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

Thesis Defense

By Dwight House

May 21st, 2010

Advisor: Dr. Xin Li

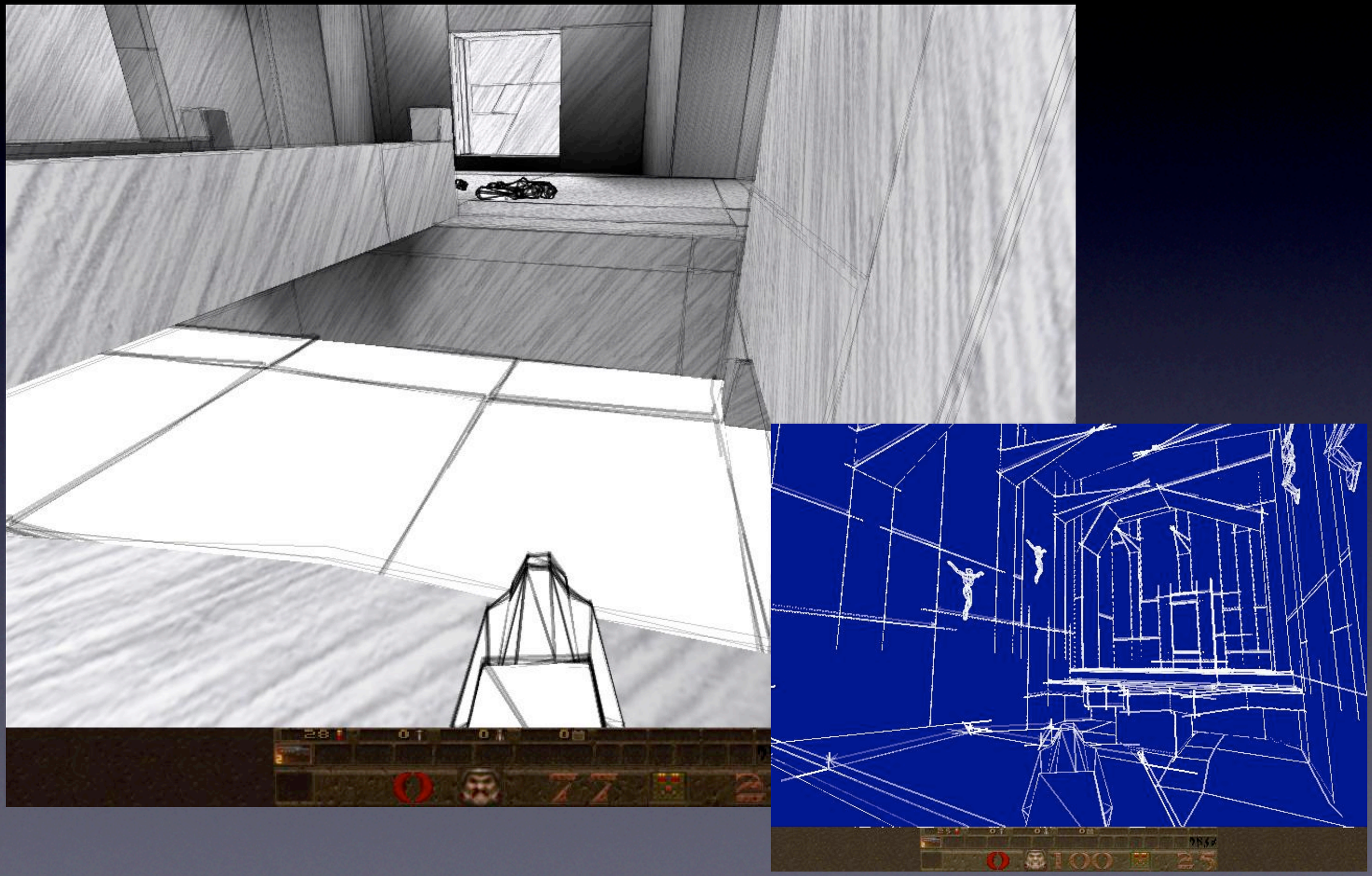
Part I

- Non-Photorealistic Rendering
- Definitions
- Survey Overview

Non-Photorealistic Rendering

- Non-Photorealistic Rendering (NPR) presents more and different information, compared to photorealistic rendering
  - Artistic styles
  - Data representation
- A few game examples...

TechnicalQuake/NPRQuake



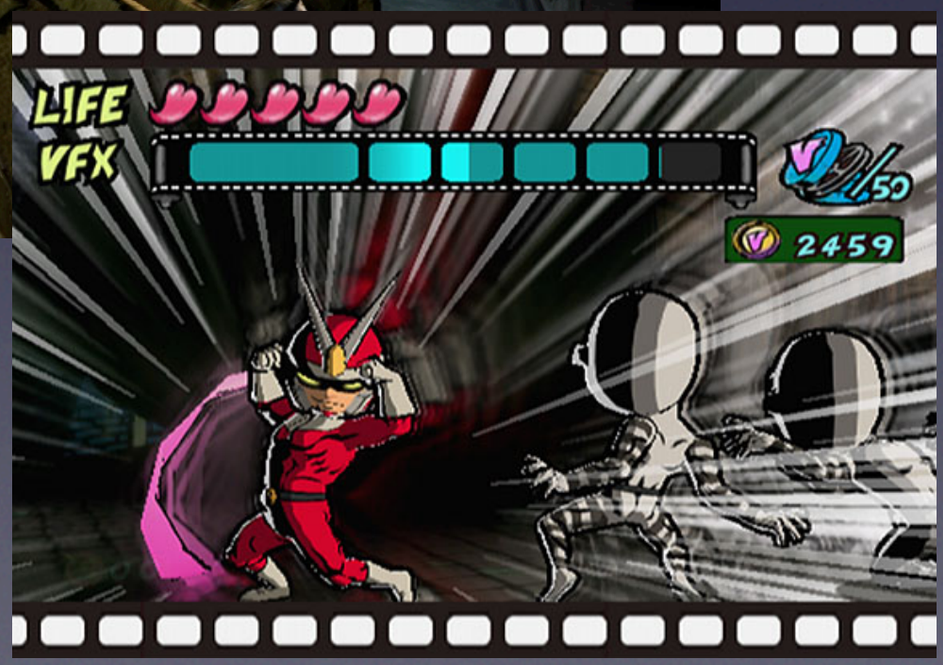
Legend of Zelda: Wind Waker

MadWorld

Okami

XIII

Others



Edges

- Edges are one of the most important structures in NPR
- They have many uses...

Applications

- Differentiate multiple objects
- Differentiate sections of objects
- Highlighting individual objects
- Enhance structural perception
- Achieve specific graphical style
- Anti-Aliasing

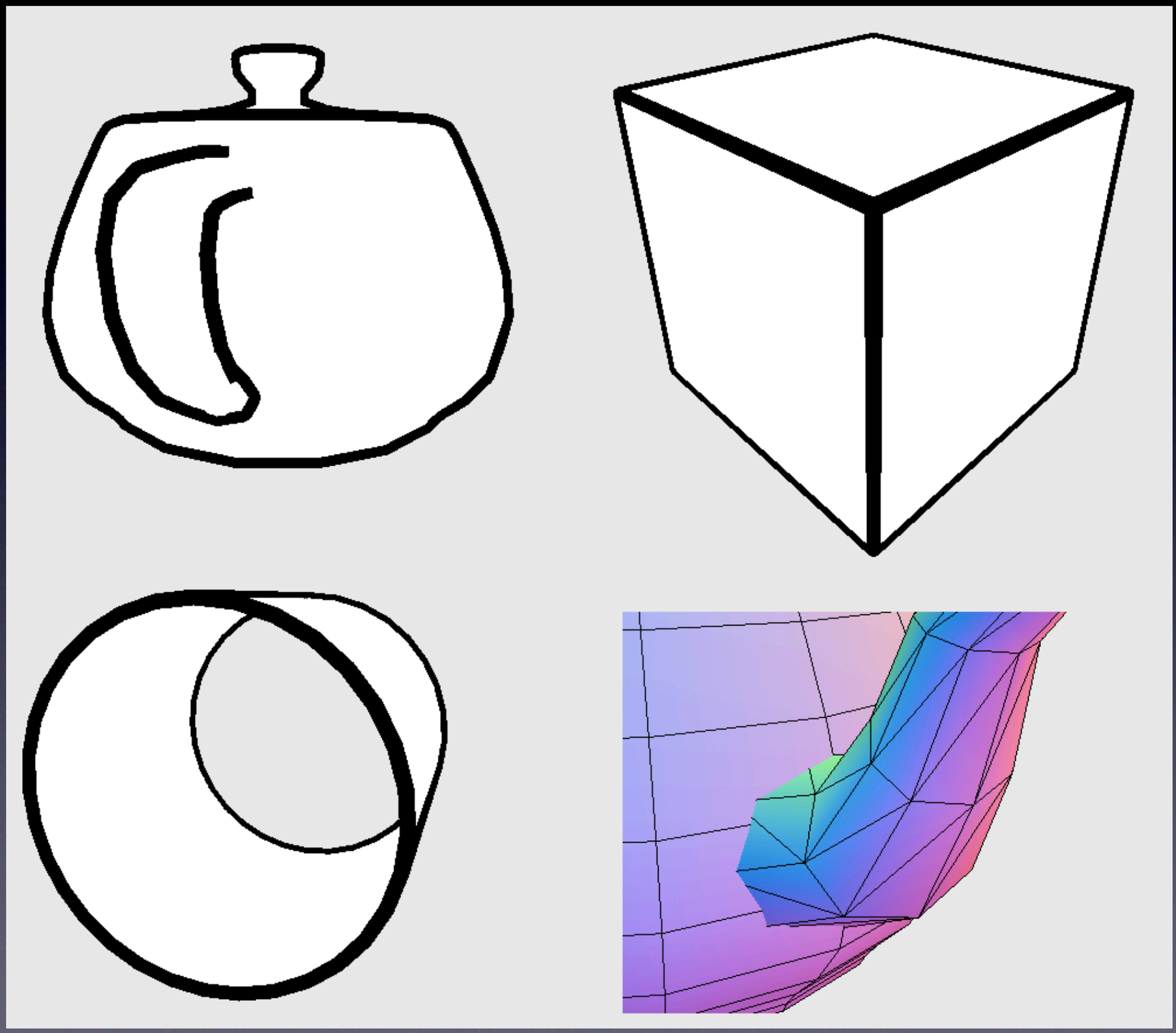
Types of Edges

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

Additional Terminology

- Silhouette - Contour on the outer border
- Ridge Crease - Crease that points at the camera
- Valley Crease - Crease that points away from the camera
- Drawable Edge - Polygon edges that will be drawn on a given frame, distinguished from all polygon edges

