Real-Time Object-Space Edge Detection using OpenCL

Dwight House
Master of Science in Computer Science
dwighthouse@gmail.com

Dr. Xin Li
Dean of Faculty, Director of Education
DigiPen Institute of Technology
xli@digipen.edu

Abstract

At its most basic, object-space edge detection iterates through all polygonal edges in each mesh to find those edges that satisfy one or more edge tests. Those that do are expanded and rendered, while the remainder are ignored. These 3D edges, and their resulting accuracy and customizability, set object-space methods apart from all other categories of edge detection. The speed and memory limitations of iterating through all polygonal edges in each mesh each frame has inspired optimization research.

In this paper, we explore methods to calculate object-space edges utilizing programmable GPU technologies, including OpenCL. The OpenCL methods explored allow for a significant reduction in calculation quantity. Some also provide a reduction in rendering artifacts and memory usage over previous GPU techniques. Unfortunately, most uses of OpenCL for edge detection results in slower performance than shader-based techniques, though variations and optimizations may reduce this disadvantage in the future.

Background

Edge detection and rendering is an important non-photorealistic rendering technique. Rendered edges can be used for a multitude of purposes including object differentiation, structural enhancement, highlighting, and blur-based anti-aliasing. They also are a required component of several graphical styles such as toon rendering and sketchy rendering.

Edge Types

There are several types of edges. Each type requires different detection tests, and may not be detectable with a given method.

- **Contour:** A polygon edge that connects two polygons, one front-facing and the other back-facing.
- **Crease:** A polygon edge that connects two polygons that are within some user-defined angular distance of each other.
- **Boundary:** A polygon edge that forms the side for only one polygon.
- **Intersection:** A collision of two polygons such that the line formed along the intersection is a polygon edge of only one or neither of the colliding polygons.
- **Marked:** A polygon edge flagged to always be rendered as an edge.

For the purposes of this paper, we ignore further discussion of intersection edges. Intersection edges are not detectable with object-space methods.

Edge Detection Method Categories

Object-space edge detection refers to one category of edge detection. The other categories are Hardware, Image-space, and Miscellaneous.

- **Hardware:** In these methods, edges are not detected, but directly rendered as a byproduct of some other operation, or series of operations. These methods are very fast, available on almost all hardware, and do not require mesh preprocessing or additional memory. However, they typically only render contours and they lack customizability.
- **Image-space:** In this method, two rendering passes are used. In the first, scene data, like depth and normal information, is rendered and stored to the GPU temporarily. In the second pass, the stored data is used with image-processing techniques to determine areas of rapid change, which generally correspond to edges. This method has constant-speed edge detection, because the speed of the second pass is dependent on the number of pixels in the viewport, not the number of objects rendered. It can also detect edges easily that other methods cannot. However, the edge accuracy is lacking and the edge thickness is inconsistent. The image-space, like the hardware methods, does not produce edges that are readily customizable.
- **Object-space:** In these methods, some or all of the polygon edges of each mesh are tested for their drawability. Those edges that pass the test are then expanded into quads or other structures, where they are rendered like any other geometry. Those polygon edges not drawable are discarded. Since the edge detecting occurs in 3D space, the results are more accurate. The output is 3D geometry, so the edges can be customized to a great degree, including texturing, animating, and shading. However, the number of tests and rendering operations for each edge causes object-space edge detection to be the slowest form of edge detection.
- **Miscellaneous:** These methods share no direct similarities with each other, though in terms of their positive and negative qualities, they resemble hardware methods.

In this paper, we only focus on object-space edge detection, as it is the method used by this paper's OpenCL-based edge detection.

OpenCL

OpenCL, which stands for Open Computing Language, creates a standardized interface for computational tasks on a variety of devices. It is specifically focused on data-parallel tasks, which require a single set of relatively simple operations be performed on massive quantities of nearly atomic data. This is essentially the form of calculation found in GPU rasterization. Not surprisingly, GPUs are often the preferred device for utilizing OpenCL. Like vertex shaders, an OpenCL program (called a kernel) can access data stored on the GPU and perform operations on it before creating output. However, OpenCL is more free in what it can do and what it can access. Though more versatile, without careful planning an OpenCL operation can take far longer than the equivalent operation implemented as a shader.

