Edge Detection with OpenCL

By Dwight House

October 20th, 2010

Quick Life Story

- Born in Shreveport, lived in Haughton
- 2007 Bachelor of Science in Computer Science
  - Louisiana State University Shreveport
- 2010 Master of Science in Computer Science
  - DigiPen Institute of Technology
  - Focused on game programming and graphics
- Now seeking employment

Talk Overview

1. Rendering Overview
2. Terminology and Properties
3. Overview of Edge Detection Methods
4. Research Inspiration
5. OpenCL Edge Detection
6. Results Analysis
7. Demos
8. Q and A

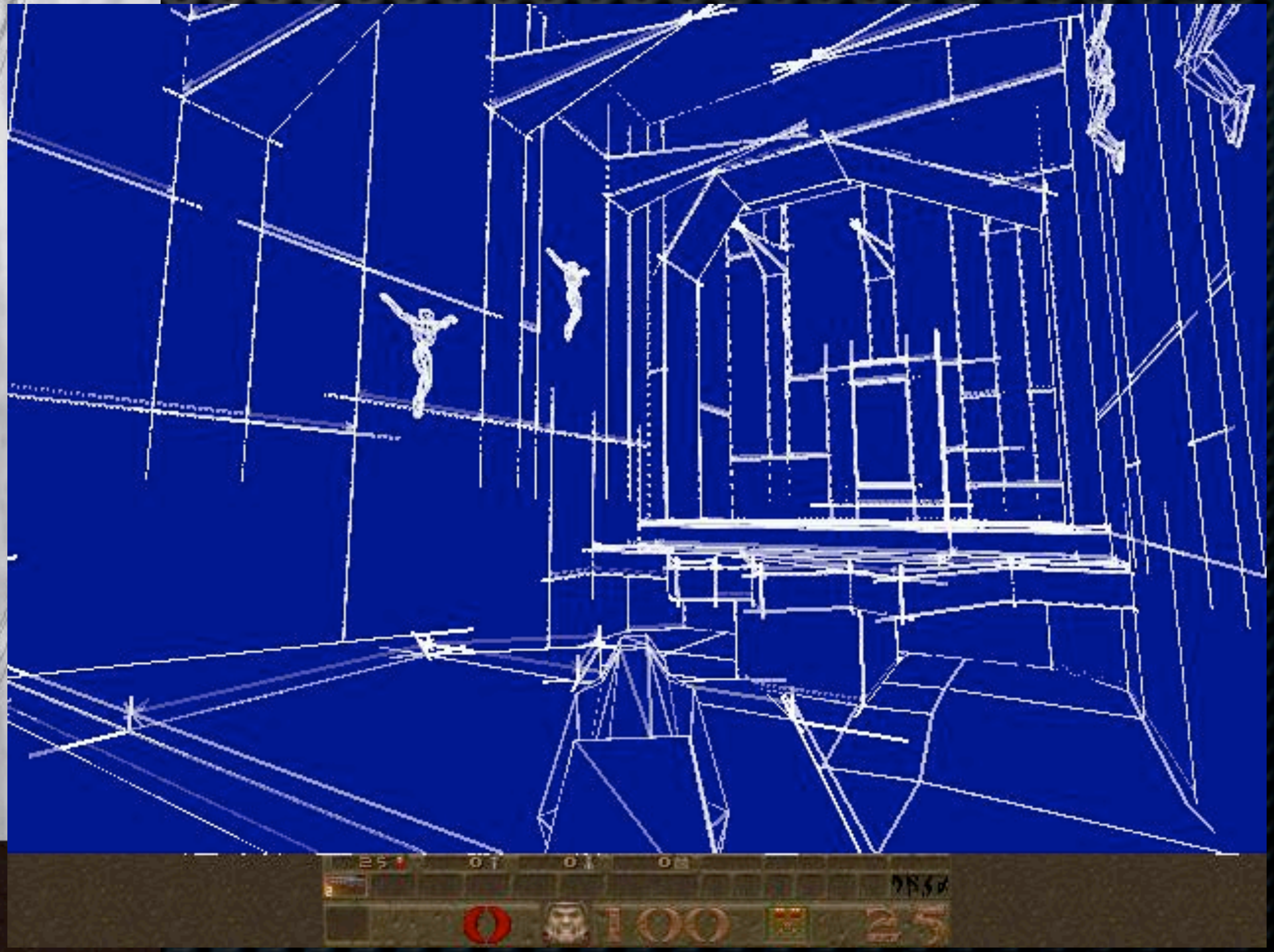
Rendering Overview

- Rendering - turning the numbers inside the computer into images on the screen
- Polygons - a set of three or more points in 3D space, defining a surface (a triangle usually)
- Using specialized matrix math, the computer “flattens” the polygons’ points onto the screen and fills space between them
  - This is rasterization, as opposed to ray tracing
- We can influence this process to create interesting effects
- For a long time, most research went into making the images as realistic as possible

Non-Photorealistic Rendering

- Now, Non-Photorealistic Rendering (NPR) is getting more attention
- NPR presents more and different information than photorealistic rendering
  - Artistic styles
  - Data representation
- A few game examples...

TechnicalQuake/NPRQuake



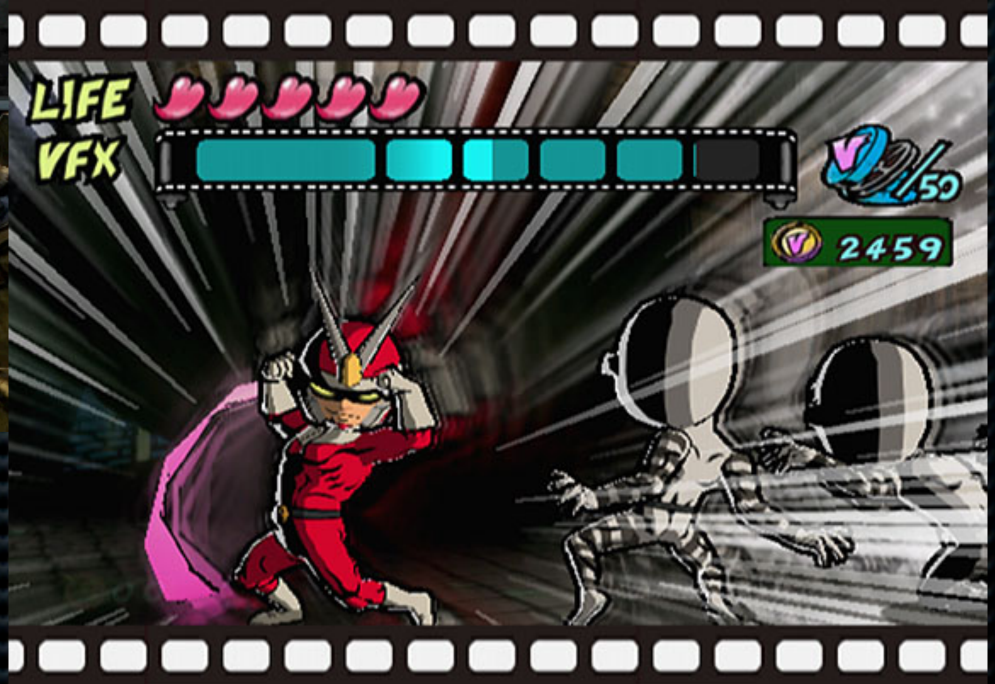
Legend of Zelda: Wind Waker

MadWorld

Okami

XIII

Others



Edge Uses

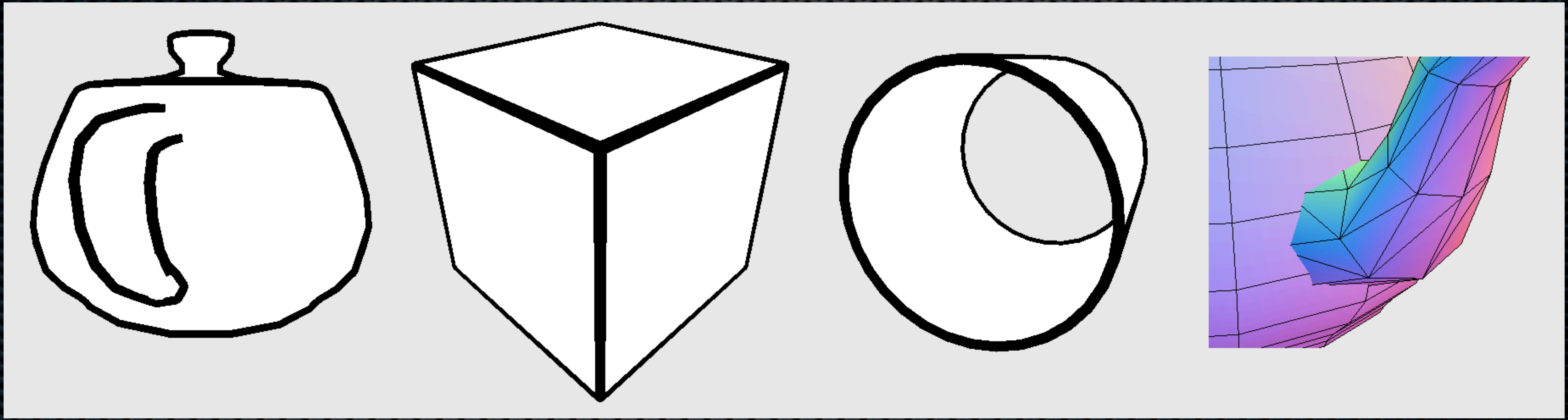
1. Differentiate multiple objects
2. Differentiate sections of objects
3. Highlighting individual objects
4. Enhance structural perception
   - Easier to figure out the object's 3D shape
5. Achieve specific graphical style
   - Watercolor, toon, etc.
6. Anti-Aliasing
   - Preventing "jaggies"

Edge Types

- Contour - Polygon edge separating a front-facing polygon from a back-facing one
- Crease - Polygon edges where the adjacent polygons' normals are greater than a user-defined angle from each other
- Boundary - Polygon edges connected to only one polygon
- Intersection - Non-polygonal edge collision of two polygons
- Marked - Polygon edges that are marked to always be drawn

- Drawable - Polygon edges that will be drawn on a given frame, as opposed to all polygon edges