How to Identify Edge Types



Edge Practical Qualities

	Contour	Crease	Boundary	Marked	Intersection
View-Independent	No	Yes	Yes	Yes	Yes
Preprocessable (No Animate/Move)	No	Yes	Yes	Yes	Yes
Preprocessable (With Animation)	No	No	Yes	Yes	No
Preprocessable (World Movement)	No	Yes	Yes	Yes	No

Survey Overview

- Method categories
  - Hardware methods
  - Image-space method
  - Object-space methods
  - Miscellaneous methods

Hardware Methods

- Render edges as a bi-product of the order and method of rendering, no specific detection step
- Pros
  - No preprocessing necessary
  - Simple to implement
  - Supported on old platforms
- Cons
  - Usually only contour edges
  - Lacks customizability

Hardware Methods

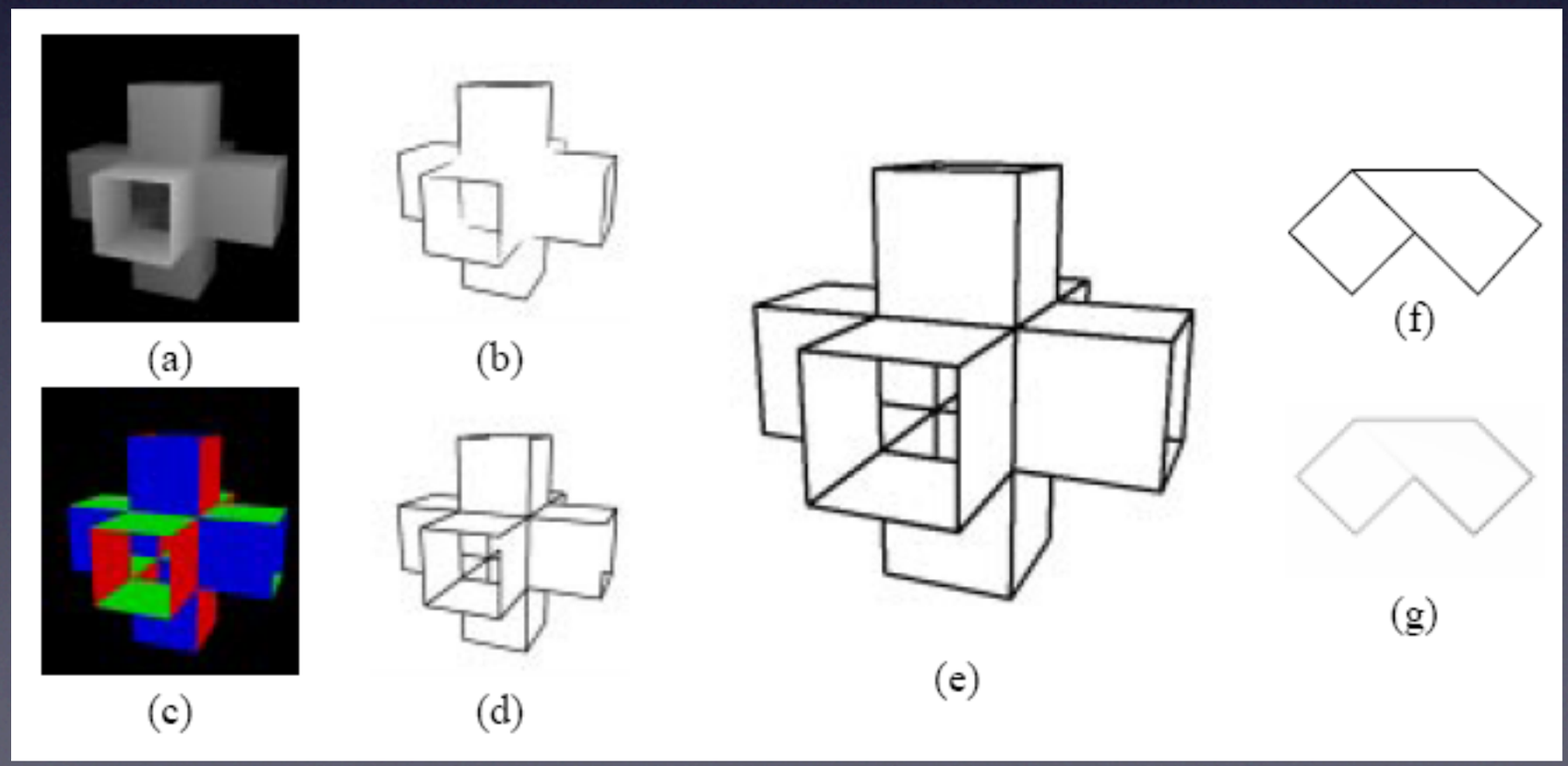
- For example,
  1. Render front-facing polygons
  2. Render back-facing polygons in wireframe mode with thickened edges
- Generates edges along the contours

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

- First, render a normal and depth buffer to textures (a, c)
- Then, apply image filter to both textures (b, d), and...
  - Final rendered image is modified by the detected edges (e)



Object-space Methods

- Detects edges in 3D space by checking individual polygon edges
- Pros
  - Very accurate
  - Easily controlled and customized
- Cons
  - Generally the slowest type of method
  - Usually requires preprocessing

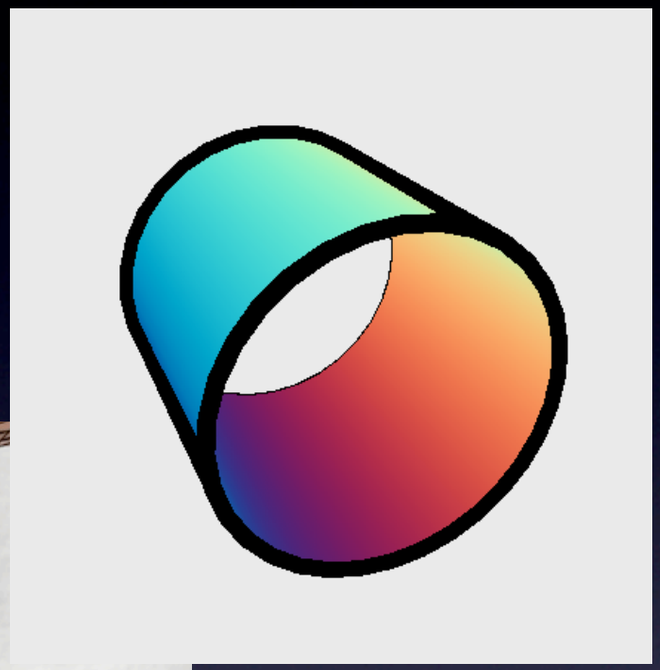
Object-space Methods

- At every frame, iterate through all unique polygon edges, checking them for drawability
- Those edges detected as drawable are expanded and rendered
- Customization is added here
  - Edge thickness
  - Edge color
  - Textured edges

Object-space Methods

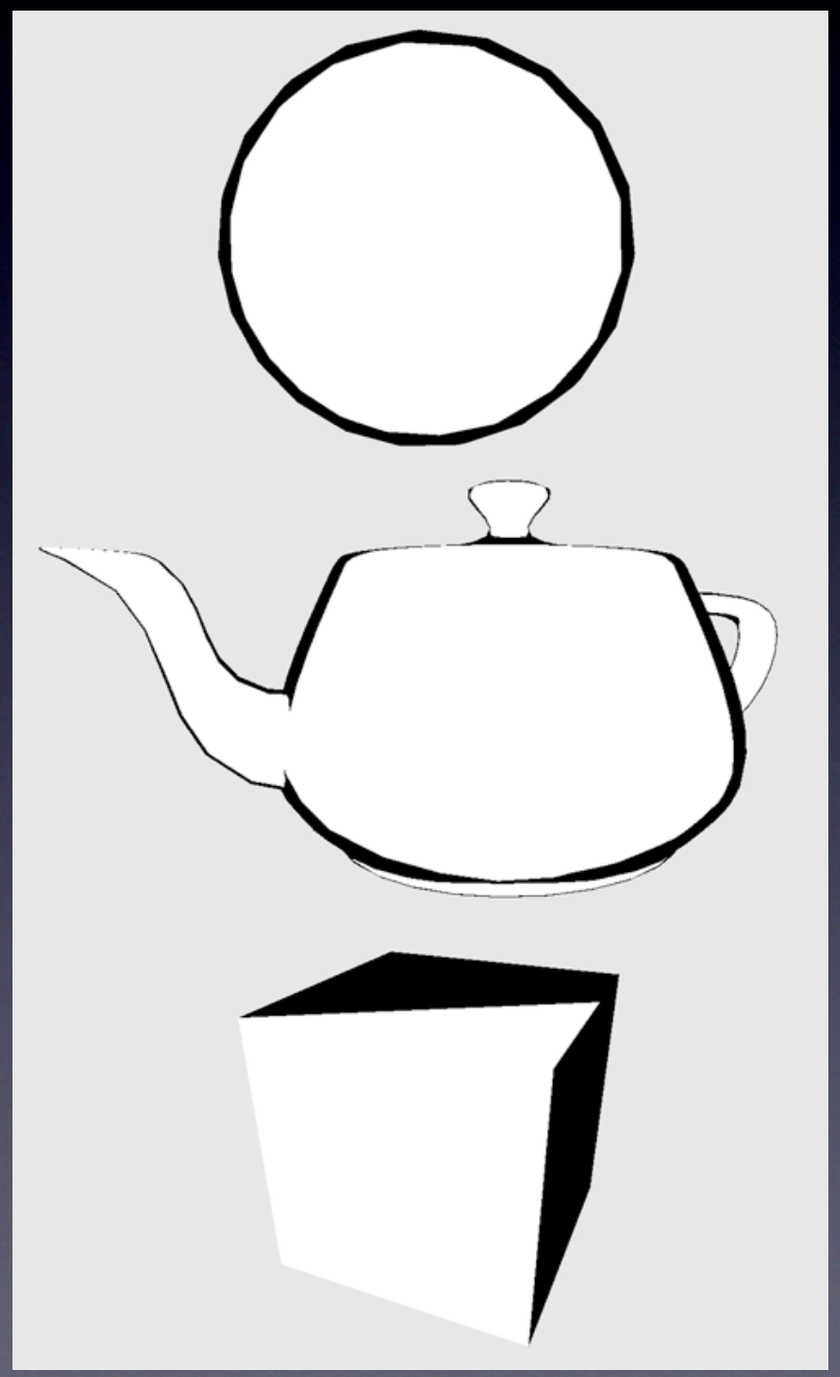
- Variations
  - Do detection on the GPU
  - Probabilistically check sub-set of total edges
  - Don’t test edges for co-planar polygons
  - Generate caps to fill gaps at edge connections
  - ...and many more