Previous Work

Edge detection algorithms are very old and well documented. However, the paper by Markosian et al. [Markosian et al. 97] represents some of the first work in real-time edge detection. They stored edge adjacency information for each polygon edge. Contour edges tend to form loops around a mesh, so when a contour edge was found, they recursively performed the contour edge test on the connecting edges first. This method tends to detect the longest, and therefore most significant, edges with a minimal number of random tests. Additionally, a small portion of contour edges detected each frame were also stored for the next frame as starting points for the next round of edge tests. Without sudden movements, a significant number of contour edges remain the same from frame to frame. By checking only a very small percentage of all edges, they found a fivefold increase in the rendering speed over testing all polygon edges individually. Of course, some contour edges could be missed entirely from frame to frame, possibly resulting in flickering.

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.

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 list of edges to test. If a mesh was constructed using primarily quads, this optimization can reduce the number of edge tests significantly: over 20% in the case of the Utah Teapot.

Other imaginative edge storage and detection methods 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. This method worked especially well with highly tessellated meshes.

Finally, 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 drawable, 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. Their paper formed the basis for OpenCL edge detection research.

Hardware Determined Feature Edges

We 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, we detail here McGuire and Hughes' work as it relates to our contributions.

McGuire and Hughes' goal was to transfer all detection and rendering of edges to the GPU, 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

Data Structure

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

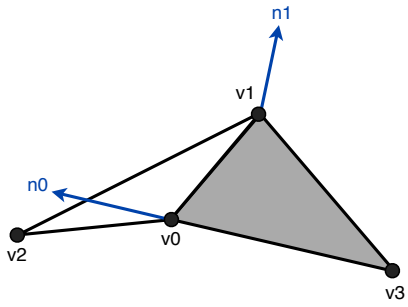

Figure 1: The important geometric values for each edge in McGuire and Hughes' method: v0, v1, v2, and v3 are vertices on two polygons that share an edge. n0 and n1 are the vertex normals for v0 and v1, respectively. v3 may not exist in the case of a boundary edge.

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

The r parameter is ignored for this paper. The i parameter is used to pick what value to output after an edge is detected drawable. If the edge is a boundary edge, there is no v3 vertex. In that case, v3 is 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.

Edge Detection

Contour edges can be detected by finding the dot product of both adjacent polygons' face normals with the view vector. If the sign of the two values is different, the polygon edge is a contour. Crease edges can be detected using the face normals of the adjacent polygons as well. Since the face normals' angular distance is inversely proportional to the angular "sharpness" of the two adjacent polygons, polygon edges are found to be creases if the dot product of the two face normals is less than the negative cosine of the user-defined angle. This will detect crease edges that are any sharper (smaller angular distance between adjacent polygons) than the user-defined angle. Finally, boundary and marked edges are detected by checking v0 and v3 for equality. The detection tests are described algorithmically below.

- **Contour:** [(DotProduct(Na, V) * DotProduct(Nb, V)) < 0]
- **Crease:** [DotProduct(Na, Nb) < -cos(θ)]
- **Boundary and Marked:** [v0 == v3]

Where Na and Nb are the face normals of the two adjacent polygons, V is the view vector, and θ is the user-defined angle.

Once an edge is detected, one of several rendering methods can be used to create a viewable edge feature.

Edge Rendering

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 vertex 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.