Edge Types



Overview of Edge Detection Methods

- Hardware Methods
- Image-space Method
- Object-space Methods
- Miscellaneous Methods

Hardware Methods

- Renders edges as a bi-product of the order and method of rendering, not a specific detection step
- Pros
  - Very fast
  - Simple to implement
  - Supported on older devices
- Cons
  - Usually only renders contour edges
  - Lacks customizability

Hardware Method Example

1. Render front-facing polygons
2. Render back-facing polygons in wireframe mode with thickened edges

- Generates edges along the contours

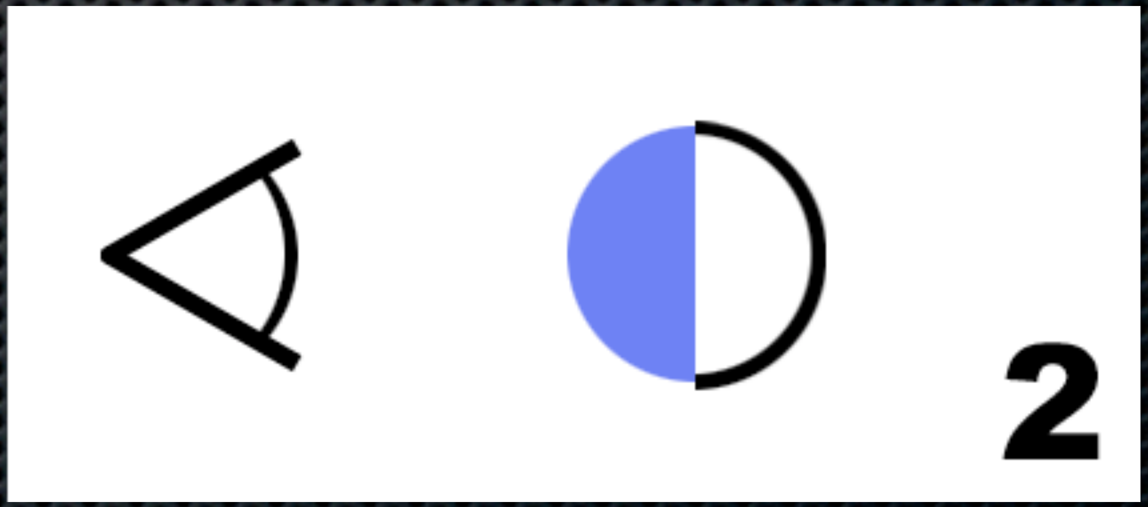
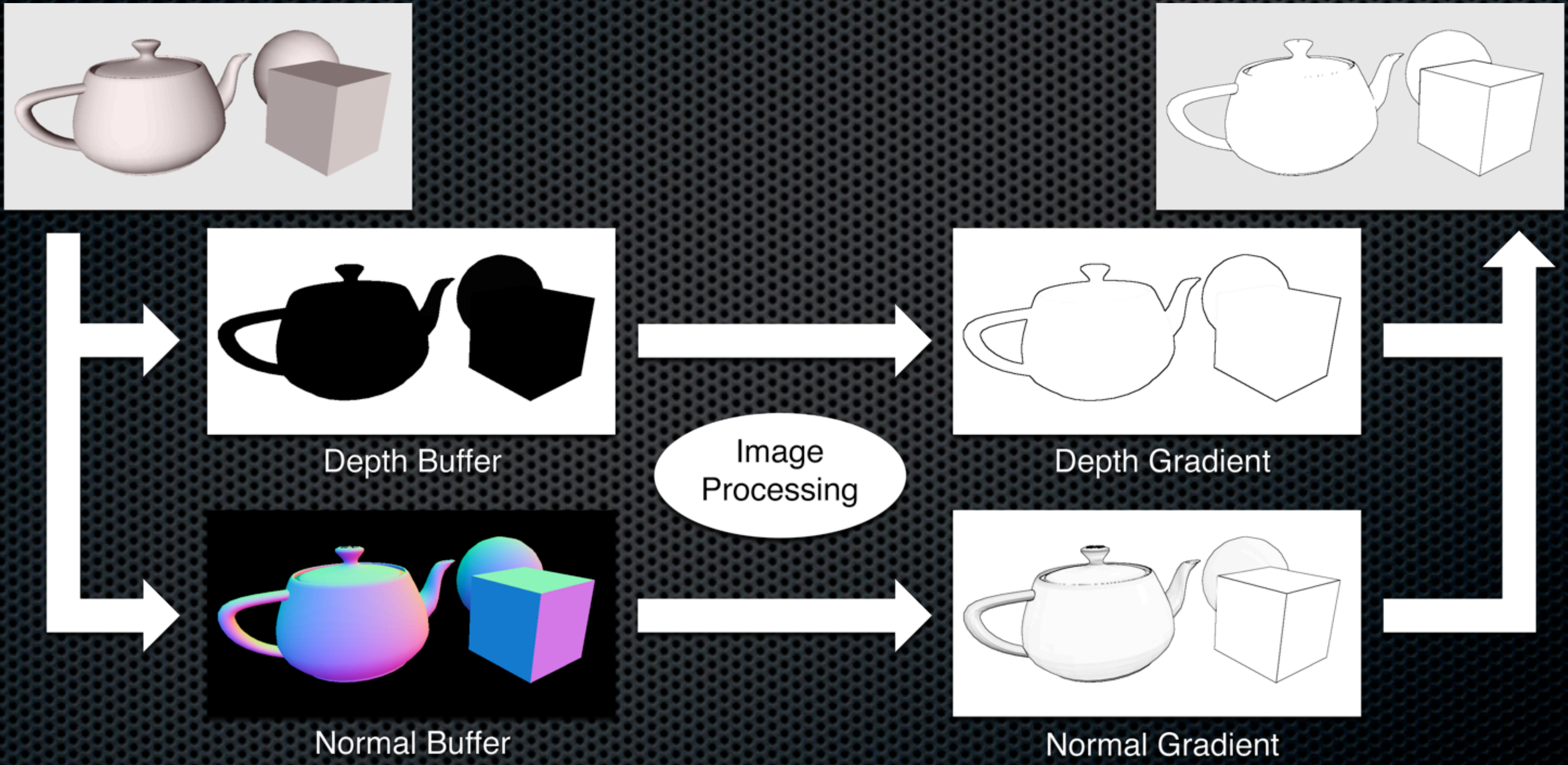


Image-space Method

- Uses image filters to detect areas of rapid change (edges) in data representations of the scene
- Pros
  - Naturally detects intersection edges
  - Constant speed regardless of scene complexity
- Cons
  - Edge thickness unpredictable
  - Lacks customizability

Image-space Method Example



Object-space Methods

- Detects edges in 3D space by checking individual polygon edges
- Pros
  - Very accurate
  - Easily controlled and customized
- Cons
  - Relatively slow
  - Usually requires preprocessing

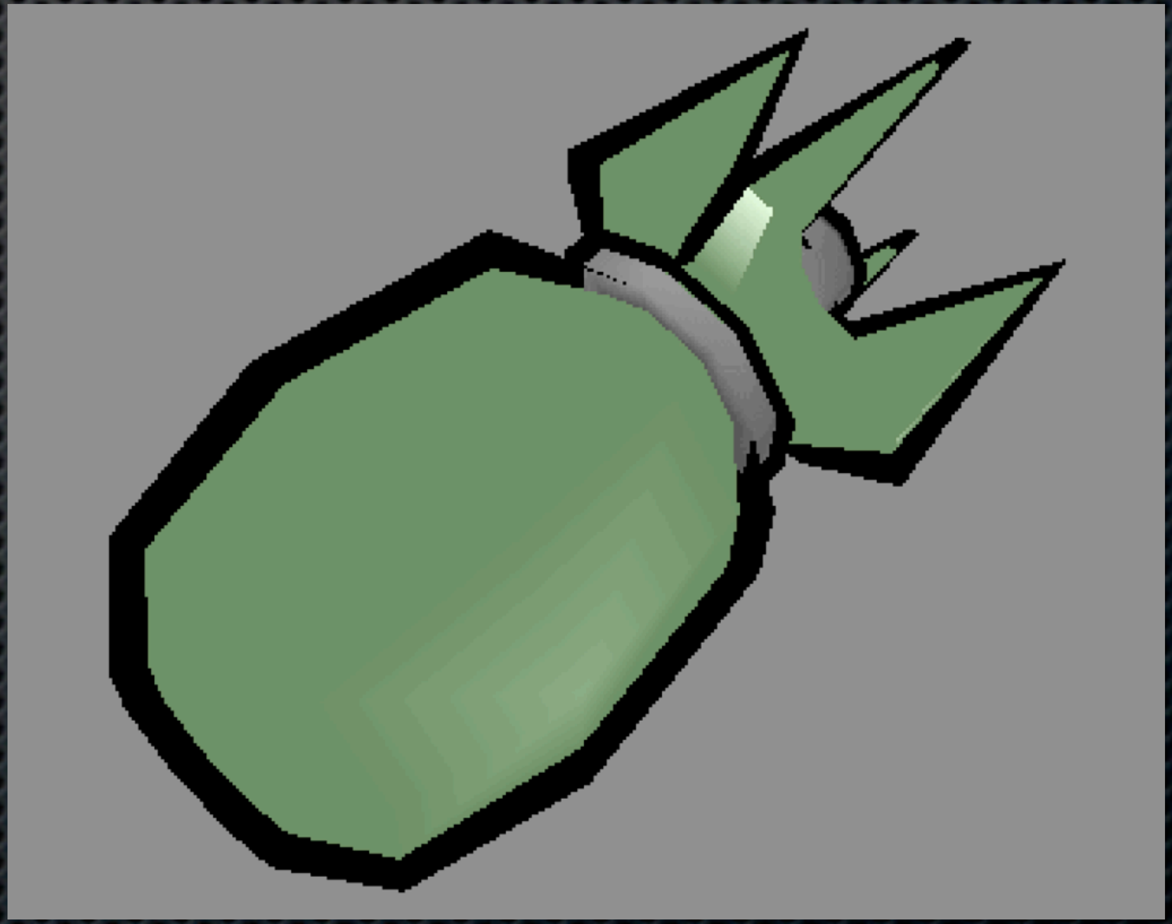
Object-space Method Detail

- Every frame, check all unique polygon edges for drawability
- Drawable edges are given their own polygons and rendered like any other object
- Customization
  - Edge thickness
  - Edge color
  - Edge geometry
  - Textured edges
- Dozens of variations on the basic premise exist, mostly dealing with optimizations

Object-space Method Examples

Miscellaneous Methods

- Render black where the normal is nearly perpendicular to the view
- Render black a scaled copy of the mesh with inverted normals and back-face culling turned on
- etc.

