic Rendering." http://www.cs.unc.edu/~raskar/HWWS/raskarHardwareNPR2001.pdf.

[Raskar and Cohen 99] Raskar, Ramesh and Michael Cohen. 1999. "Image Precision Silhouette Edges." http://www.cs.unc.edu/~raskar/NPR/sil-i3d99.pdf.

[Rossignac and Emmerik 92] Rossignac, Jarek R. and Maarten van Emmerik. 1992. "Hidden contours on a frame-buffer." http://www.gvu.gatech.edu/~jarek/papers/Hidden.pdf.

[Saito and Takahashi 90] Saito, Takafumi and Tokiichiro Takahashi. 1990. "Comprehensible Rendering of 3-D Shapes." ACM: Computer Graphics, Volume 24, Number 4. August. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.83.4139&rep=rep1&type=pdf.

[Sander et al. 00] Sander, Pedro V., Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. 2000. “Silhouette Clipping.” http://research.microsoft.com/en-us/um/people/hoppe/silclip.pdf.

Image Credits

Viewtiful Joe Screenshot - Viewtiful Joe. 2003. Developed by Capcom Production Studio 4. Published by Capcom. Image from The Next Level (http://www.the-nextlevel.com/reviews/gamecube/viewtiful_joe_import/).

Ōkami Screenshot - Ōkami (Wii). 2008. Developed by Ready at Dawn. Published by Capcom. Image from Binge Gamer (http://www.bingegamer.net/2008/28-new-okami-wii-screenshots/).

Bomb Image - Evans, Christopher. “Thick Lines for Real-Time Cel Shading.” 2003. Image from ChrisEvans3D.com (http://www.chrisevans3d.com/tutorials/cel_lines/).