McGuire and Hughes create edges, and later caps, 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
- **m1:** n1 in screen-space
- **p:** normalized perpendicular vector to the screen-space edge vector (s1 - s0)

These values are calculated via:

s0
```
vec4 s0 = MVP * vec4(v0.xyz, 1.0);
s0.xy = (s0.xy / s0.w) * vec2(Width, Height);
```

s1
```
vec4 s1 = MVP * vec4(v1.xyz, 1.0);
s1.xy = (s1.xy / s1.w) * vec2(Width, Height);
```

m0
```
vec4 temp = MVP * vec4(v0.xyz + n0.xyz, 1.0);
temp.xy = (temp.xy / temp.w) * vec2(Width, Height);
vec2 m0 = normalize(temp.xy - s0.xy);
```

m1
```
vec4 temp = MVP * vec4(v1.xyz + n1.xyz, 1.0);
temp.xy = (temp.xy / temp.w) * vec2(Width, Height);
vec2 m1 = normalize(temp.xy - s1.xy);
```

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

Where Width and Height are the width and height of the viewport, respectively. The above functions use component-based division and swizzle vector operations. The p vector's length can be modified to adjust the screen-space thickness of rendered edges.

Full Quad Edges

Full quad edges render on both sides of the screen-space edge, as seen in figure 2. On contour edges, the "inner" half would penetrate the mesh itself, which can sometimes lead to artifacts. However, for most marked and boundary edges,

and all crease edges that are not also contours, these are the preferred output type. Using the i value, one of four calculations listed below are performed on the input edge vertex to create the output vertex. Note that the output vertices are converted from screen-space back to projection-space, which is the expected output format for vertex shaders.

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)`

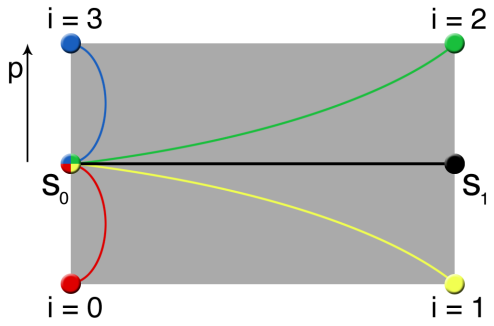

Figure 2: Depending on the i value, the output point is calculated using the two screen-space edge points and the perpendicular vector.

Half-Quad Edges

Half-quad edges are only suitable for contour edge rendering. To ensure that the half-quad is rendered on the "outside" of the mesh, the p value is modified so it always points in the same direction as the m0 vector via the sign function. As with the full quad method, the i value is used to determine which of four output values to create. Figure 3 provides an illustration of the output.

i = 0 `vec4(s0.xy / vec2(Width, Height) * s0.w, s0.zw)`

i = 1 `vec4(s1.xy / vec2(Width, Height) * s1.w, s1.zw)`

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)`

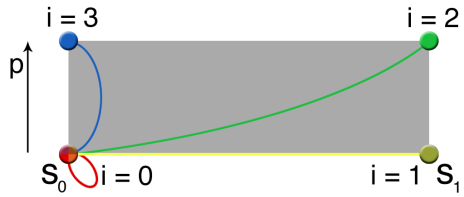

Figure 3: The p vector is modified to point outward, so the edge half-quad will only render on the outside of the mesh.

Non-Drawable Edges

If an edge fails all the edge tests, it is a non-drawable edge. Optimally, these edge vertices would be removed from the pipeline at this point, since they do not create valid output. However, vertex shaders cannot delete vertices. McGuire and Hughes solved this issue by outputting a single vertex for all i values: (0, 0, -1, 1). These vertices will not only be degenerate, but in projection-space they are behind the camera and will be clipped before reaching the fragment shader.

Edge Caps

Edges rendered with a thickness of greater than a few pixels will cause visible gaps to occur at many of the edge connection points, as in the left side of figure 4. McGuire and Hughes accounted for these by rendering half-caps on both ends of each drawable edge, connecting at a point along the screen-space normal of the side in question to create a cap that fills the gap completely (figure 4 on right). The half-caps can be generated from the same edge mesh data as the edge, 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.

As mentioned earlier, the two caps are generated in separate render passes: the first deals with the v0 side and the second deals with the v1 side. They are rendered as triangles, rather than quads. The output vertices are described below.