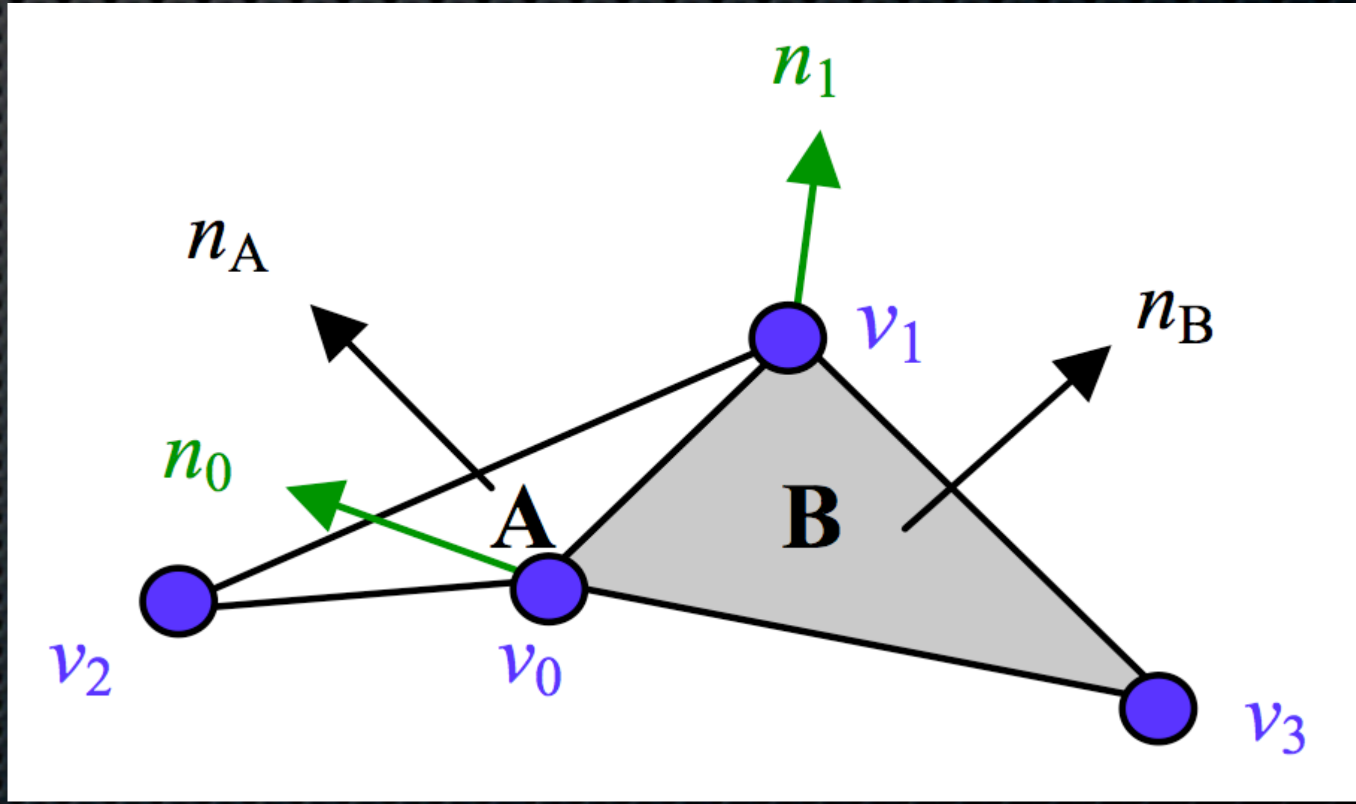


Research Inspiration

- While studying NPR techniques for my thesis, I discovered a 2004 paper by Morgan McGuire and John F. Hughes
- Object-space edge detection on the graphics card (GPU) via shaders
  - 30 times slower than rendering the object normally
  - Uses 9 times more data per object
  - But, <u>15 to 30 times faster</u> than doing the same thing on the CPU

McGuire and Hughes' Method Detail

- Preprocessing (before rendering the first time)
  1. Find all unique polygon edges in an object
  2. Obtain the edge data: v0, v1, v2, v3, n0, n1, and i
  3. Duplicate the edge data 3 times (4 total)
  4. Make sure i is unique (0, 1, 2, 3)
  5. Store data on GPU

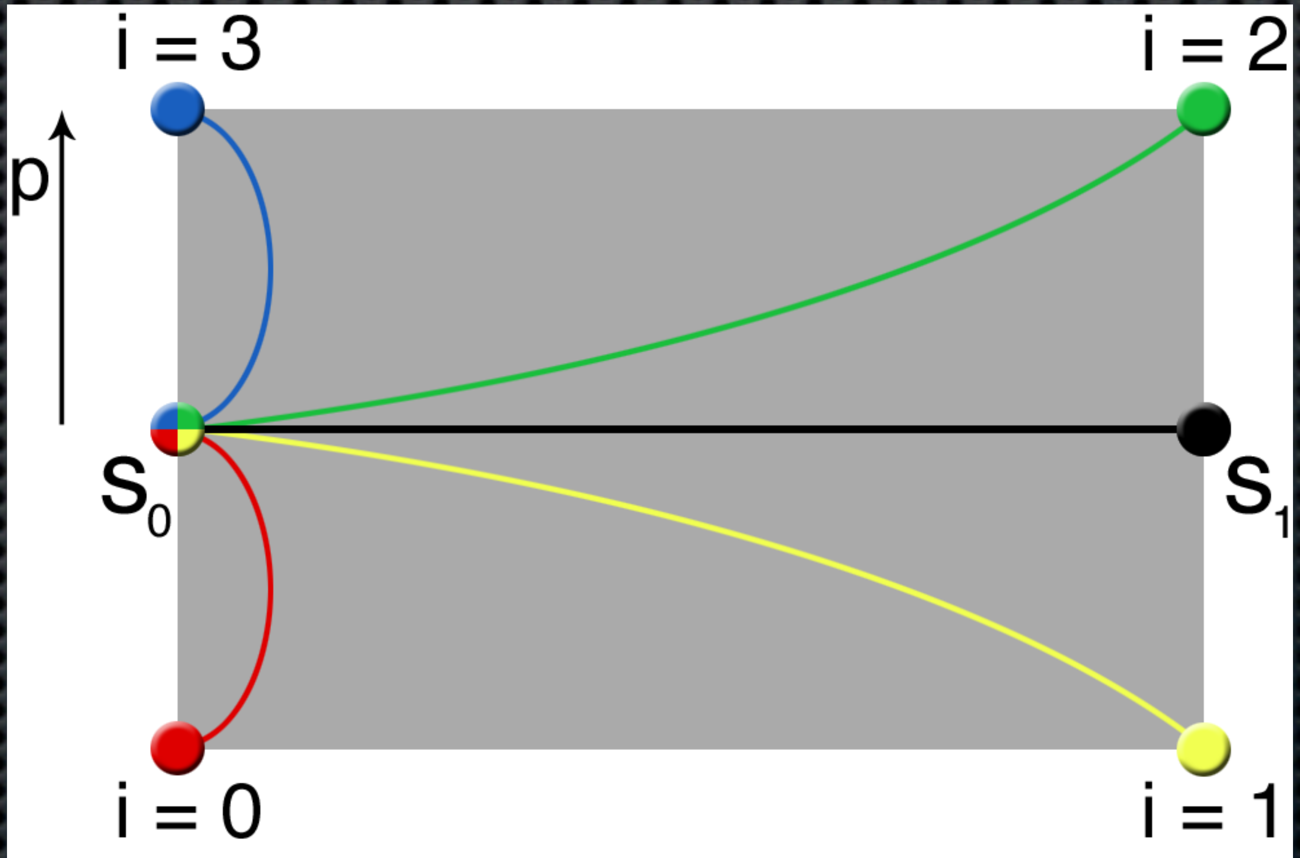


McGuire and Hughes’ Method Detail

- Render time (each frame)
  - Determine if the edge is drawable
    - Contour: [dot(nA, (eye - v0)) * dot(nB, (eye - v0)) < 0]
    - Crease: [dot(nA, nB) < -cos(θ)]
    - Marked/Boundary: [v3 == v0]
    - (nA and nB are the face normals of the two polygons)
    - (θ is a user-defined angle)
    - (eye is the position of the camera)

McGuire and Hughes' Method Detail

- Render time (each frame)
  - For drawable edges, depending on the i value, output one of four points that make up a screen-aligned edge quad
  - Non-drawable edges output degenerate quad points