Object-space Examples



Miscellaneous Methods

- Everything else, for example:
  - Render black at places where the normal is nearly perpendicular to the view vector (right)
  - For a scaled-up copy of mesh, render it black, with inverted normals, and back-face culling (bottom)



Part II

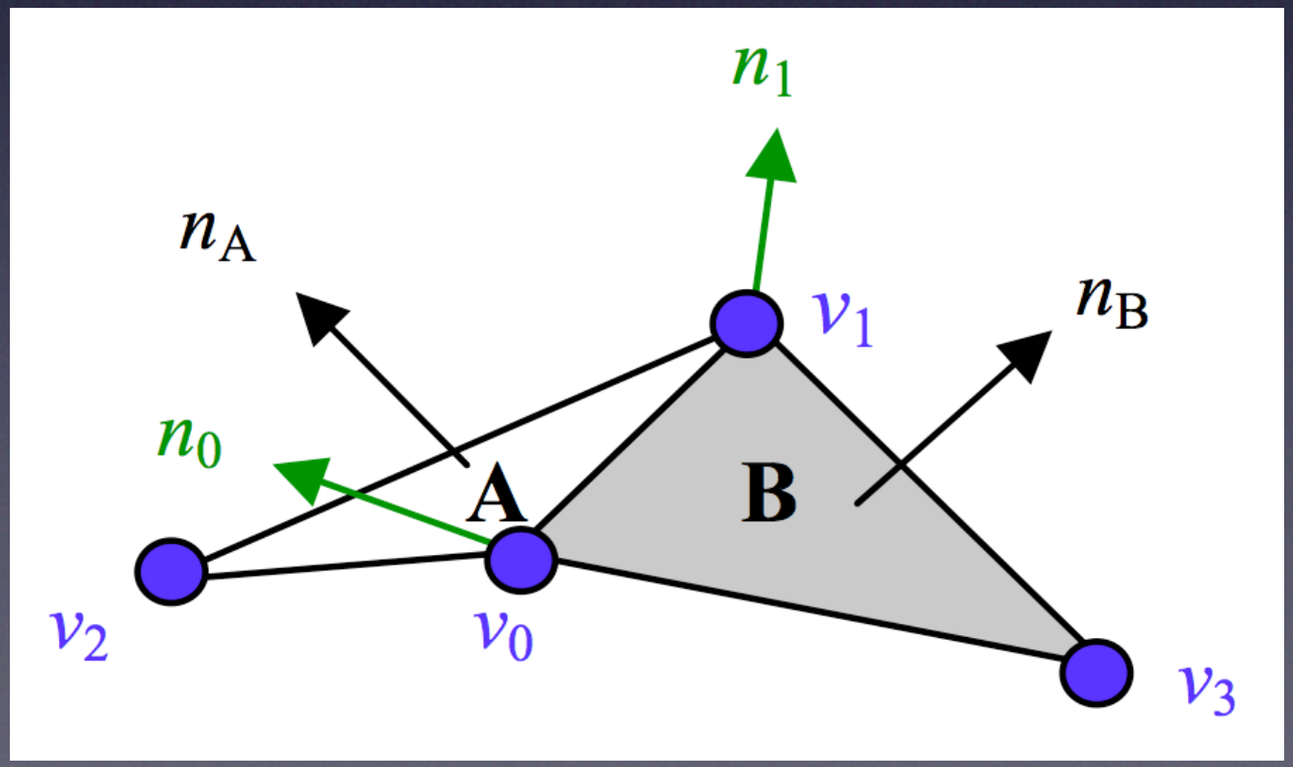
- Specific Area of Study
- Contributions
- Analysis
- Demo
- Future Work

Object-space Edges on GPU

- Morgan McGuire and John F. Hughes' paper from 2004 took the brute-force object-space edge detection and implemented it on the GPU
- Trades higher memory usage for a speed increase (due to parallel processing)
- 30 times slower than rendering the mesh normally
- 15 to 30 times faster than the same edge detection operation on the CPU for complex meshes

How it Works

- Find all unique edges in a mesh
- Obtain the edge data: v0, v1, v2, v3, n0, n1, r, and i
- Duplicate the edge data 3 times (4 total)
- Make sure i is unique (0, 1, 2, 3)
- Send to vertex buffers

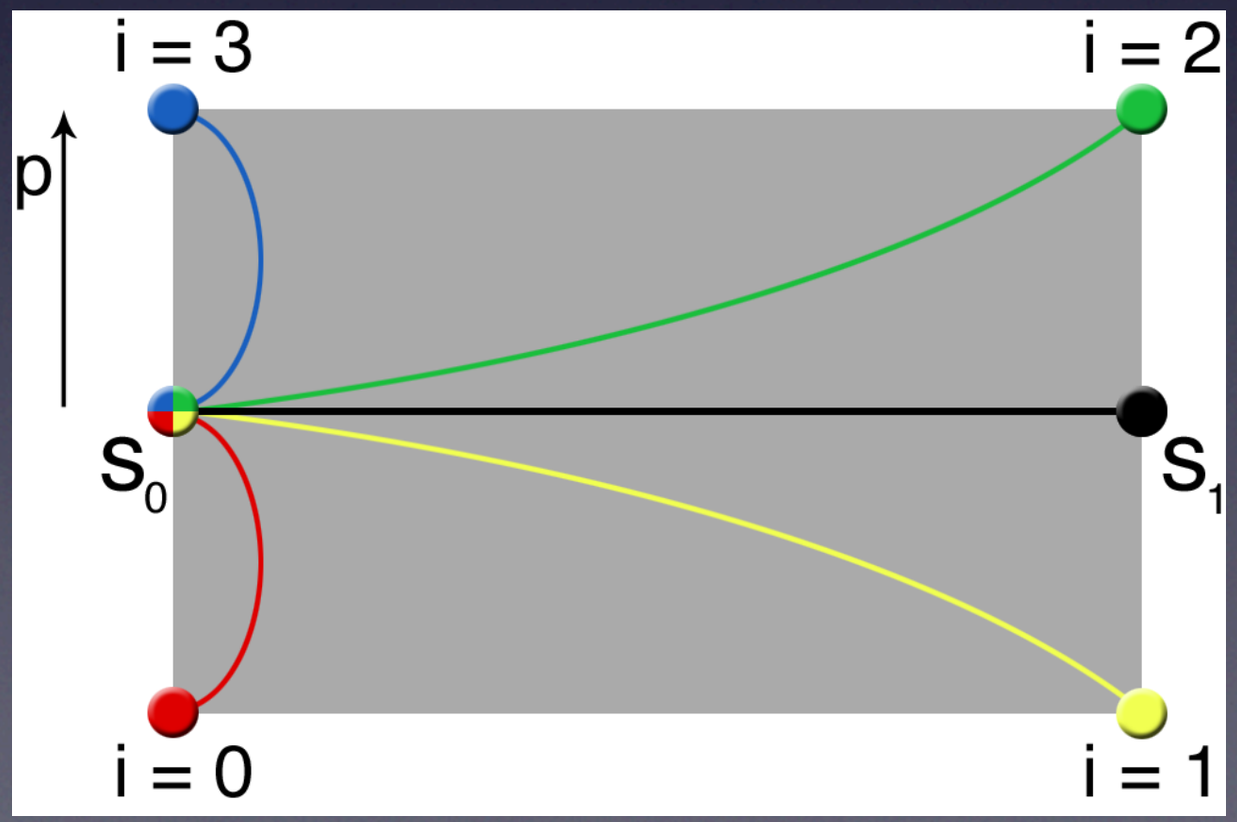


How it Works

- Drawability tests:
  - `Contour: [dot(nA, (eye - v0)) * dot(nB, (eye - v0)) < 0]`
  - `Ridge Crease: [dot(nA, nB) < -cos(θR)] && dot((v3 - v2), nA) <= 0]`
  - `Valley Crease: [dot(nA, nB) < -cos(θV)] && dot((v3 - v2), nA) > 0]`
  - `Marked: [v3 == v0]`
  - `Boundary: [v3 == v0]`
- Where
  - θR and θV are user-defined angles for the ridge and valley creases
  - nA and nB are the left and right adjacent face normals
  - eye is the world camera position