Half-Cap Output On v0 Side

i = 0 `vec4(s0.xy / vec2(Width, Height) * s0.w, s0.zw)`

i = 1 `vec4((s0 + p * sign(dot(m0, p))) / vec2(Width, Height) * s0.w, s0.zw)`

i = 2 `vec4((s0.xy + m0) / vec2(Width, Height) * s0.w, s0.zw)`

Half-Cap Output On v1 Side

i = 0 `vec4(s1.xy / vec2(Width, Height) * s1.w, s1.zw)`

i = 1 `vec4((s1 + p * sign(dot(m1, p))) / vec2(Width, Height) * s1.w, s1.zw)`

i = 2 `vec4((s1.xy + m1) / vec2(Width, Height) * s1.w, s1.zw)`

The p, m0, and m1 vectors should all be scaled by the same value used when scaling the edge if the user desires a specific screen-space thickness.

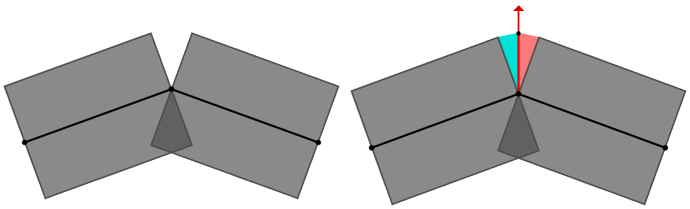

Figure 4: On the left, two thick screen-space edge quads connect to create a visible gap. Using the screen-space projection of the vertex normal they share, the gap can be filled with a cap formed from one half-cap from each edge (one blue, one red), as seen on the right.

Issues

McGuire and Hughes' edge and capping method are highly effective and very fast on modern hardware, but they have a few drawbacks.

- **Calculation Duplication:** Each polygon edge is tested for drawability ten times: four for the edge and three for each half-cap. Once an edge is determined drawable, all the calculations necessary to create the output vertices is also duplicated. Geometry shaders usage was proposed as a solution, since it can output more than one vertex per invocation.
- **High GPU Memory Usage:** The edge mesh also requires at least seventy-two extra floats and four integer values for every polygon edge. All of this information is already on the GPU in the form of vertices, vertex normals, and indices. McGuire and Hughes suggested using data textures to reduce the memory usage by a factor of four.
- **Incorrect Caps:** 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.
- **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.

Using OpenCL can help with the first three issues. The last issue is dealt with in the appendix.

Object-Space Edge Detection Using OpenCL

After researching and implementing McGuire and Hughes' method, we attempted to improve upon it. The most extensive research focused on finding a replacement technique that would take advantage of the new capabilities of OpenCL.

Relevant OpenCL Capabilities

OpenCL has several abilities that make it more flexible than shaders. Those that are relevant to the detection of edges are listed below.

- **OpenGL Buffer Interoperability:** An instance of OpenCL, if built using an OpenGL context, will have direct access to buffers from OpenGL for both

reading and writing. However, locking the buffers is necessary in some cases.

- **Read Freedom:** With access to all buffers on the GPU, OpenCL can access multiple buffers from within the same kernel. OpenCL can also request data from buffers out of order, unlike shaders.
- **Write Freedom:** As long as there is enough space, a single OpenCL kernel can output any number of values to any number of buffers.

With these abilities, an OpenCL-based edge detection algorithm can make use of data already on the GPU, removing the need for data duplication. The calculations also need not be repeated, since multiple values can be output from a single kernel.

Edge Data Structures

Because almost all the data needed for the edge detection step already exists on the GPU for normal mesh rendering, OpenCL's edge mesh structure need only contain the vertex indices for each edge's adjacency information.

- **i0:** index of first vertex in the edge
- **i1:** index of second vertex in the edge
- **i2:** index of final vertex that, with i0 and i1, indexes the first polygon
- **i3:** index of final vertex that, with i0 and i1, indexes the second polygon

These values can be used to index directly into the vertex and normal buffers. We only need one copy of them because OpenCL can output multiple values from the same kernel. This can represent a significant reduction in the memory requirements of the edge detection.

Because OpenCL cannot export vertices to the rendering pipeline directly, the output values, both edge quads and degenerate quads must be temporarily stored to the GPU's memory for rendering in another pass. This edge out buffer must be created to hold 4E four dimensional vertices (or 16E floats) values, where E is the number of edges in the mesh.