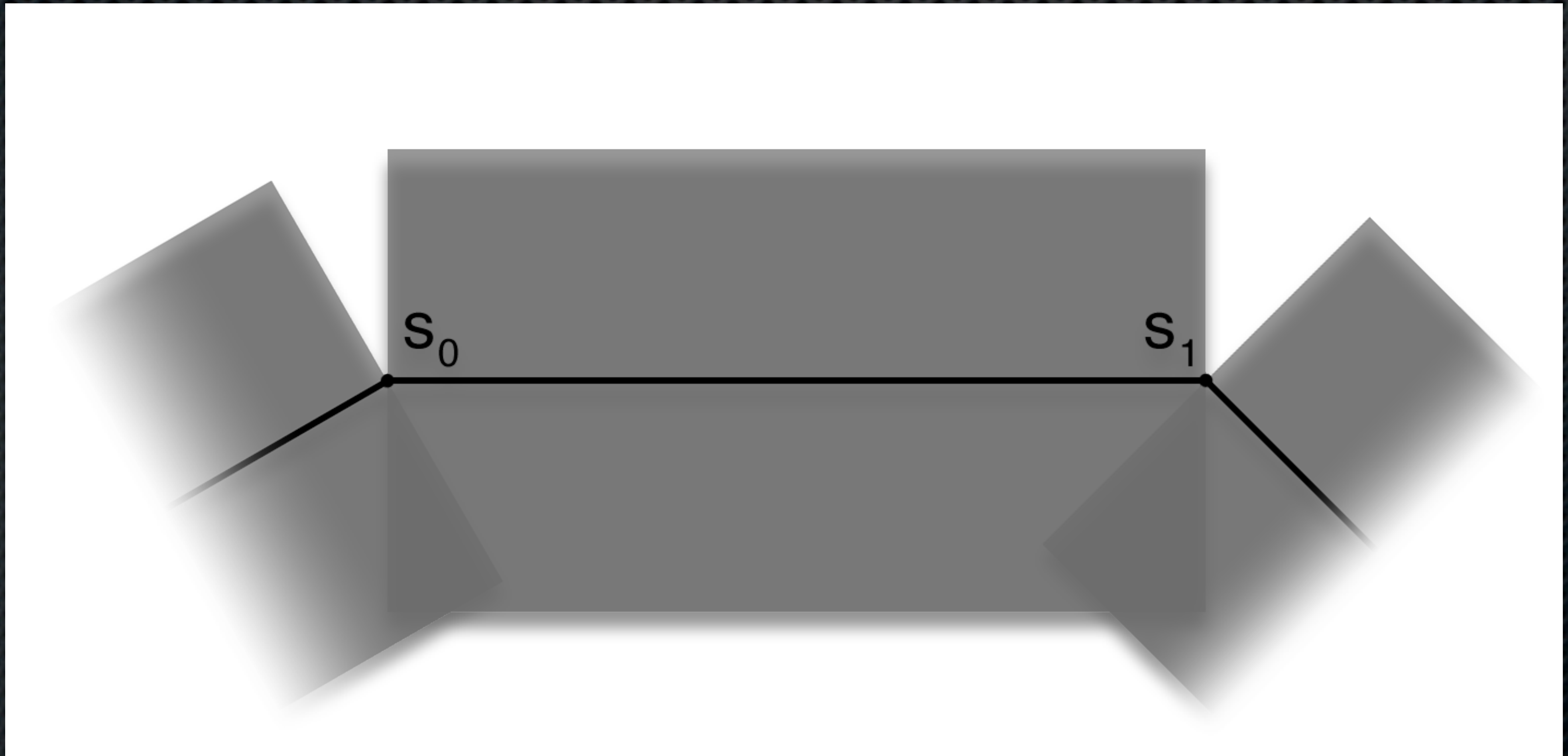


McGuire and Hughes' Method Variations

- Half-quads for contours
  - Prevents some artifacts, but complicates the implementation
- Rasterized lines (thin lines) instead of quads
  - Faster and cheaper, but no customization

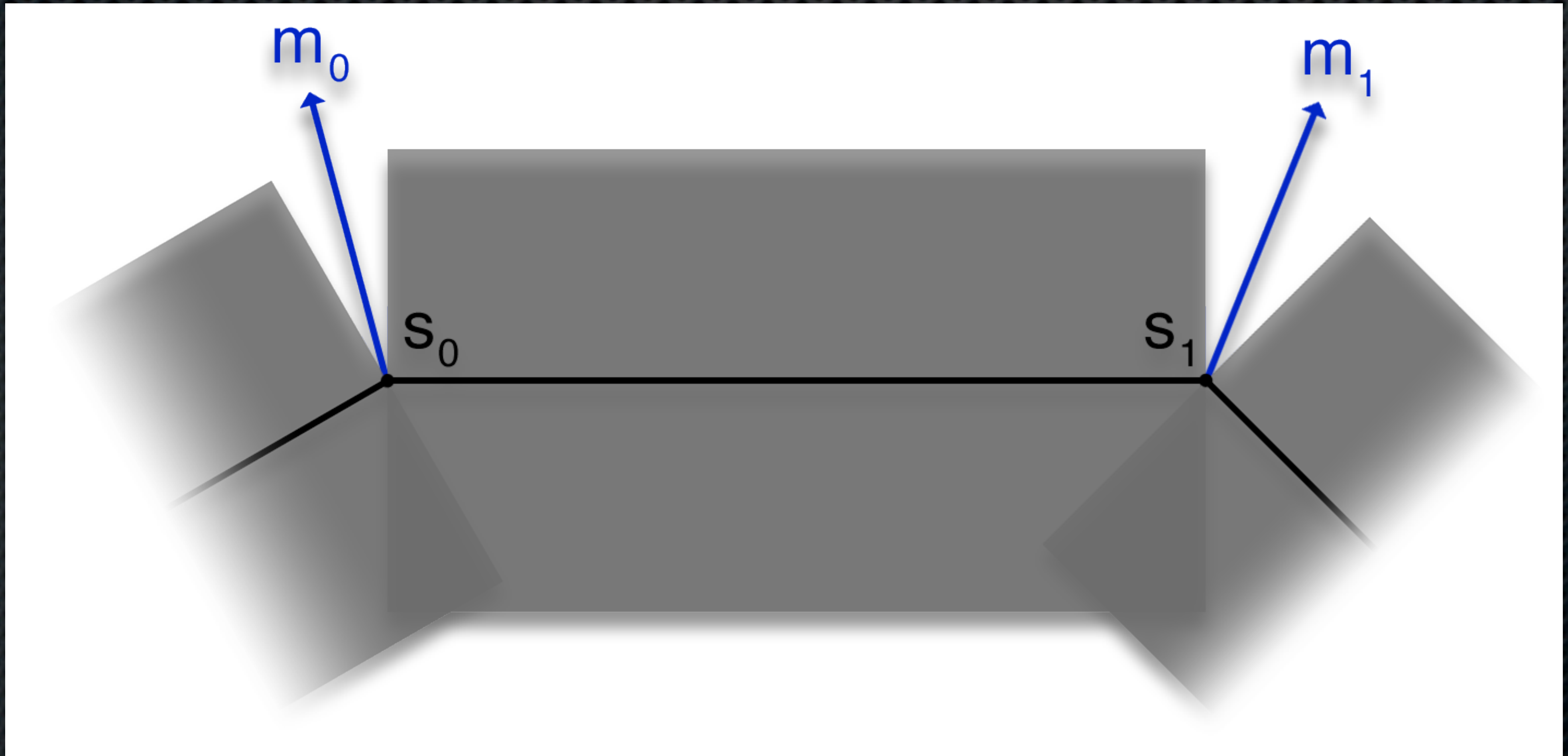
McGuire and Hughes' Method Capping

- Gaps are formed between thick edges
- Fill them with two caps, each covering half of the gap



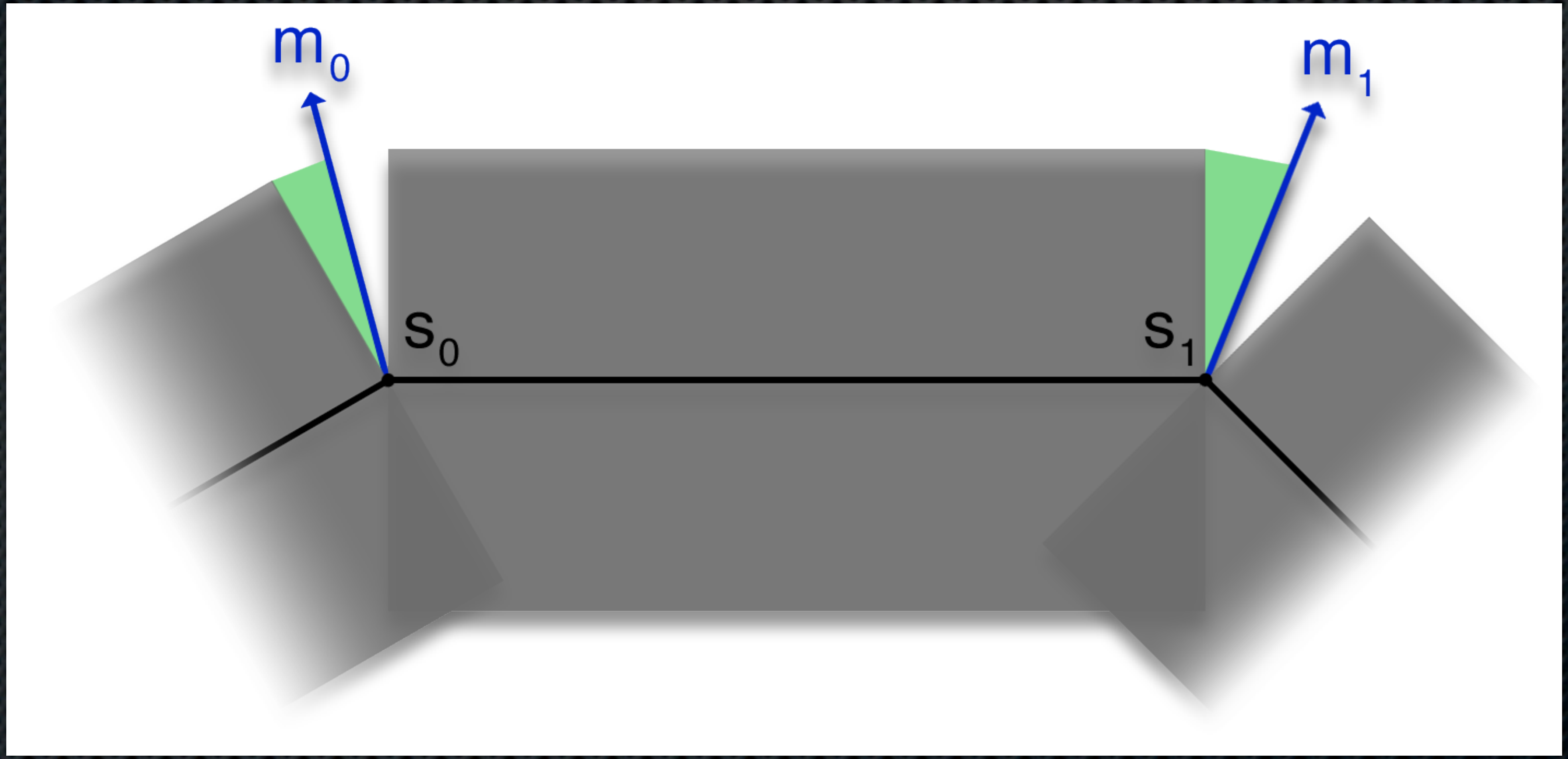
McGuire and Hughes' Method Capping

- Gaps are formed between thick edges
- Fill them with two caps, each covering half of the gap