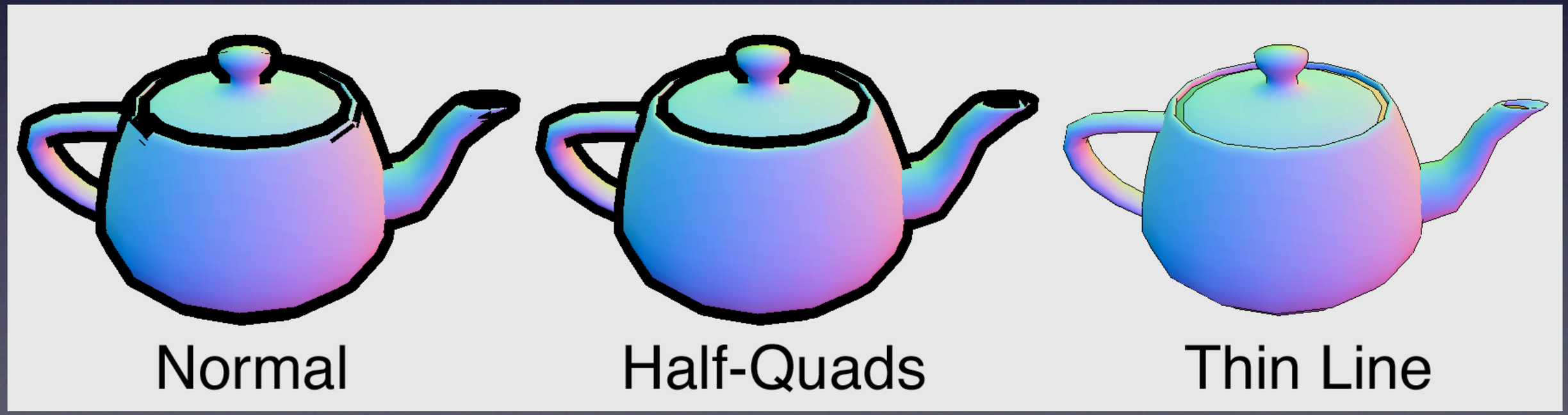
How it Works

- For drawable edges, generate the four points of a screen-space quad
- For non-drawable edges, generate a degenerate quad
- The p vector is the perpendicular vector to the screen-space edge vector



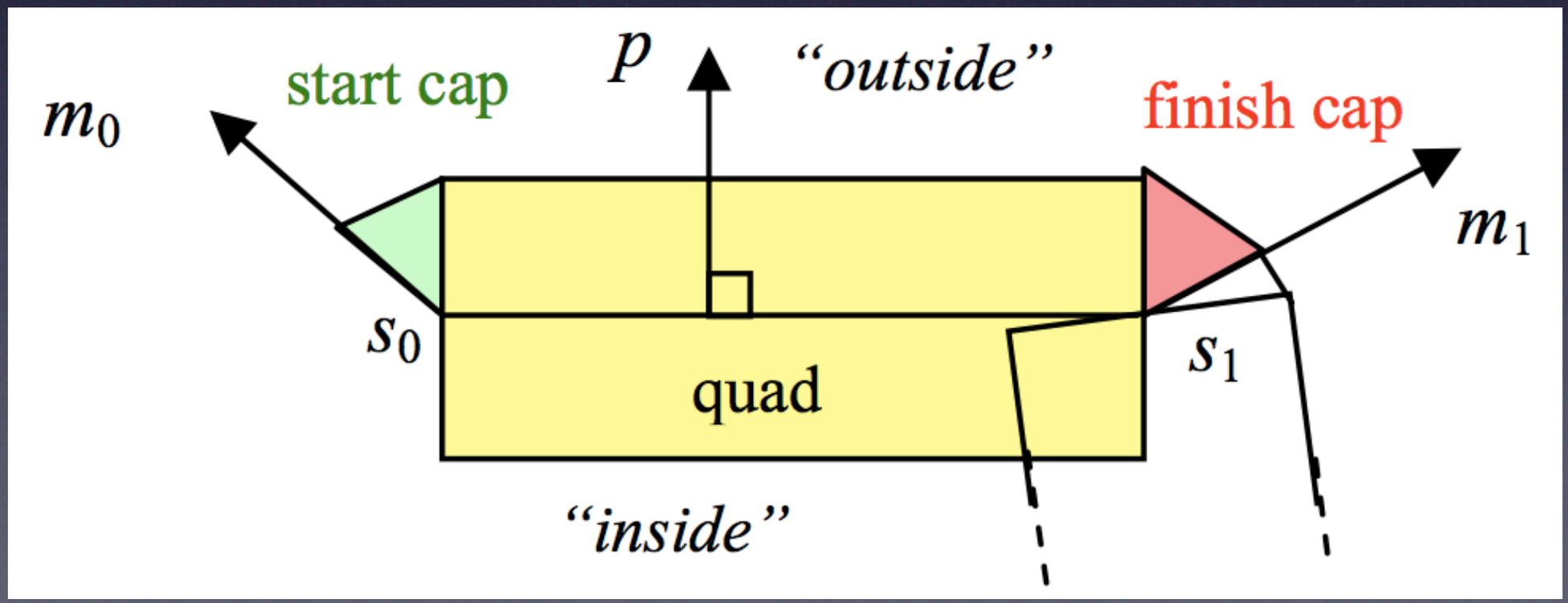
Variations

- Rasterized lines can be used instead of quads, requiring only two duplicates
- Contour edges can be rendered as a “half-quad,” which only renders outside the mesh



Capping

- McGuire and Hughes generated two half caps
- One on each side of drawable edges
- Line up along the screen-space vertex normals



The Complete Setup

- Four Passes
  - Render mesh with depth offset
  - Render thick edges
  - Render edge caps for left vertex
  - Render edge caps for right vertex

Problems

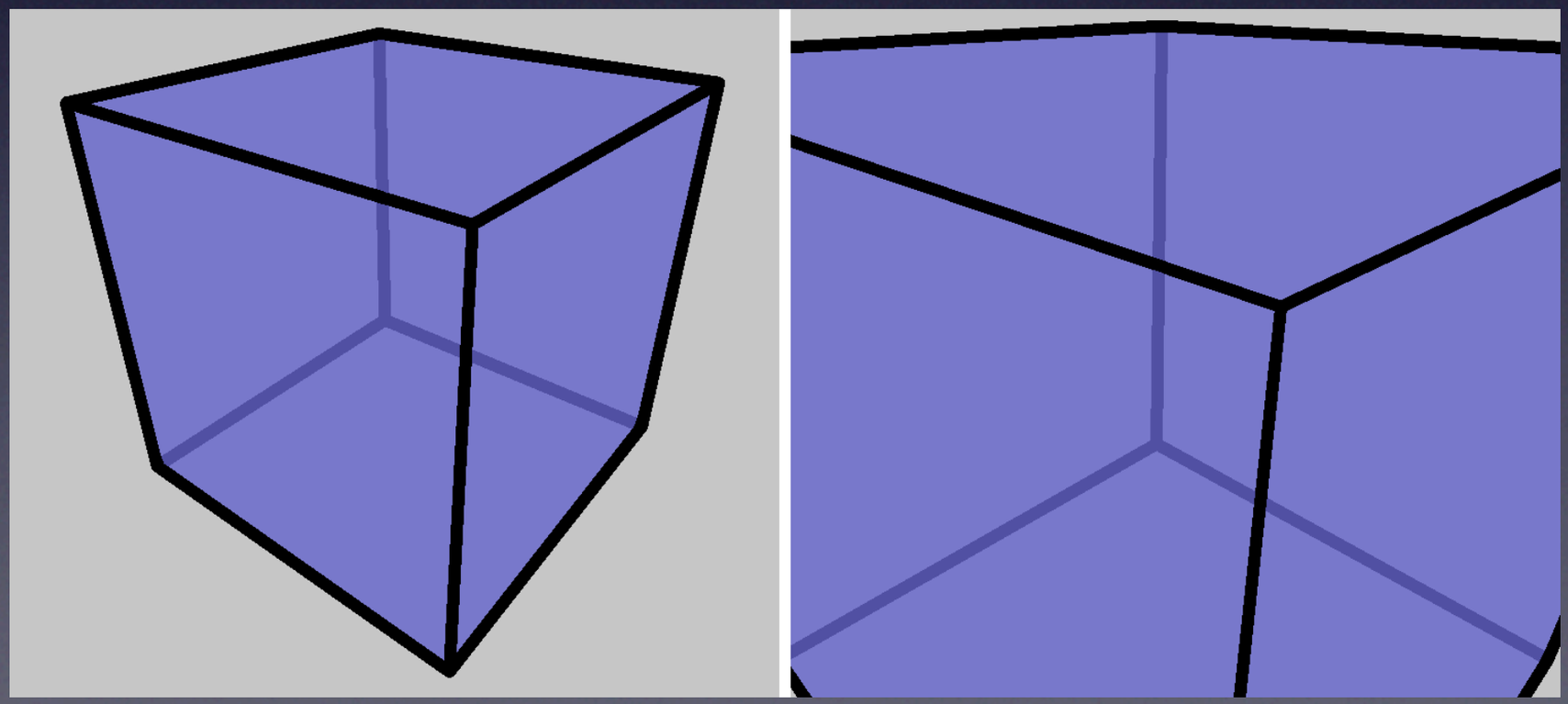
- Screen-space thickened edges can overpower the mesh
- Normals may not represent curvature of the surface creating bad caps (called “bad normal” problem)
- High memory usage and computation duplication
  - McGuire and Hughes suggested the use of geometry shaders and data textures

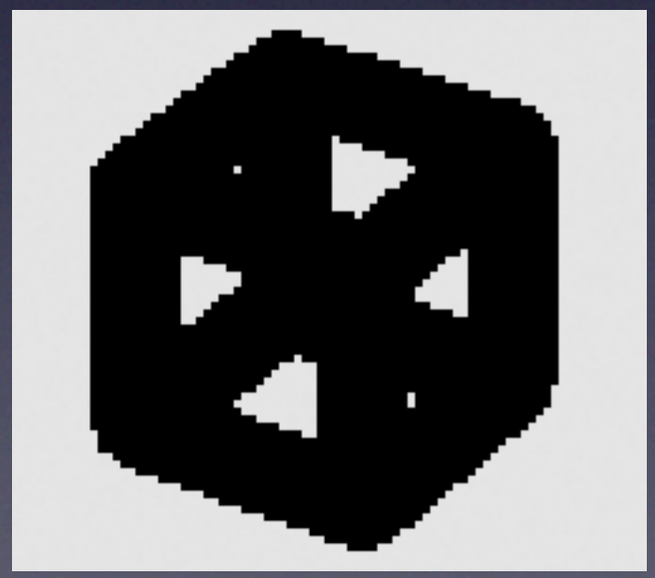
Contributions

- Depth-based thickness of edges
- Solve the “bad normal” problem
- Reduce render passes with alternate edge types
- Attempt to use new technology to make better edges and caps
  - Fewer computations
  - More accurate caps