Edge Detection and Rendering

The edge detection tests are exactly the same as those in McGuire and Hughes' method with one exception. Since index values are available, the boundary and marked edge tests can be accomplished by checking for index equality of i0 and i3, rather than checking the indexed vertices.

As before, if the edge is not drawable, a degenerate quad should be exported to the edge out buffer made up of the vertex (0, 0, -1, 1). For drawable edges, they should be expanded as in McGuire and Hughes' method before being exported. In both cases, all the vertices in the projection-space quads are exported in a single step to their appropriate location in the edge out buffer, using the same identifying index used to get the input edge information.

The edge detection kernel will need access to the ModelViewProjection matrix and the viewport dimensions to calculate the screen-space position of the vertices. Since OpenCL is not part of the shading environment, these values must be manually sent to the kernel each frame. The edge out buffer, after the edge detection step is complete, will contain sets of four projection-space vertices representing the edge quads or degenerate quads. Therefore, when rendering the edge out buffer as quads, the vertex shader should apply no transforms on the vertices at all.

McGuire and Hughes-based Capping

Beyond edges, the McGuire and Hughes' capping method can also be implemented in OpenCL. In fact, it requires no additional data to integrate it into the edge detection system described above. It only requires two additional output buffers capable of storing at least 3E four dimensional vertices (12E floats) each, where E is the number of edges in the mesh. Again, this is due to OpenCL's inability to export data directly to the graphics pipeline.

If the edge was detected drawable, the two edge half-caps can be immediately generated. The normal information needed can be accessed using the index values i0 and i1 on the normal buffer. From then on, the vertex creation process is identical. As with the edges, the rendering passes for the cap buffers can simply pass their vertex data through the vertex shader.

Edge Adjacency Capping

OpenCL's abilities to access and manipulate many buffers provides for another method of edge capping. In McGuire and Hughes' caps, if the projected normal did not correspond well to the curvature of the mesh, caps could be generated on the wrong side of the edge. With access all edges, we can make a list of connected edges. Using this, the capping process is no longer a blind activity that generates two half-caps per drawable edge. Instead, caps are created relative to the vertex to which they connect. With both edges connecting at that vertex available, we can determine exactly which caps should be created, and exactly what size to make them so they always fill the gap perfectly. This eliminates the need for vertex normals and prevents the problems with caps being generated incorrectly. A number of edge orientations and the caps they form are displayed in figure 5.

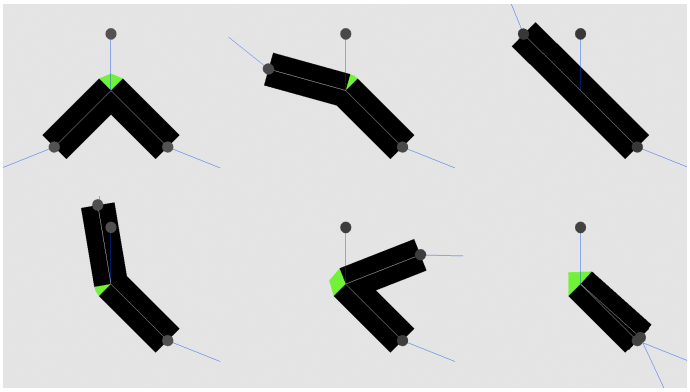

Figure 5: No matter the orientation of the two edges or the normal, a perfect cap is formed to fill the gap every time.

A cap buffer must be preprocessed. It stores indices like the edge buffer. However, the indices reference index sets in the edge buffer. This makes a single potential cap from every pair of connected edges. Each vertex in this "cap mesh" contains:

- **e0:** index of first edge
- **e1:** index of second edge

As with the edge buffer, the cap buffer needs a corresponding cap out buffer to store the output caps for later rendering. It must be big enough to hold 4C four dimensional vertices (16C floats), where C is the number of cap vertices in the cap buffer.