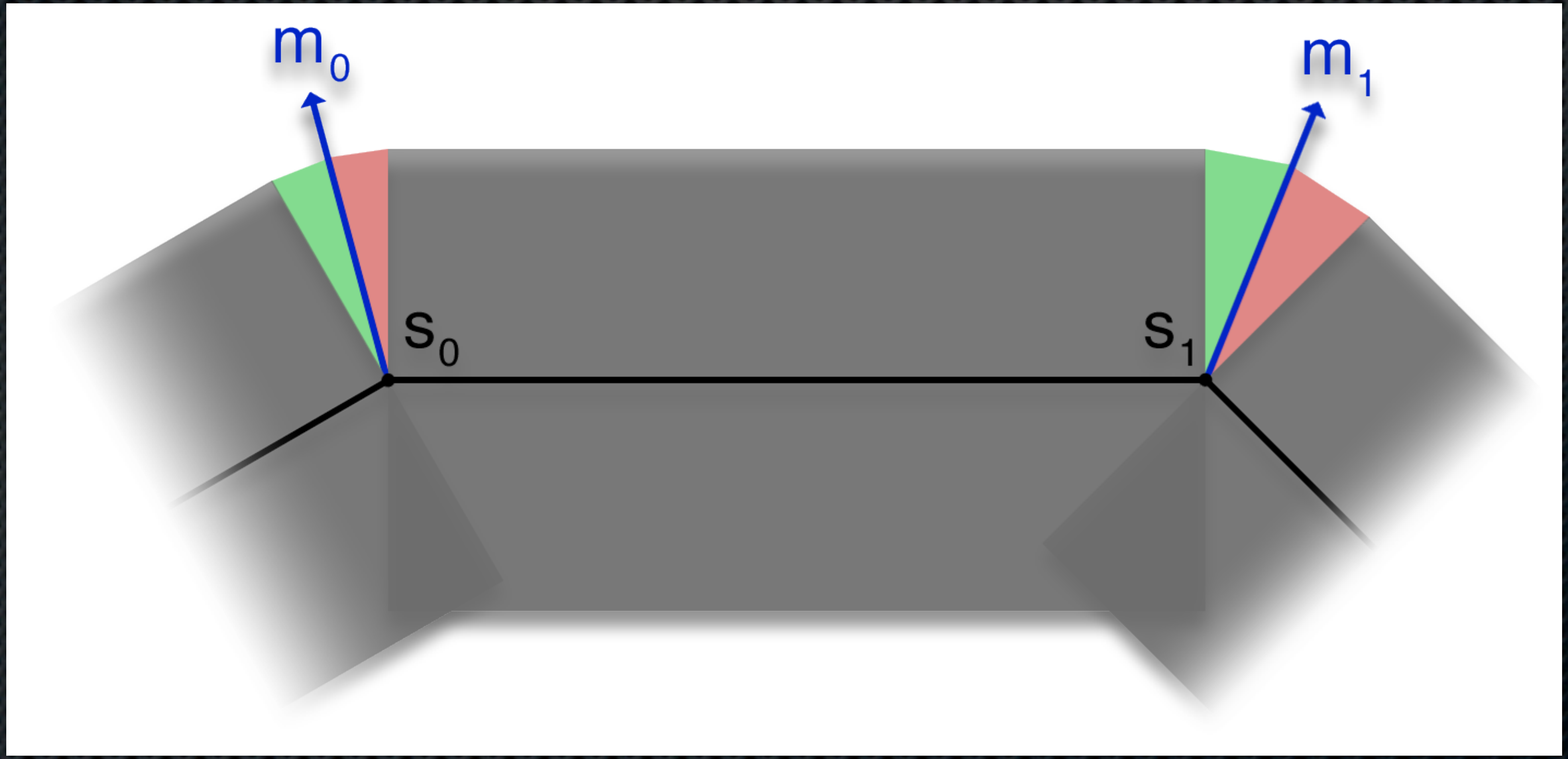
McGuire and Hughes' Method Capping

- Gaps are formed between thick edges
- Fill them with two caps, each covering half of the gap



McGuire and Hughes' Method Capping

- Gaps are formed between thick edges
- Fill them with two caps, each covering half of the gap



McGuire and Hughes' Method Steps

- The whole process involves four rendering passes per frame
  1. Render mesh with depth offset
  2. Render thick edges
  3. Render edge caps for left vertex
  4. Render edge caps for right vertex

Issues

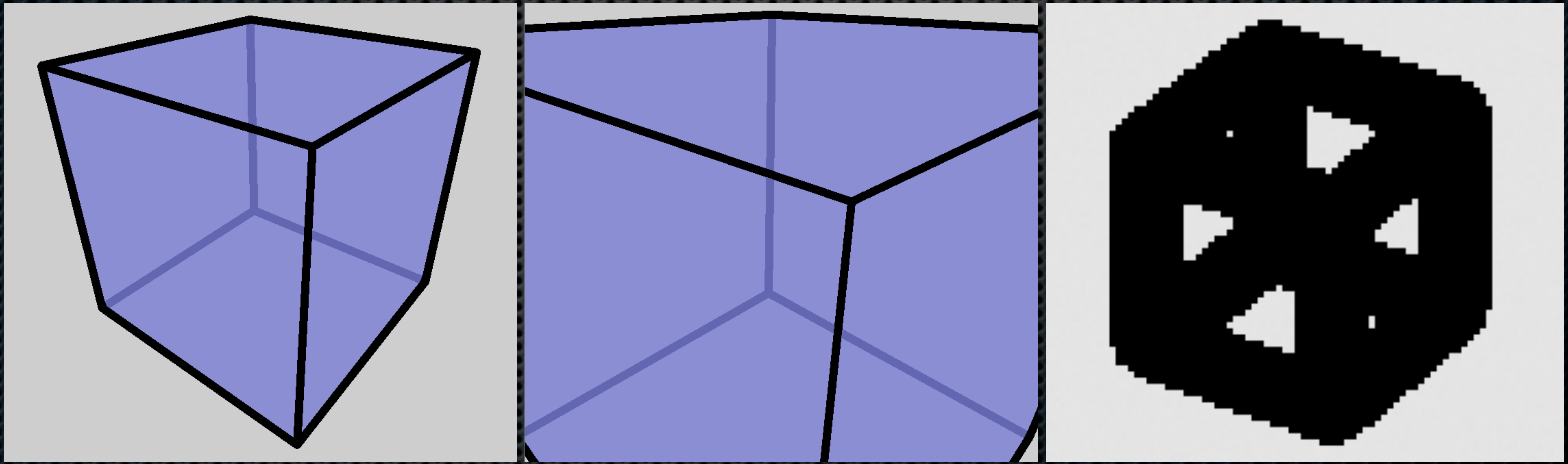
- Screen-space thickened edges can overpower the mesh
- Normals may not represent curvature of the surface creating bad caps (the “bad normal” problem)
- High memory usage and computation duplication
  - McGuire and Hughes suggested the use of geometry shaders and data textures

Research

- Thesis Goal: alleviate the issues and explore alternative solutions
- Aspects researched were:
  - Depth-based edge thickness
  - “Bad normal” solutions
  - Reduce render passes with alternate edge types
  - Attempt to use OpenCL to make the whole process better
    - Fewer computations
    - More accurate caps