Screen-based Thickness

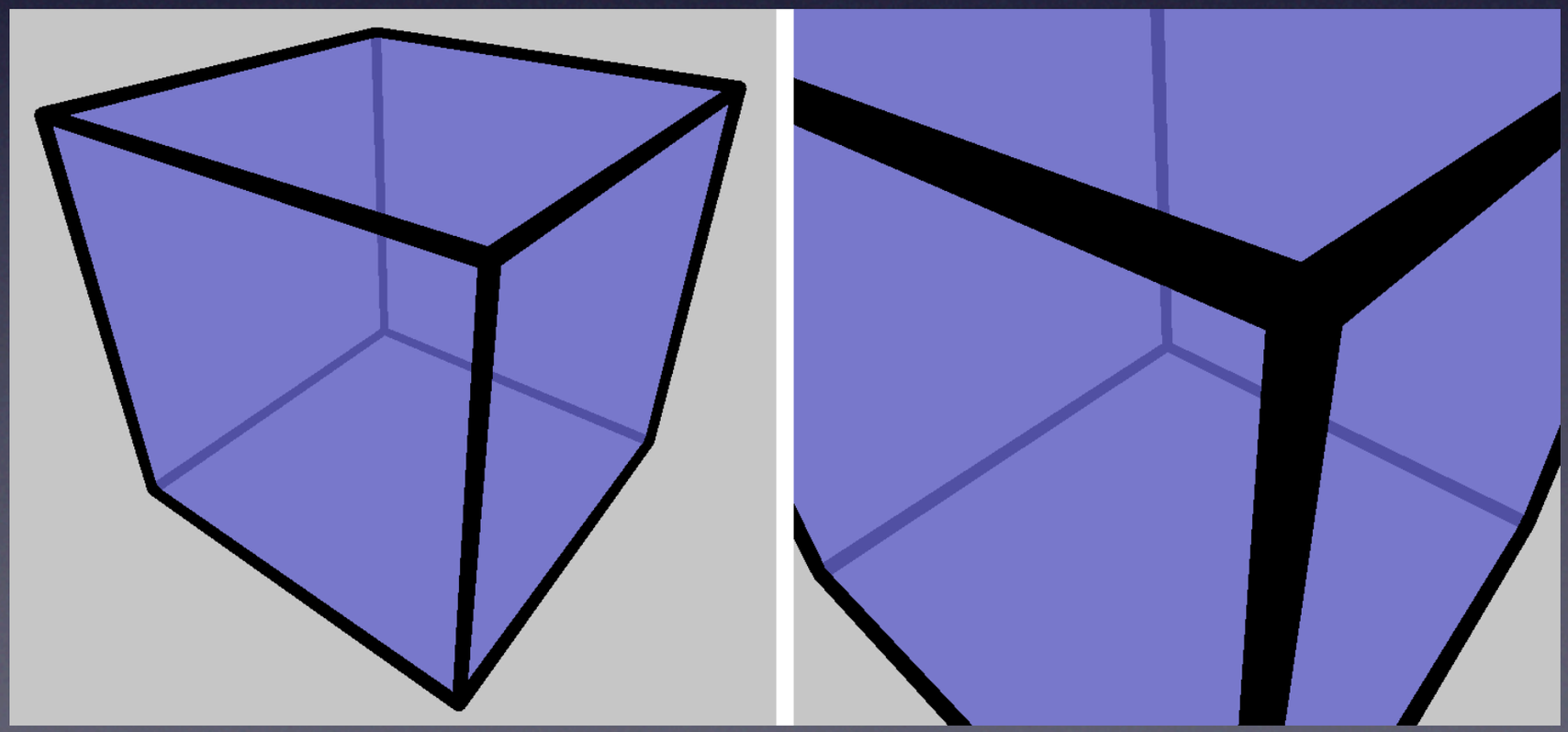
- Affects object depth perception
- Distant edges can overwhelm the mesh

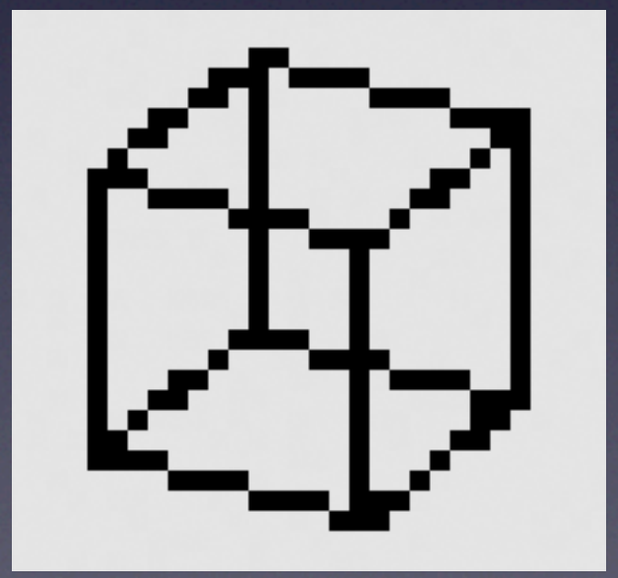




Depth-based Thickness

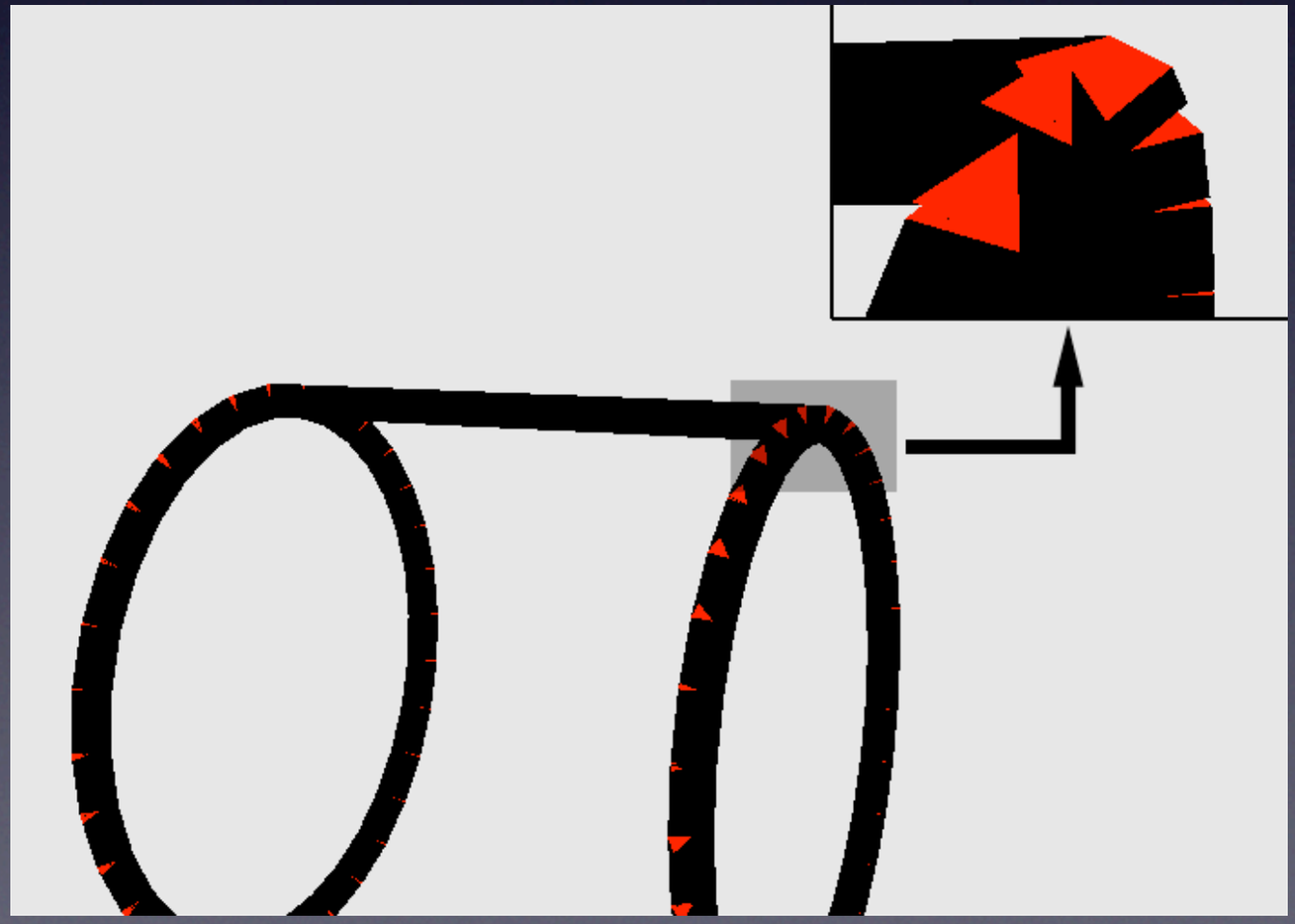
- Fixes both previous issues
- Adding a minimum prevents loss of distant edges