Cap Detection

In this method of capping, only those caps for whom both connected edges are drawable should be drawn. Thankfully, because the edge detection step must store its output data temporarily, it is trivial to determine if both are drawable. After obtaining the projection-space edge quads from both connected edges, drawability can be determined by checking one of the vertices in each for equality with the degenerate vertex (0, 0, -1, 1). To reduce the test to a single comparison, during the edge test, degenerate quads can be exported that use the vertex (NaN, 0, -1, 1). Then drawability can be determined by checking the x component of the first vertex of each quad for being NaN (Not a Number). The values remain degenerate in this variation, so they will not generate artifacts when rendering the edges.

Cap Creation and Rendering

If both edges are drawable, the cap should be created. The projection-space edge quads are used to determine where to place the cap. However, there is no guarantee of the edges' orientations. We must ensure they are aligned with each other, which for this paper means that the i1 side of edge one is connected to the i0 side of edge two. This is easily done by using the cap buffer's indices to look up the edge indices for each edge and determining if one or both edges' output values should be flipped before use.

Once aligned, the projection-space edge points can be transformed to screen-space using homogeneous division and multiplication by the viewport width and height. At this point, the values can differ significantly depending on the usage of half-quads. If half-quads were used in addition to full quads, a large amount of logic or additional data is needed to successfully determine the exact connection point for the aligned edges. For the sake of time, we assume that only full quads were used.

For the first edge quad, the point halfway between the first and last screen-space edge points represents the left edge point, called sL. The halfway point between the second and third screen-space edge points for the first edge quad is the point of connection for the two edges, called sM. Finally, the halfway point between the second and third screen-space edge points of the second edge quad represent the right edge point, called sR.

The "middle vector" can be determined from sL, sM, and sR. It takes the place of the screen-space normal in the cap creation process. It is the vector bisecting the two edge vectors, inverted. It is calculated via the formula below.

```
vec2 middleVector = -normalize(normalize(sL.xy - sM.xy) +
    normalize(sR.xy - sM.xy));
```

Once the middle vector is obtained, the only remaining required values are the two perpendicular vectors, one for each edge. After calculation, they have to be aligned to point in the same direction as the middle vector. They are calculated as below.

```
vec2 p0 = normalize((vec2)(sL.y - sM.y, sM.x - sL.x));
p0 = p0 * sign(dot(p0, middleVector));

vec2 p1 = normalize((vec2)(sR.y - sM.y, sM.x - sR.x));
p1 = p1 * sign(dot(p1, middleVector));
```

With all these values available, the four cap vertices can be created, as listed in the table below. These are all exported to the cap out buffer in one pass. As with the edges, those caps that should not be drawn should output a degenerate quad using the vertex (0, 0, -1, 1).

```
0  vec4((sM.xy + middleVector) / sM.w * vec2(Width,
       Height), sM.zw)
1  vec4((sM.xy + p0) / sM.w * vec2(Width, Height), sM.zw)
2  vec4(sM.xy / sM.w * vec2(Width, Height), sM.zw)
3  vec4((sM.xy + p1) / sM.w * vec2(Width, Height), sM.zw)
```

Comparative Analysis

We compare the memory usage, speed, and rendered output of McGuire and Hughes' method in shaders, McGuire and Hughes' method in OpenCL, and our new OpenCL method.

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. We compare the number of bits required to store the data necessary both to calculate the edges and caps, and in the case of OpenCL, the bits required to temporarily store the output for later rendering. The shader implementation must have some memory to store the output of its calculations, but it is hidden by the implementation and not directly measurable. 32-bit float and integer types were assumed. Values are in bits.

McGuire & Hughes (shader)		McGuire & Hughes (OpenCL)		New OpenCL	
Edge	*Cap*	*Edge*	*Cap*	*Edge*	*Cap*
2688	96	640	512	640	576

At this point, the edge to cap ratios differ. The number of caps necessary to test is significantly higher for the new OpenCL capping method, whereas in both methods that implement McGuire and Hughes' capping, the cap to edge ratio is always 2. To get a good sample, a variety of meshes were tested, rendered from a camera position of (5, 5, 5) with the unit-sized mesh at the origin. From this, the below table illustrates the total number of bits used for each type of object in all three methods.