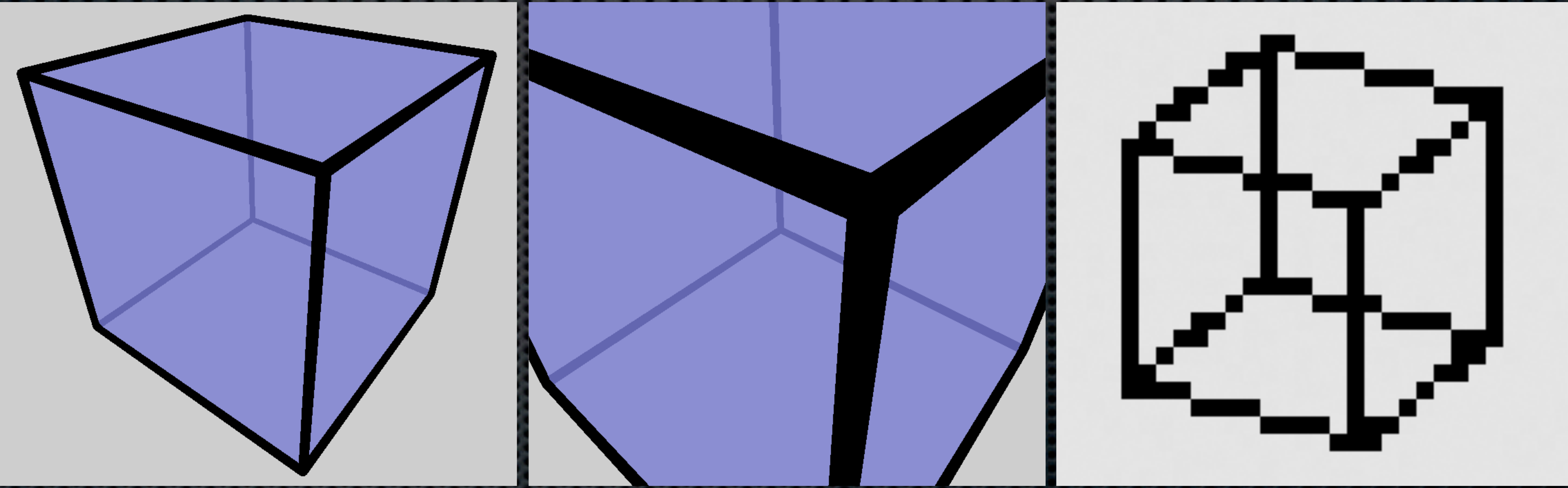
Screen-space Edge Thickness Issue

- Hurts depth perception
- Edges can overpower distant objects



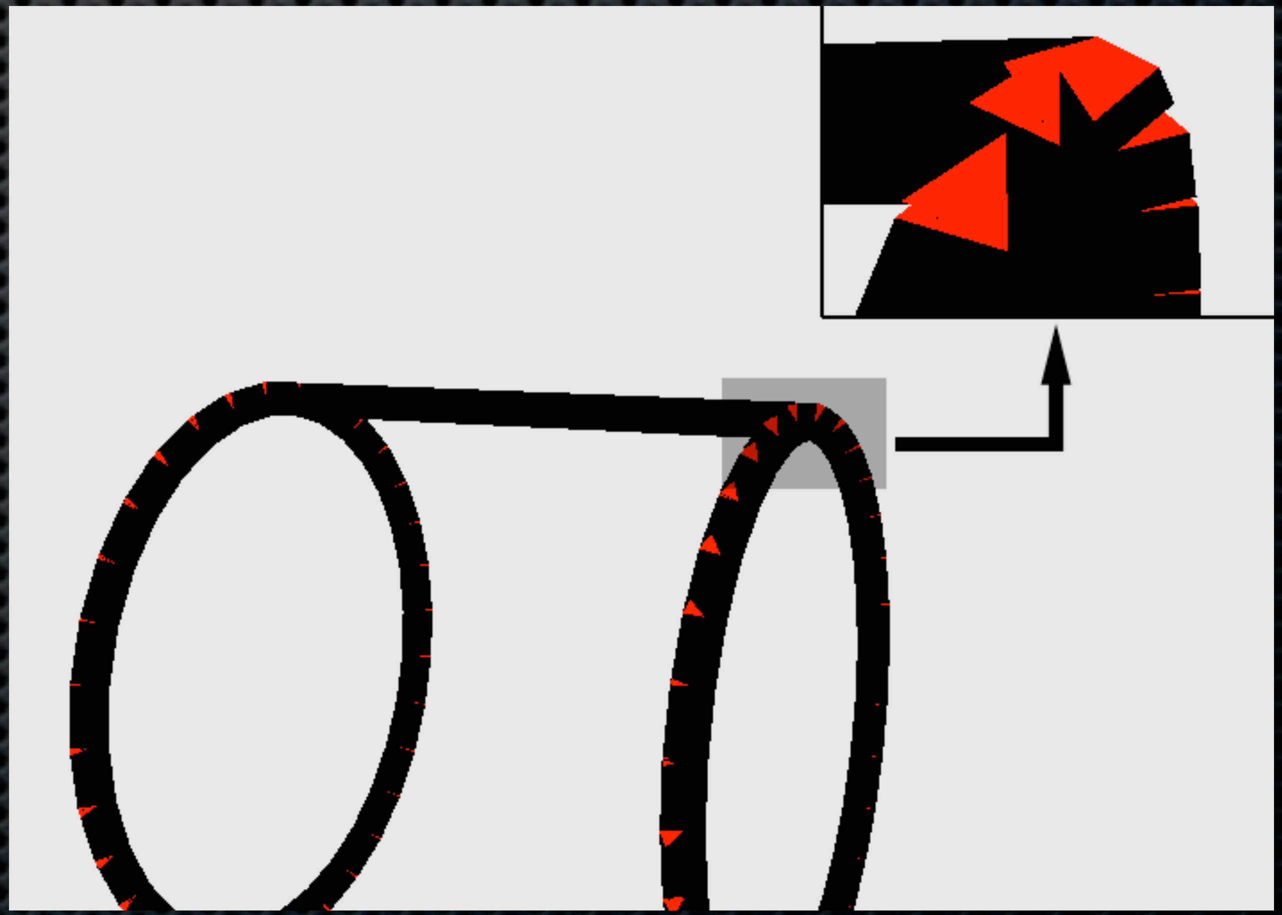
Depth-based Edge Thickness

- Depth perception is maintained
- Edges don't overpower distant objects
- Adding a minimum prevents loss of distant edges



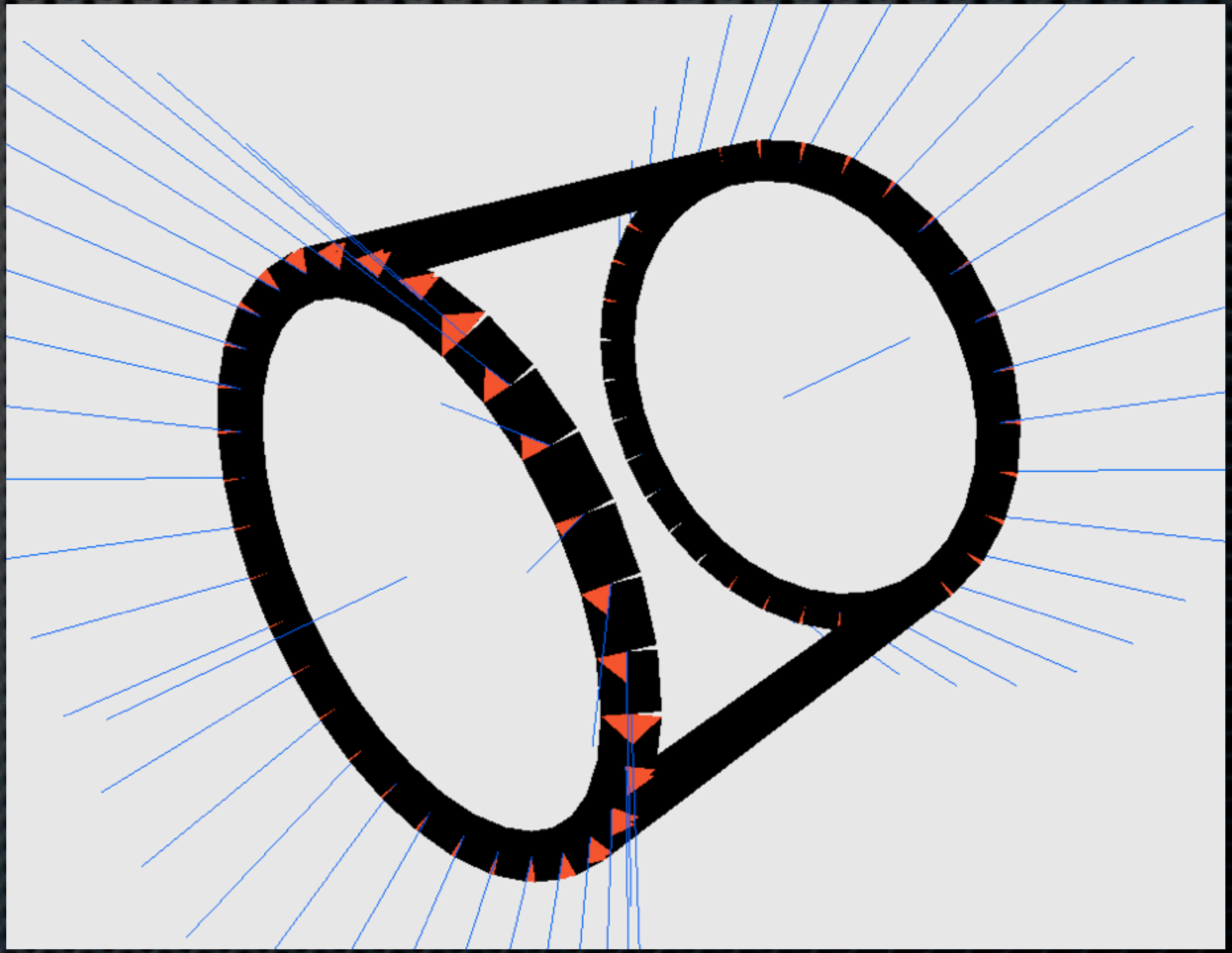
"Bad Normal" Problem

- Case I: three or more drawable edges converge
  - More than one normal is correct
  - Edge redundancy usually prevents full failure