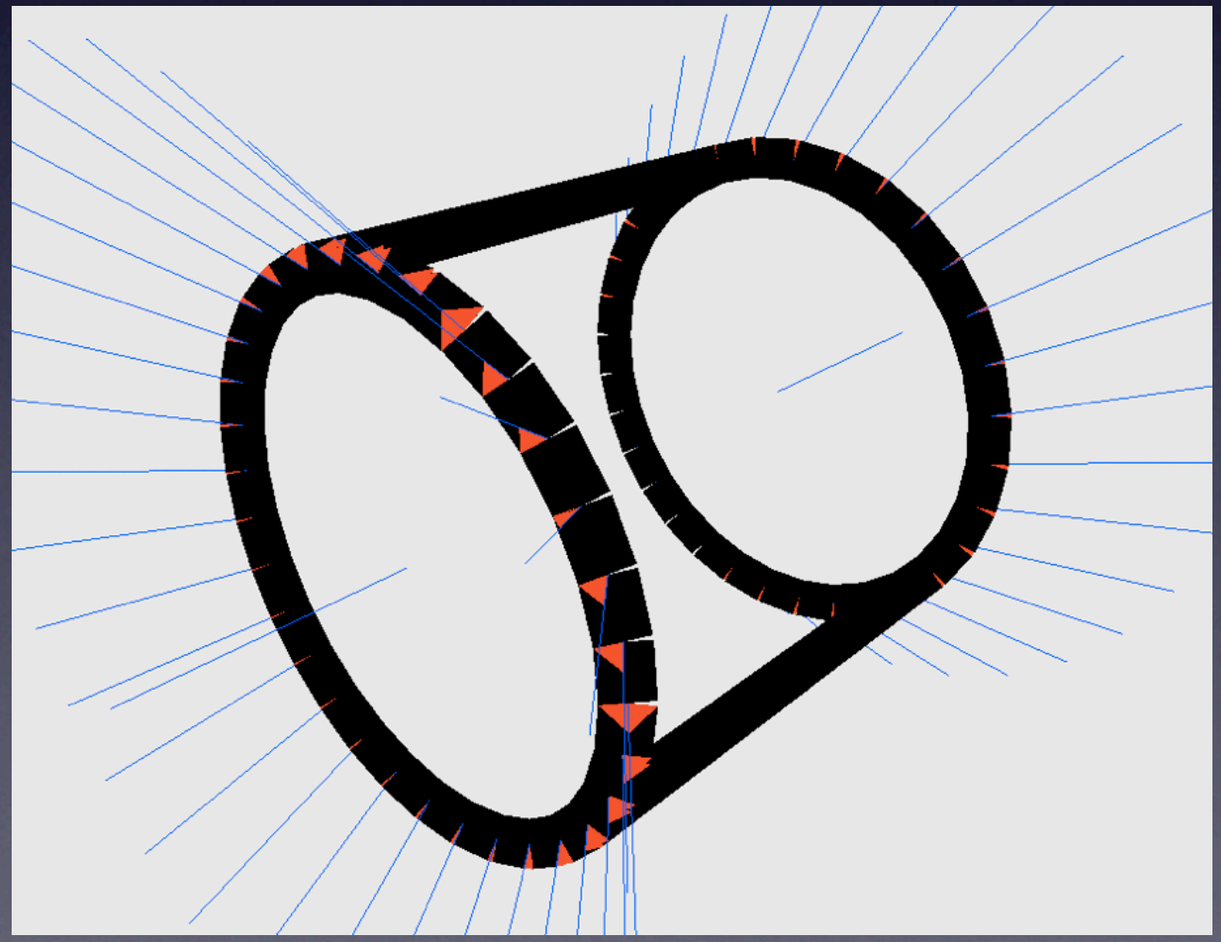
Bad Normal Problem

- Case 1: three or more drawable edges converge
  - Three (or more) normals are correct, though edge redundancy usually prevents complete failures



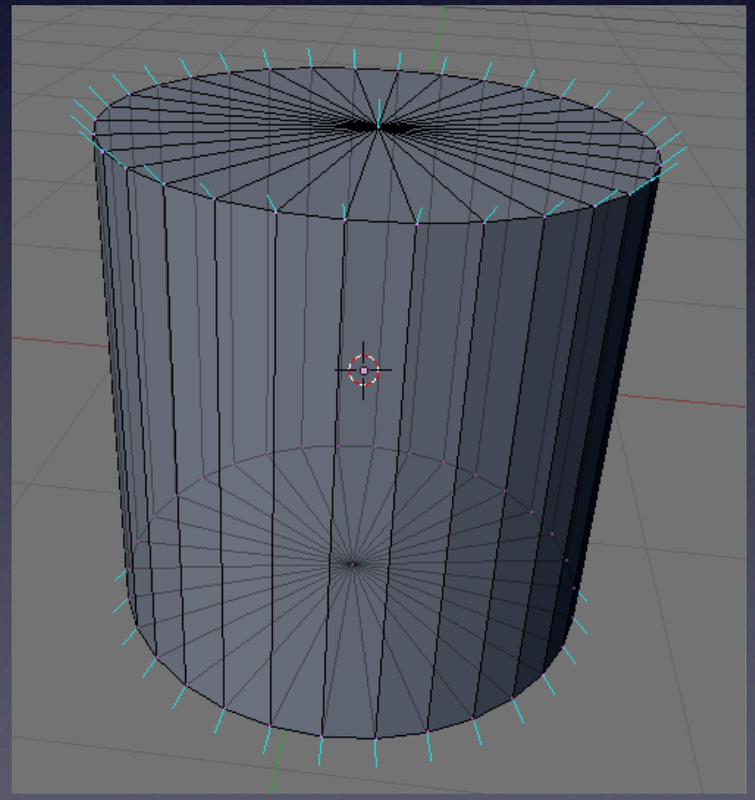
Bad Normal Problem

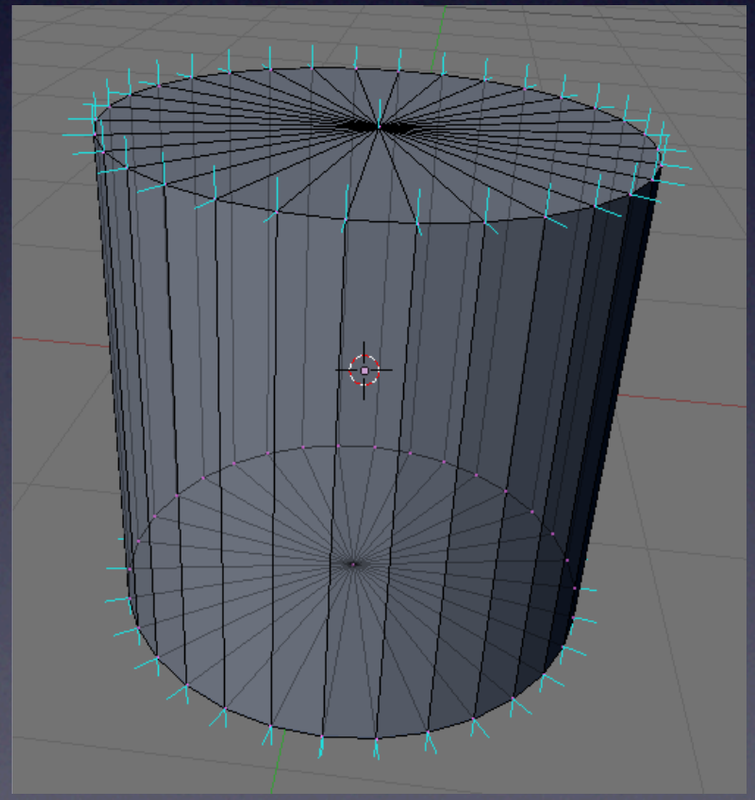
- Case 2: curved area abuts a flat area
  - Only one normal is needed