	McGuire & Hughes (shader)	McGuire & Hughes (OpenCL)	New OpenCL
Cube	69120	39936	29184
Cylinder	276480	159744	245760
Cone	184320	106496	381952
Quad Sphere	5806080	3354624	5289984
Ico Sphere	5529600	3194880	6741120
Teapot	3398400	1963520	3301120
Monkey	4173120	2411136	5067648
Bunny	59938560	34631168	75118720

By converting these bit requirements relative to the amount used by our base case (McGuire and Hughes' algorithm in shader), we get a list of memory usage ratios.

	M&H (OpenCL) to M&H (Shader)	New OpenCL to M&H (Shader)
Cube	0.577	0.422
Cylinder	0.577	0.888
Cone	0.577	2.07
Quad Sphere	0.577	0.911
Ico Sphere	0.577	1.21
Teapot	0.577	0.971
Monkey	0.577	1.21
Bunny	0.577	1.25

Implementing McGuire and Hughes' algorithm on the GPU, even with the extra temporary storage requirements, yields a significant memory savings. The new OpenCL capping method, on the other hand, varies significantly based on the number of connections per point in the mesh. Overall, the memory usage is about the same.

Speed

To measure the speed of the three methods, we compare the framerates from the same view direction for the same set of example objects.

	McGuire & Hughes (shader)	McGuire & Hughes (OpenCL)	New OpenCL
Cube	1173	762	760
Cylinder	1103	735	733
Cone	1244	670	724
Quad Sphere	713	584	416
Ico Sphere	739	593	360
Teapot	850	631	464
Monkey	770	634	334
Bunny	134	159	31

Clearly, the shader implementation is the fastest for most meshes. Only for very complicated meshes is the McGuire and Hughes' method implemented in OpenCL faster than the shader. All tests of the new OpenCL capping method resulted in significantly slower speeds than the shader-based edge detection.

The reasons for the speed differences are numerous. OpenCL cannot short-circuit code, so the worst-case calculation path is effectively always used. The new OpenCL capping method relied upon short-circuiting for its theoretically faster speeds. McGuire and Hughes' algorithm did not rely on short-circuiting, and so had effectively fewer instructions than the new OpenCL method. Additionally, memory access in the shader is always cached, because it is accessed in order. Both OpenCL methods access the memory out of order to save on memory usage. This prevents the manual caching methods from being possible. Other issues, such as render deferment and the newness of the OpenCL API likely contributed to the slower results.

Rendered Output

McGuire and Hughes' algorithm, whether implemented via shader or OpenCL, generate identical output. In this section, we compare their quantity of rendered items and the quality of the output with the new OpenCL capping method.

Rendered Quantity

The number of rendered objects effects the final rendering speed to some degree. All three methods generate the same number of edge quads from a given angle. However, though the new OpenCL capping method has about double the number of caps to check, the number of output caps is usually about half that of McGuire and Hughes' capping method.

Rendered Quality

Figure 6 shows a sample object rendered with full quads. The new OpenCL method on the left shows all edge gaps completely filled. The other artifacts are from edge quads pushing through the mesh. On the right, McGuire and Hughes' method of capping is shown. Notice the missing caps on the eyebrows and extraneous caps sticking off the ends of other edges.

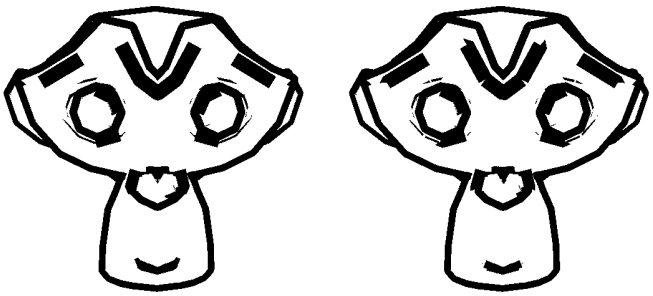

Figure 6: On the left, the new OpenCL capping method has far fewer artifacts than the McGuire and Hughes' capping method displayed on the right.

The quality can be further enhanced with the use of half-quads on contour edges, but this would result in a further slowdown due to the complexity of calculating the correct caps when there is such variability in the shape of connecting edges.