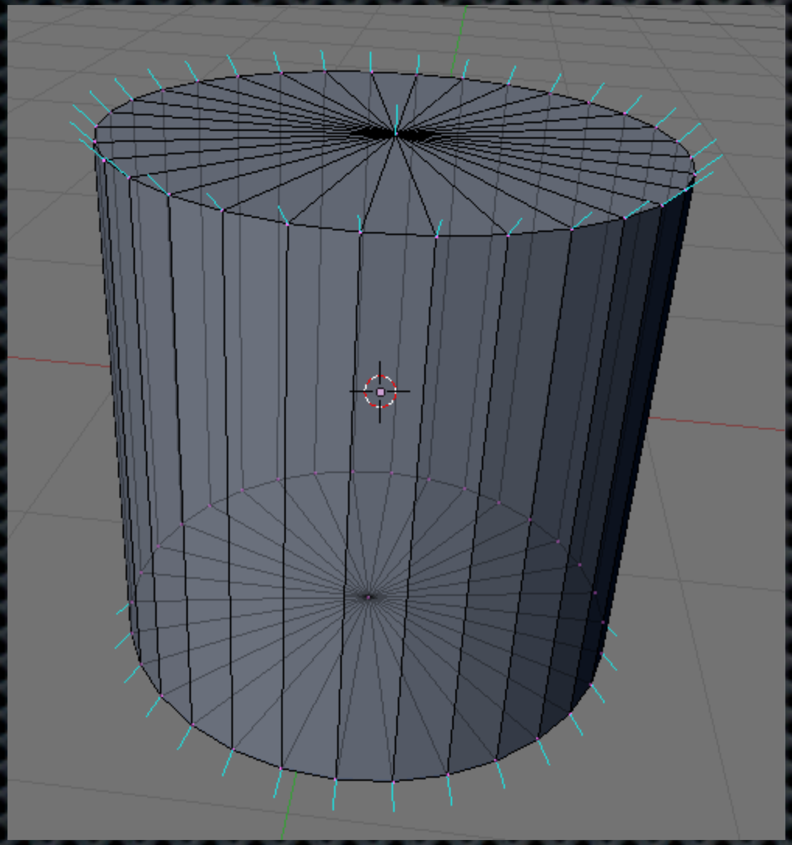
“Bad Normal” Problem

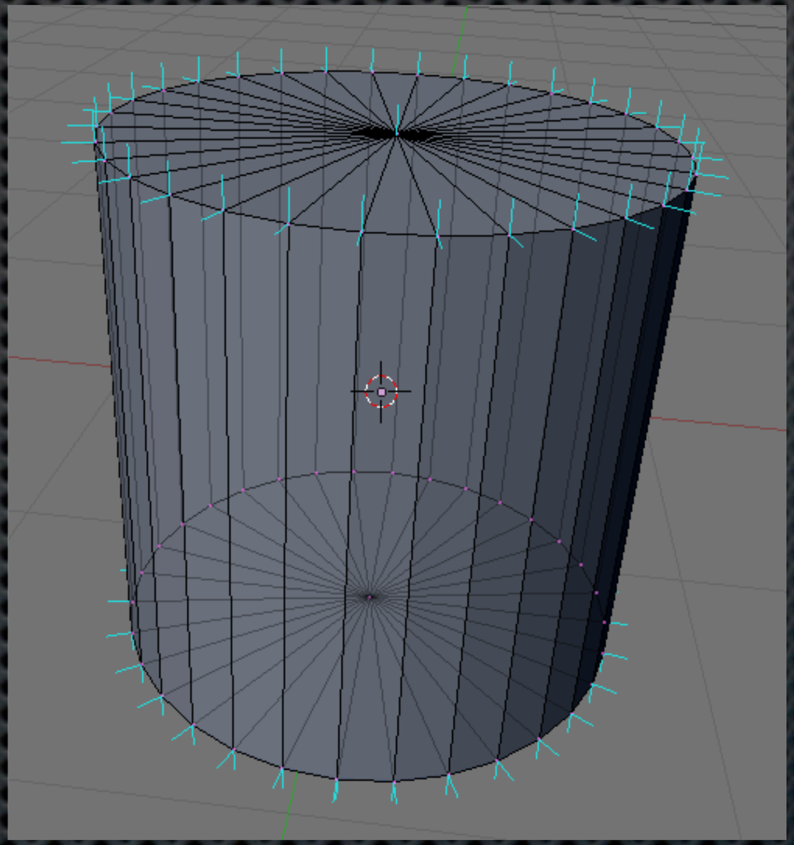
- Case II: curved area abuts a flat area
  - Only one normal is correct
  - It is difficult to find

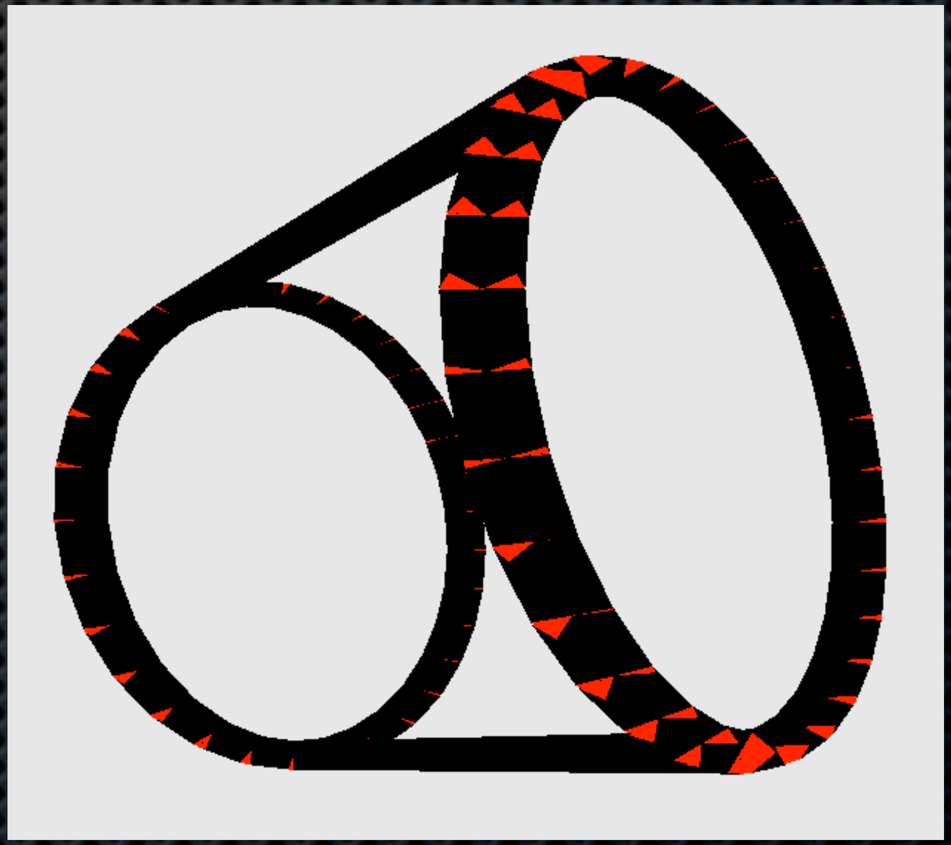


Solving the “Bad Normal” Problem 1

- Allow for duplicate edges
- Export models with edges already split over the user-defined angle

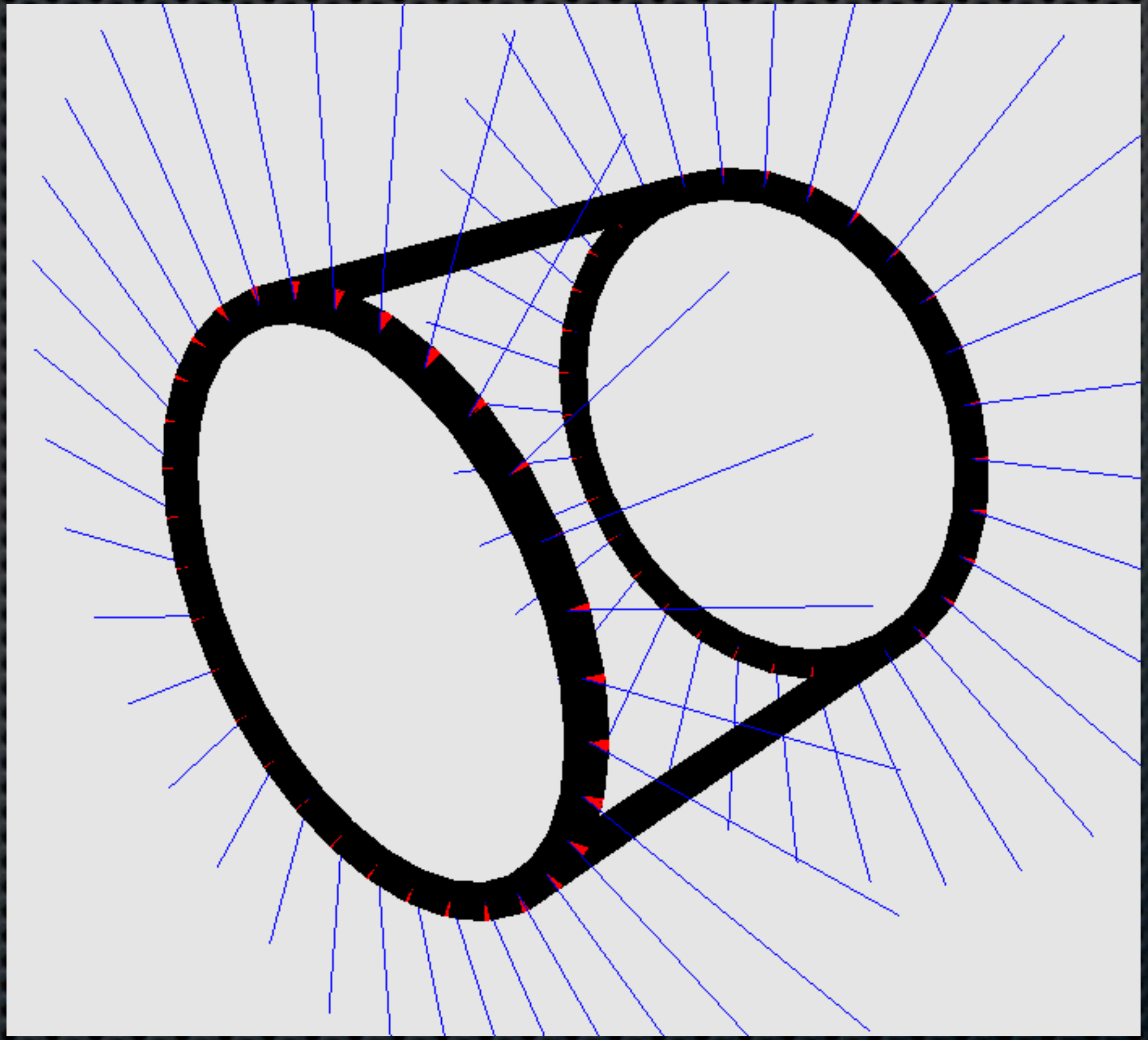






Solving the “Bad Normal” Problem 2

- Pick better normals for the edges during the mesh creation process
- This is a manual solution



Solving the “Bad Normal” Problem 3

- Use an alternate form of capping not based on normals
- More later...