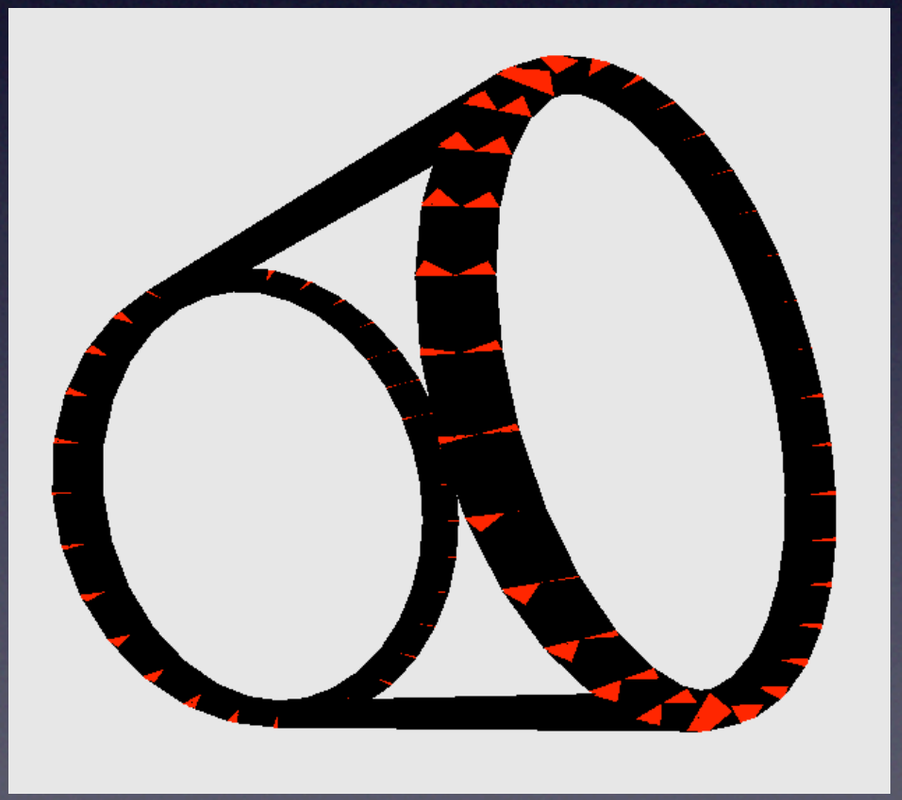


Solving Bad Normals

- Use “edge splitting” command in 3D editor and allow for duplicate edges

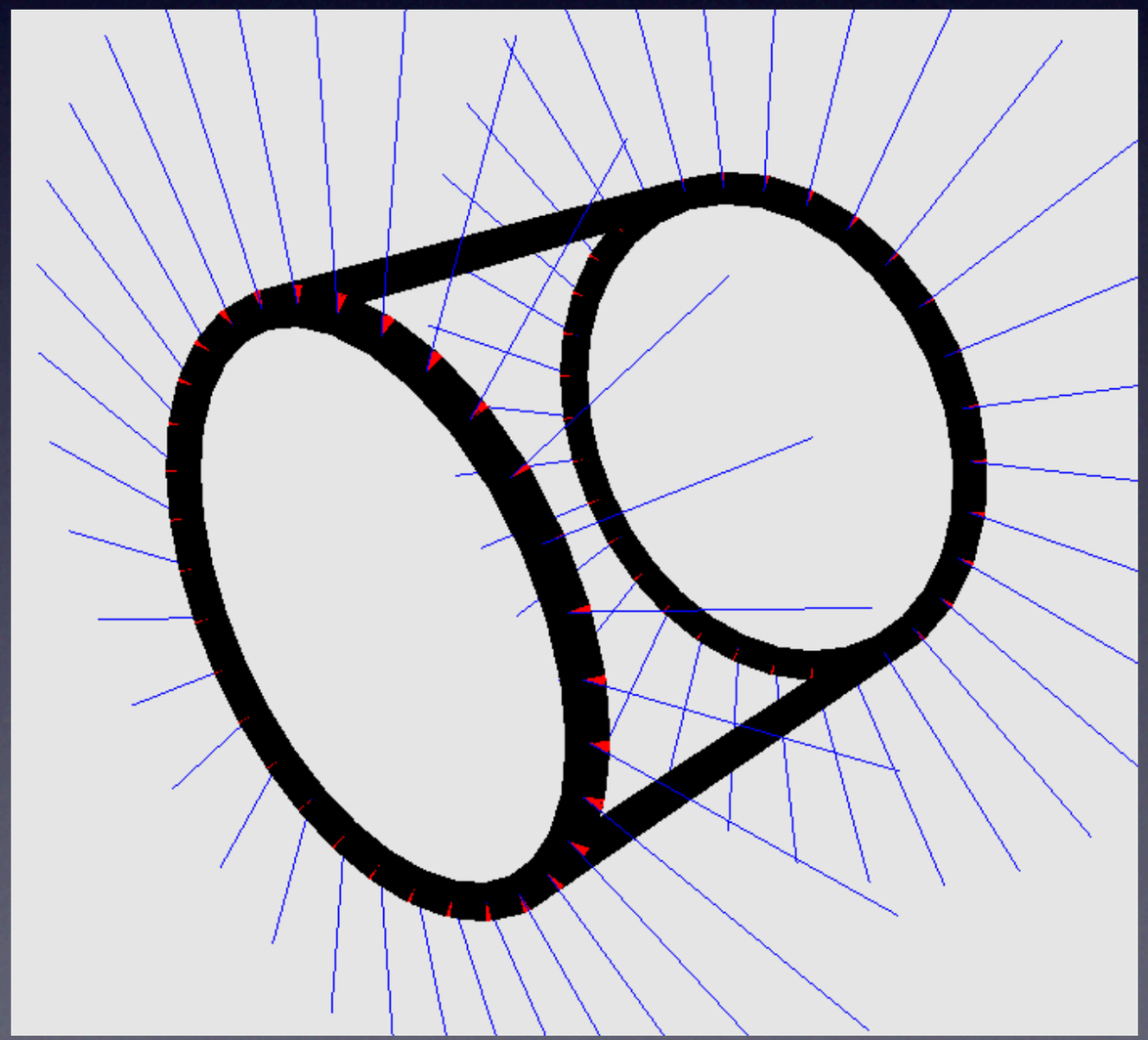






Solving Bad Normals

- Pick better normals for the edges during the mesh creation process



Alternate Edge/Cap Types

- Combine caps into edges to reduce render passes
  - Half Hex Method
  - House Method
  - Plug Method
- Double caps to handle bad normals

Demo

New Technology

- Last year, version 1.0 of OpenCL (Open Compute Library) was released
- General, massively parallel computation on GPUs and other devices
- Interoperable with OpenGL

Non-Duplicate Parallel Edge Detection and Capping

- OpenCL's abilities allow another method of edge detection similar to McGuire and Hughes'
- Removes calculation and data duplication
- Allows for higher accuracy caps

How it Works

- Store only adjacency information
- Use mesh vertex data already on the GPU
  - Index list describes the edge, not the polygons
- Edge detection and generation is largely the same as in McGuire and Hughes' paper
- Output vertices must be temporarily stored

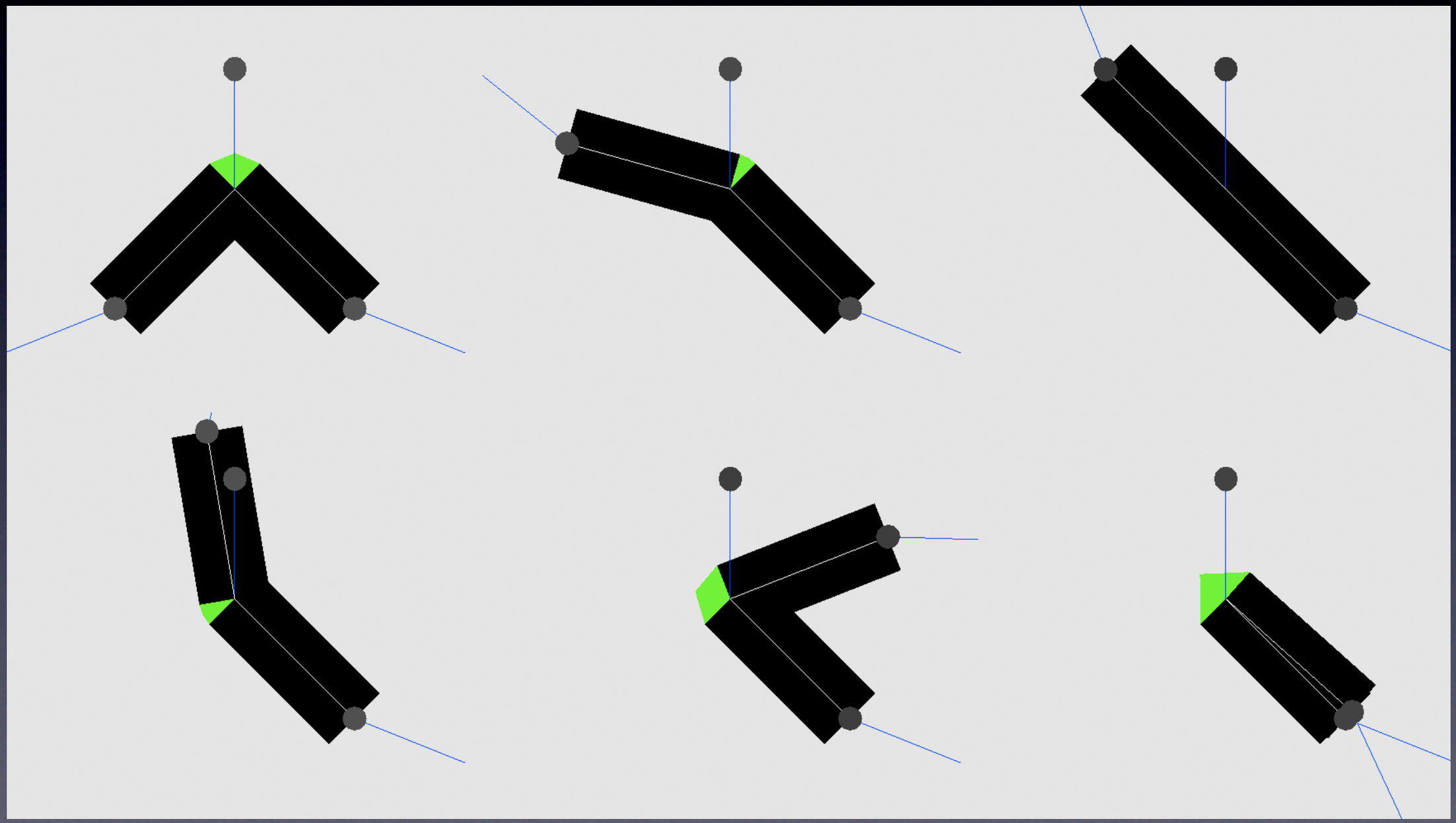
How it Works

- Output from the edge detecting step can be used to detect and create appropriate caps
- Cap buffer stores indexes of connected edges
- No normals needed

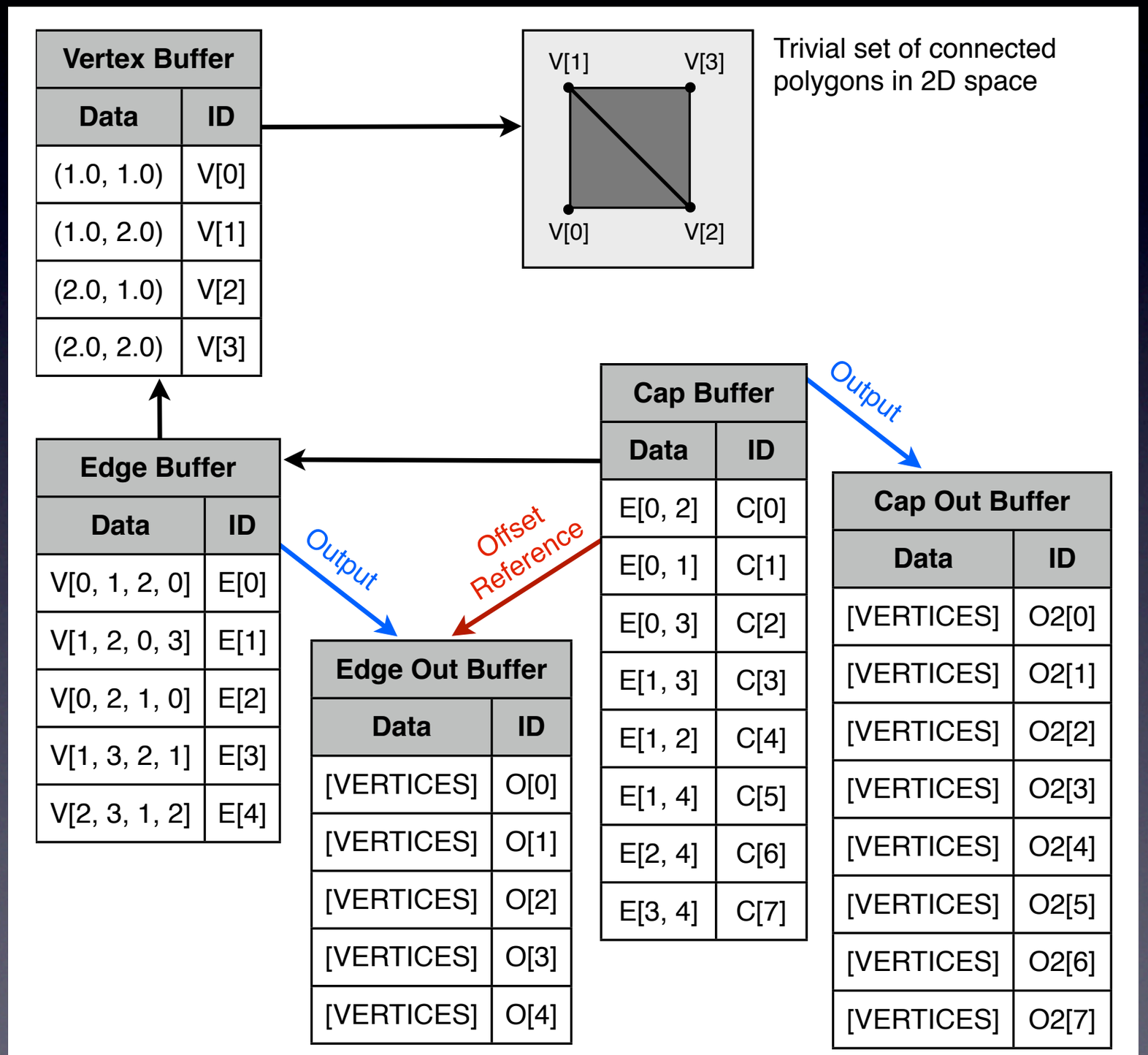
How it Works

- Determine the vectors along the edges from the edge detection output
- Calculate the “middle vector” (used like a normal)
  - `-Normalize(normalizedEdgeVector1 + normalizedEdgeVector2);`
- With the middle vector and the two edges’ perpendicular vectors, generate a quad that perfectly fills the gap