Conclusion

Our new method of capping is more accurate, but much slower. It also has no real memory advantages over the shader implementation. However, using OpenCL to implement McGuire and Hughes' algorithm holds much promise. It saves a lot of memory and can render faster than shaders for some very complicated meshes. Future improvements to OpenCL and the algorithm may result in even higher speeds and better performance for simpler meshes.

Future Work

OpenCL was used in this paper to take the place of data texture lookups and geometry shader usage suggested by the future work of McGuire and Hughes' paper. Further, texture memory access is cached on GPUs, unlike other buffers. The new OpenCL capping method could be implemented via geometry shaders and data textures, which provides the higher accuracy of the new capping method with speed closer to those of the shader-based algorithm.

If one implemented McGuire and Hughes' algorithm on the GPU such that the input data was duplicated rather than indexed, OpenCL's manual caching methods could be used. This should result in a much higher speed, but at the cost of higher memory usage.

Microsoft's DirectCompute shaders have many of the same abilities as OpenCL and any of the discussed methods could be implemented using that technology. One advantage it has over OpenCL is the ability to export data directly 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 could increase the speed slightly.

Chris Peters [Peters 10] suggested a capping method that preprocessed edges so that both ends contain a "next" index. The next index points to an adjacent edge connected to the same end point on a triangle. After doing the edge compute pass, each drawable edge would be operated on twice, once for each end. At each end of each drawable edge, the next index would be used to check the next edge for its drawability. If it is drawable, a cap is formed between those two edges. If it is not drawable, the next index of the second edge is checked, and so on, until either a drawable edge is found, or the original edge is reached. If a drawable edge connects to no drawable edges, no cap is drawn. This system should give the same accuracy caps as new capping method, but without needing to store or check every possible combination. The worst case number of memory accesses is C + 1 + DATA, where C is the number of connecting edges and DATA is the other drawable edge's data needed to form the cap. This technique would still use uncached GPU memory, but requires vastly fewer memory accesses for the capping operation. This method could also potentially lead to faster speeds for more complex meshes.

References

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

[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.

[Peters 10] - Peters, Chris. 2010. Personal Communication.

[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.

Appendix: Depth-Scaled Edge Thickness

When scaling an edge's thickness in screen-space, two types of artifacts occur. First, perspective scaling is lost on objects, possibly confusing the user about the distance of the object. Second, as a mesh gets further from the camera, the edges will take up such a large portion of the displayed area that they overpower the mesh itself, as seen in Figure 7.

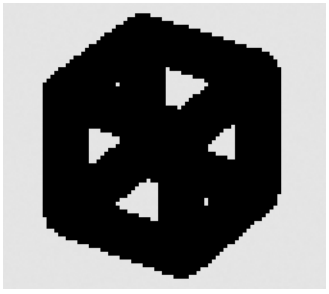

Figure 7: This cube is far away from the camera, yet the edges remain the same pixel width, overpowering the mesh.

To prevent this, we introduce a depth-based scaling factor into the width calculation of the edges and caps. This value would be multiplied into the perpendicular vector, screen-space normal vectors, and the middle vector at the time of output.

In the process of creating the values necessary to create the edge, the edge points v0 and v1 are taken from model-space, through view-space, and into projection space, before later being sent to screen-space. Since the projection-space location is known, we can get the view-space depth from the projection-space z coordinate multiplied through the inverse projection matrix. As it turns out, the inverse projection matrix as applied to the z coordinate is the projection-space w coordinate multiplied by -1. To transform this value into a proper scaling factor, it must be inverted and negated due to the orientation of the camera in view-space. Further, we can prevent the creation of edges thinner than a single pixel by capping the minimum value at 1. The resulting scaling factor is displayed below:

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

A different scaling factor is needed for both ends of the edge. In the equation above, projected represents the s0 or s1 values before they have been sent to screen-space.

The user can then freely change the value over the projected point's w to whatever custom thickness scaling factor they desire. Edges and caps will realistically grow smaller as they grow more distant, but never fully disappear.