Alternate Edge Types

- Concept: Combine caps into the edge rendering to reduce passes
  - Half Hex Method
  - House Method
  - Plug Method
  - Double caps to handle bad normals

Demo

OpenCL

- Version 1.0 of OpenCL (Open Compute Library) was released in 2009
- Allows massively parallel computation on GPUs and other devices
- Interoperable with OpenGL

OpenCL Edge Detection

- OpenCL's abilities allow another method of edge detection similar to McGuire and Hughes'
  - Reduces data and calculation duplication
  - Higher accuracy caps

How It Works: Edges

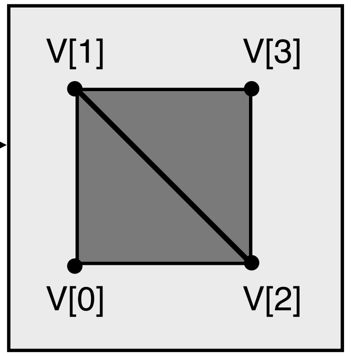
- Vertex data is already on the GPU
- Only need to store connectivity information
  - A vertex list on the GPU representing edges, not polygons
- Edge detection and edge vertex generation are both nearly identical to McGuire and Hughes' method
  - The output vertices must be temporarily stored, OpenCL cannot render to the screen

How It Works: Caps

- The temporarily stored drawable edges define what edges need caps
- Like the edges, keep a buffer of connectivity information that defines the possible caps
- Since the positions of the drawable edges are known, no normals are necessary, removing the “bad normal” problem

Data Representation

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]



Trivial set of connected polygons in 2D space

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]

Output

Offset Reference

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

Output

Cap Out Buffer	
Data	**ID**
[VERTICES]	O2[0]
[VERTICES]	O2[1]
[VERTICES]	O2[2]
[VERTICES]	O2[3]
[VERTICES]	O2[4]
[VERTICES]	O2[5]
[VERTICES]	O2[6]
[VERTICES]	O2[7]

OpenCL Method Steps

- Compute Passes
  1. Create Edges
  2. Create Caps
- Render Passes
  3. Draw Object
  4. Draw Edges (simple)
  5. Draw Caps (simple)

Problems

- OpenCL implementation was slow
- Several potential causes were found
  - Using non-vector memory loads/stores
  - Memory access speeds are not equivalent
  - GPU operations occur in lock-step
  - Output must be stored temporarily before rendering
  - ...and possibly several others

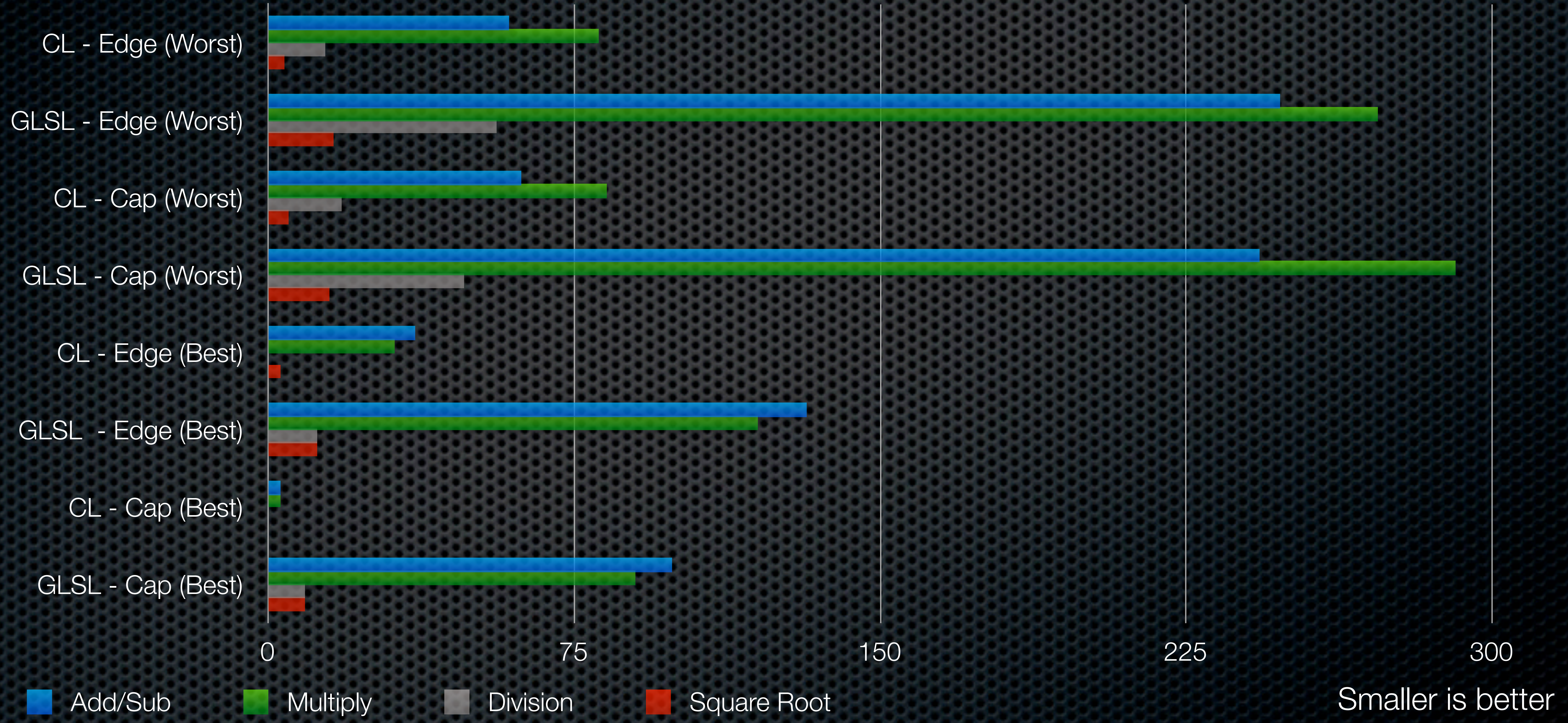
Addressing Issues

- Optimized kernel with vector memory operations
- Implemented both methods on the CPU, where memory speeds are equivalent and short-circuiting is possible
- Implemented McGuire and Hughes' capping method in OpenCL
- Experimented with other methods as well

Results Analysis

- How many operations are performed?
- How much memory is used?
- How much geometry is drawn?
- How fast are they?

Arithmetic Operations



Logical/Other Operations

Processing Ratios

	Edge CL:GLSL Ratio Worst Case	Cap CL:GLSL Ratio Worst Case	Edge CL:GLSL Ratio Best Case	Cap CL:GLSL Ratio Best Case
Add/Sub	0.238	0.255	0.272	0.030
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

Smaller is better

Note that though specific values will change from implementation to implementation, the ratios remain approximately the same

GPU Memory Usage

	Memory Usage Per Item
CL Edge	640
GLSL Edge	2688
CL Cap	576
GLSL Cap	96

Values are in bits
Assumes 32 bit floats/integers
GLSL caps are small due to reuse of data in GLSL edges
GLSL quantities do not include the transparent memory used within the pipeline
Smaller is better

Edge and Cap Quantities

	Edges (CL/GLSL)	Caps (CL)	Caps (GLSL)	Cap:Edge Ratio (CL)	Cap:Edge Ratio (GLSL)
Normal Cube	24	24	48	1	2
Simple Cube	12	24	24	2	2
Cylinder	96	320	192	3.33	2
Merged Cylinder	96	192	192	2	2
Cone	64	592	128	9.25	2
Quad Sphere	2016	6944	4032	3.44	2
Ico Sphere	1920	9570	3840	4.98	2
Teapot	1180	4420	2360	3.75	2
Monkey	1449	7188	2898	4.96	2
Bunny	20812	107290	41624	5.16	2

Edge and Cap Memory Usage

	Total CL Memory	Total GLSL Memory	Memory Ratio (CL : GLSL)
Cube	29184	69120	0.422
Merged Cube	21504	34560	0.622
Cylinder	245760	276480	0.888
Merged Cylinder	172032	276480	0.622
Cone	381952	184320	2.07
Quad Sphere	5289984	5806080	0.911
Ico Sphere	6741120	5529600	1.21
Teapot	3301120	3398400	0.971
Monkey	5067648	4173120	1.21
Bunny	75118720	59938560	1.25

McGuire and Hughes’ caps in OpenCL

	Total CL Memory	Total GLSL Memory	Memory Ratio (CL : GLSL)
Cube	39936	69120	0.577
Merged Cube	19968	34560	0.577
Cylinder	159744	276480	0.577
Merged Cylinder	159744	276480	0.577
Cone	106496	184320	0.577
Quad Sphere	3354624	5806080	0.577
Ico Sphere	3194880	5529600	0.577
Teapot	1963520	3398400	0.577
Monkey	2411136	4173120	0.577
Bunny	34631168	59938560	0.577

McGuire and Hughes’ caps, implemented in OpenCL, save a lot of memory over the shader version
Values in bits, assumes 32 bit floats/integers
Smaller is better

Drawable Geometry

	Edges (CL & GLSL)	Caps (CL)	Caps (GLSL)	Cap:Edge Ratio (CL)	Cap:Edge Ratio (GLSL)
Cube	24	24	48	1	2
Merged Cube	12	24	24	2	2
Cylinder	130	136	260	1.04	2
Merged 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

- The number of drawable edges and caps is usually view dependent
    - These numbers assume the camera is pointing at the origin while positioned at (3, 3, 3)
    - The models are at the origin, and generally are of dimension 1
- Smaller is better

Speed: CPU

	Framerate (CL)	Framerate (GLSL)	Ratio (CL:GLSL)
Cube	1135	1156	0.982
Merged Cube	1155	1180	0.979
Cylinder	1139	1182	0.964
Merged Cylinder	1126	1128	0.998
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

Bigger is better

Speed: GPU

	Framerate (CL)	Framerate (GLSL)	Ratio (CL:GLSL)
Cube	760	1173	0.647
Merged Cube	768	1184	0.648
Cylinder	733	1103	0.664
Merged Cylinder	741	1142	0.648
Cone	724	1244	0.581
Quad Sphere	416	713	0.583
Ico Sphere	360	739	0.487
Teapot	464	850	0.545
Monkey	334	770	0.433
Bunny	31	134	0.231

Bigger is better

Speed: Original Caps in OpenCL

	Framerate (CL)	Framerate (GLSL)	Ratio (CL:GLSL)
Cube	762	1173	0.649
Merged Cube	770	1184	0.65
Cylinder	735	1103	0.666
Merged Cylinder	749	1142	0.655
Cone	670	1244	0.538
Quad Sphere	584	713	0.819
Ico Sphere	593	739	0.802
Teapot	631	850	0.742
Monkey	634	770	0.823
Bunny	159	134	1.18

Bigger is better

Conclusion

- OpenCL has some potential for complex objects in terms of speed, but it’s not quite there yet
- Higher accuracy caps are the only real advantage to the OpenCL implementation at this time
- The other concepts (depth-based thickening, methods of reducing bad normals, and alternate edge methods) work now in shaders

Demo

Future

- Implementing McGuire and Hughes' method with the duplicate data would allow for memory caching and probably faster speeds
- OpenCL capping method could be implemented with a geometry shader/data texture setup
- Microsoft's DirectCompute can render to the screen, unlike OpenCL
- Chris Peters suggested another capping method that could lead to equal quality caps without checking every possible combination of edges
- OpenCL's implementation will inevitably become faster over time

Questions?