Capping with Adjacency



How it Works


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]
V[1]
V[3]
V[0]
V[2]
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]

The Complete Setup

- 2 Compute Passes
  - Create Edges
  - Create Caps
- 3 Render Passes
  - Mesh
  - Edges (simple)
  - Caps (simple)

Setbacks

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

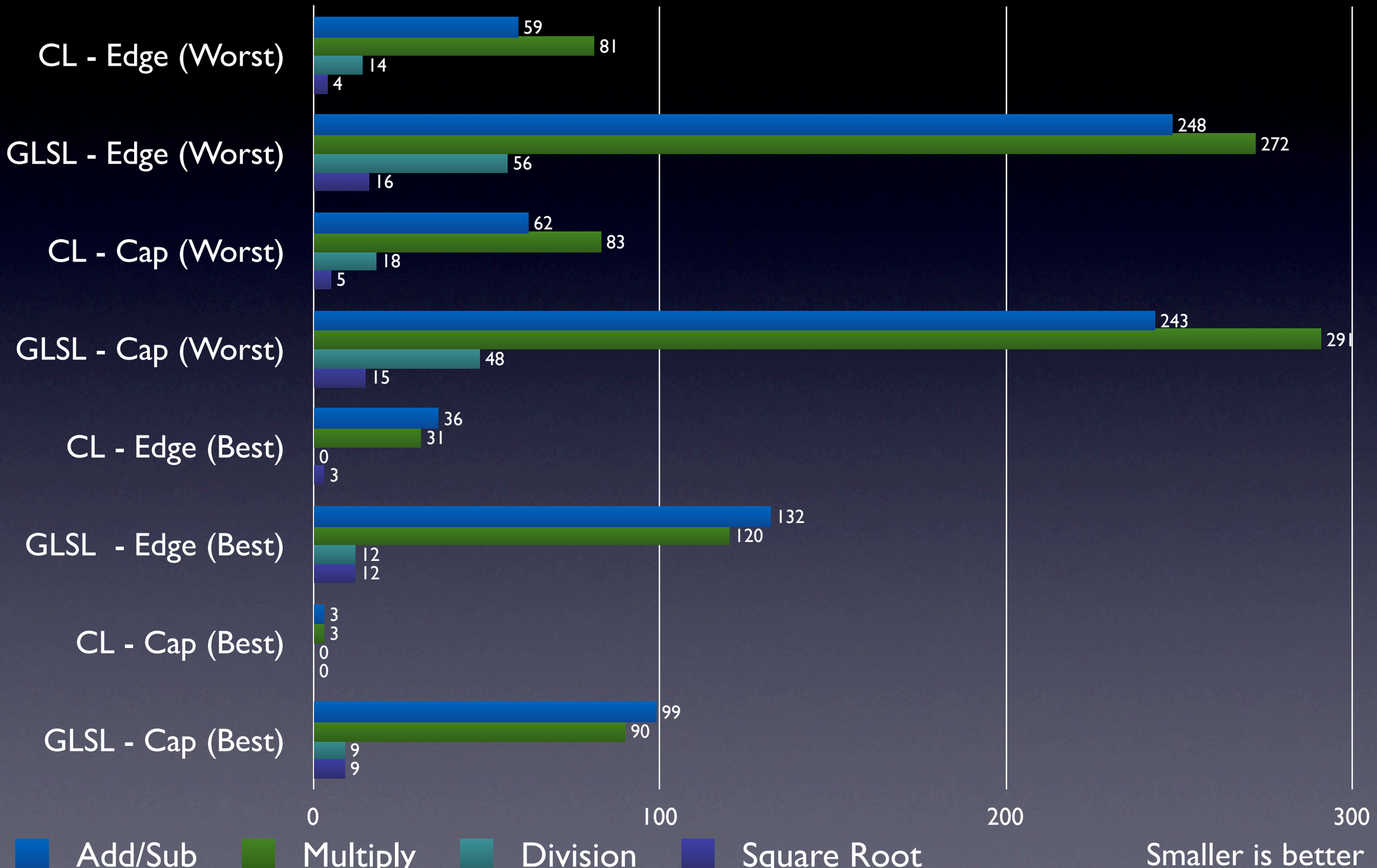
Method Alterations

- 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

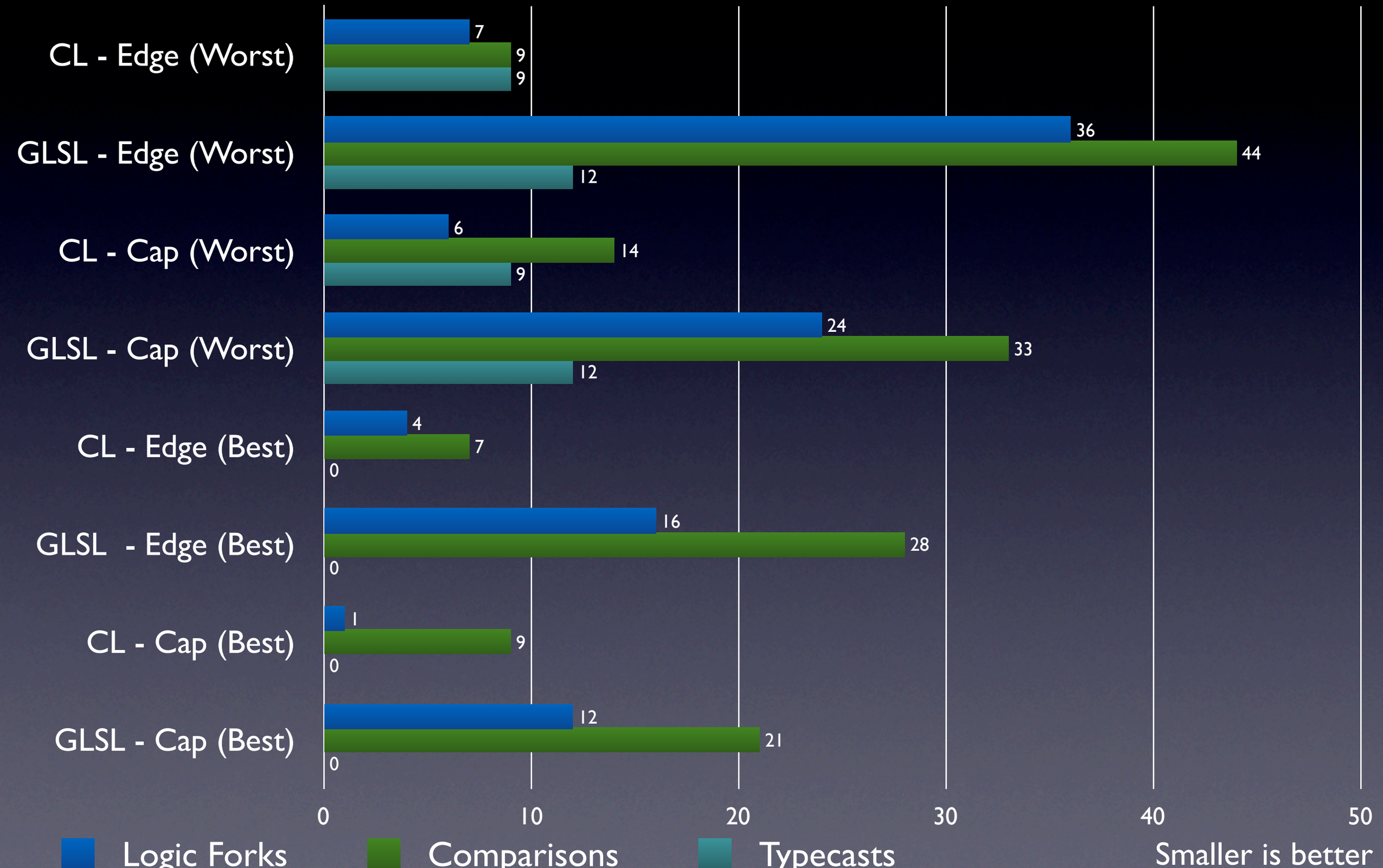
Comparative Analysis

- 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

In general, the CL method is about as memory intensive as McGuire's method
Values in bits, assumes 32 bit floats/integers
Smaller is better

Memory Usage with McGuire and Hughes' Caps

	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 Comparison

	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 Comparisons: 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 Comparisons: 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 Comparisons: GPU with McGuire and Hughes' Capping

	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 meshes 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 contributions (depth-based thickening, methods of reducing bad normals, and alternate edge creation methods) work now

Demo

Future Work

- Fully implementing McGuire and Hughes' method down to the duplicate data would allow for memory caching and probably faster speeds
- My capping method could be implemented with a geometry shader/data texture setup
- Microsoft's DirectCompute has an advantage over OpenCL in that it can send data directly to the graphics pipeline
- Chris Peters suggested another capping method that could lead to equal quality caps without checking every possible combination of edges

Questions?