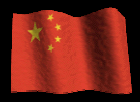
Fast Triangle Reordering for Vertex Locality and Reduced Overdraw

Pedro V. Sander

Hong Kong University of Science and Technology 

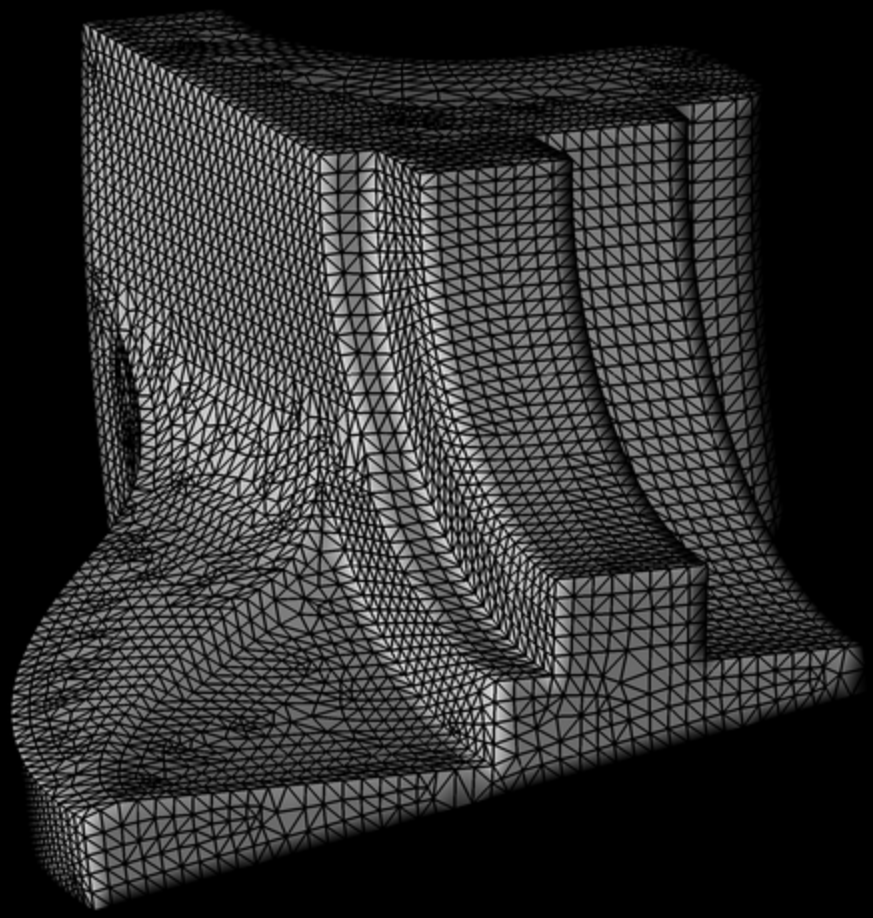
Diego Nehab

Princeton University 

Joshua Barczak

3D Application Research Group, AMD 

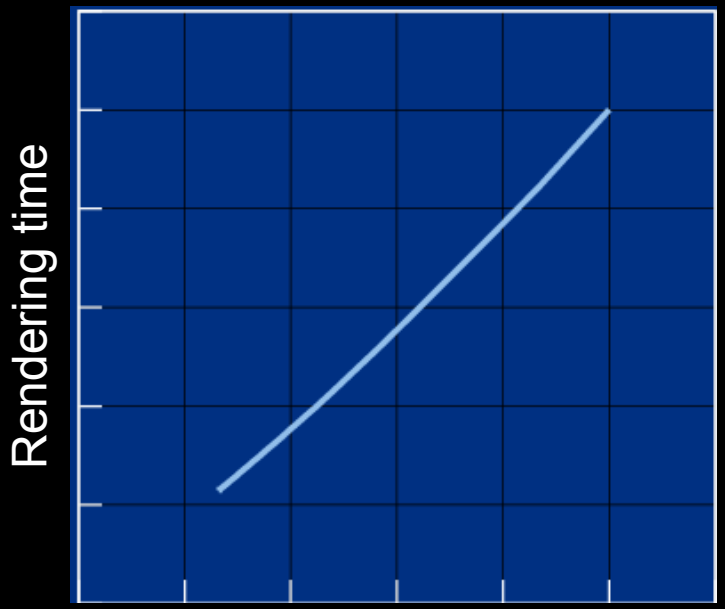
Triangle order optimization



Objective: Reorder triangles to render meshes faster

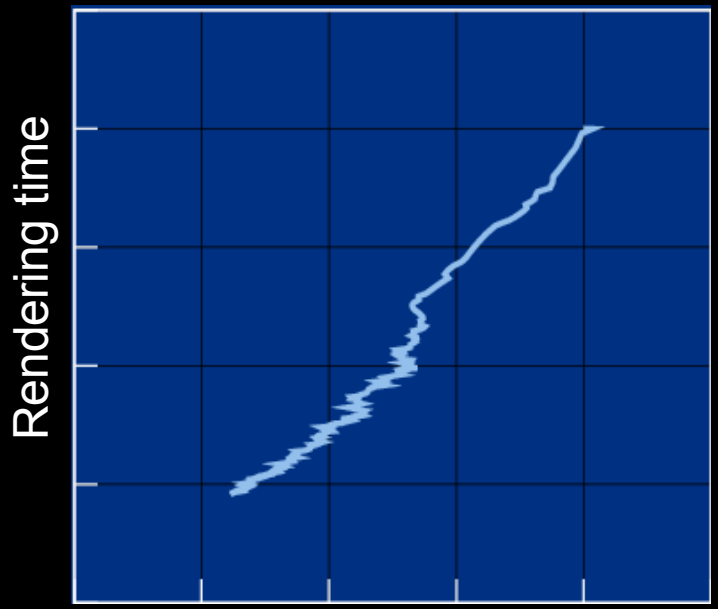
Motivation: Rendering time dependency

Vertex-bound scene



vertices processed

Pixel-bound scene



pixels processed

Reduce!
(transparently)

Goal

- Render faster
  - Two key hardware optimizations
    - Vertex caching (vertex processing)
    - Early-Z culling (pixel processing)
  - Reorder triangles efficiently at run-time
  - No changes in rendering loop
  - Improves rendering speed transparently

Algorithm overview

- Part I: Vertex cache optimization
- Part II: Overdraw minimization

Part I:
The Post-Transform Vertex cache

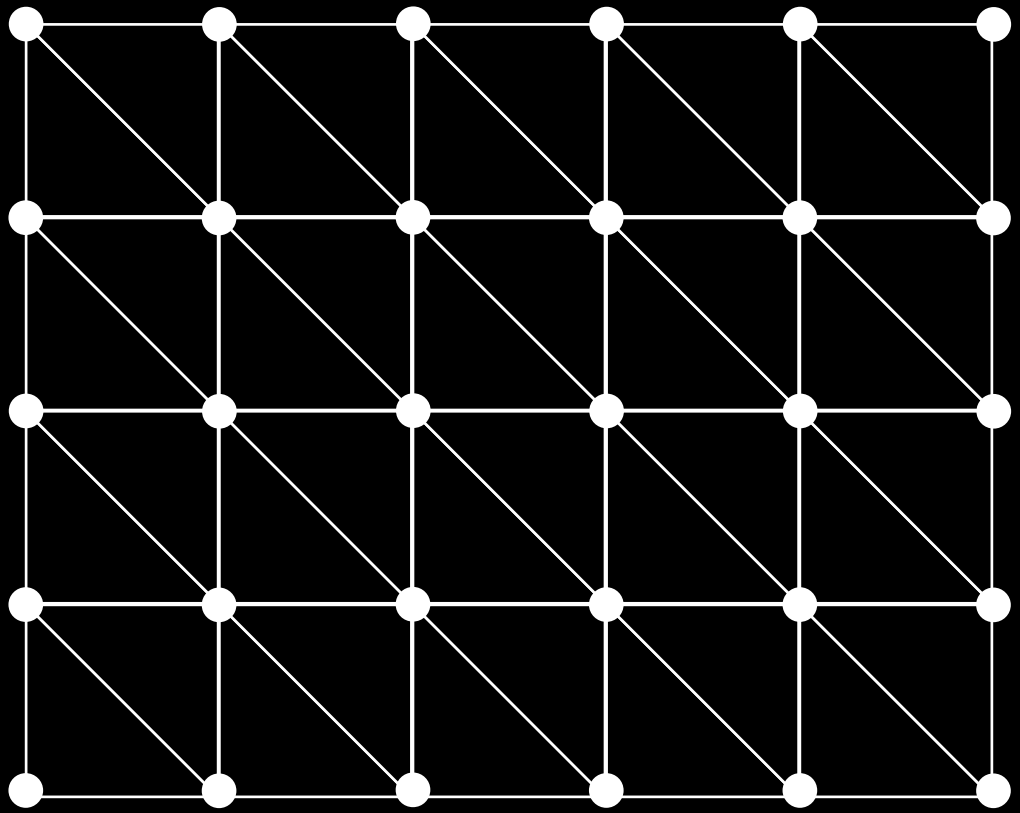
- Transforming vertices can be costly
- Hardware optimization:
  - Cache transformed vertices (FIFO)
- Software strategy:
  - Reorder triangles for vertex locality
- Average Cache Miss Ratio (ACMR)
  - # transformed vertices / # triangles
  - varies within [0.5–3]

ACMR Minimization

- NP-Complete problem
  - GAREY et. al [1976]
- Heuristics reach near-optimal results [0.6–0.7]
  - Hardware cache sizes range within [4–64]
- *Substantial* impact on rendering cost
  - From 3 to 0.6 !
  - Everybody does it

Parallel short strips

Very close to optimal!



~0.5 ACMR

Previous work

- Algorithms sensitive to cache size
  - MeshReorder and D3DXMesh [HOPPE 1999]
  - K-Cache-Reorder [LIN and YU 2006]
  - Many others…
  - Recent independent work [CHHUGANI and KUMAR 2007]

Previous work

- Algorithms oblivious to cache size
  - dfsrendseq [Bogomjakov et al. 2001]
  - OpenCCL [Yoon and Lindstrom 2006]
- Based on space filling curves
- Asymptotically optimal
  - Not as good as cache-specific methods
  - Long running time
- Do not help with CAD/CAM

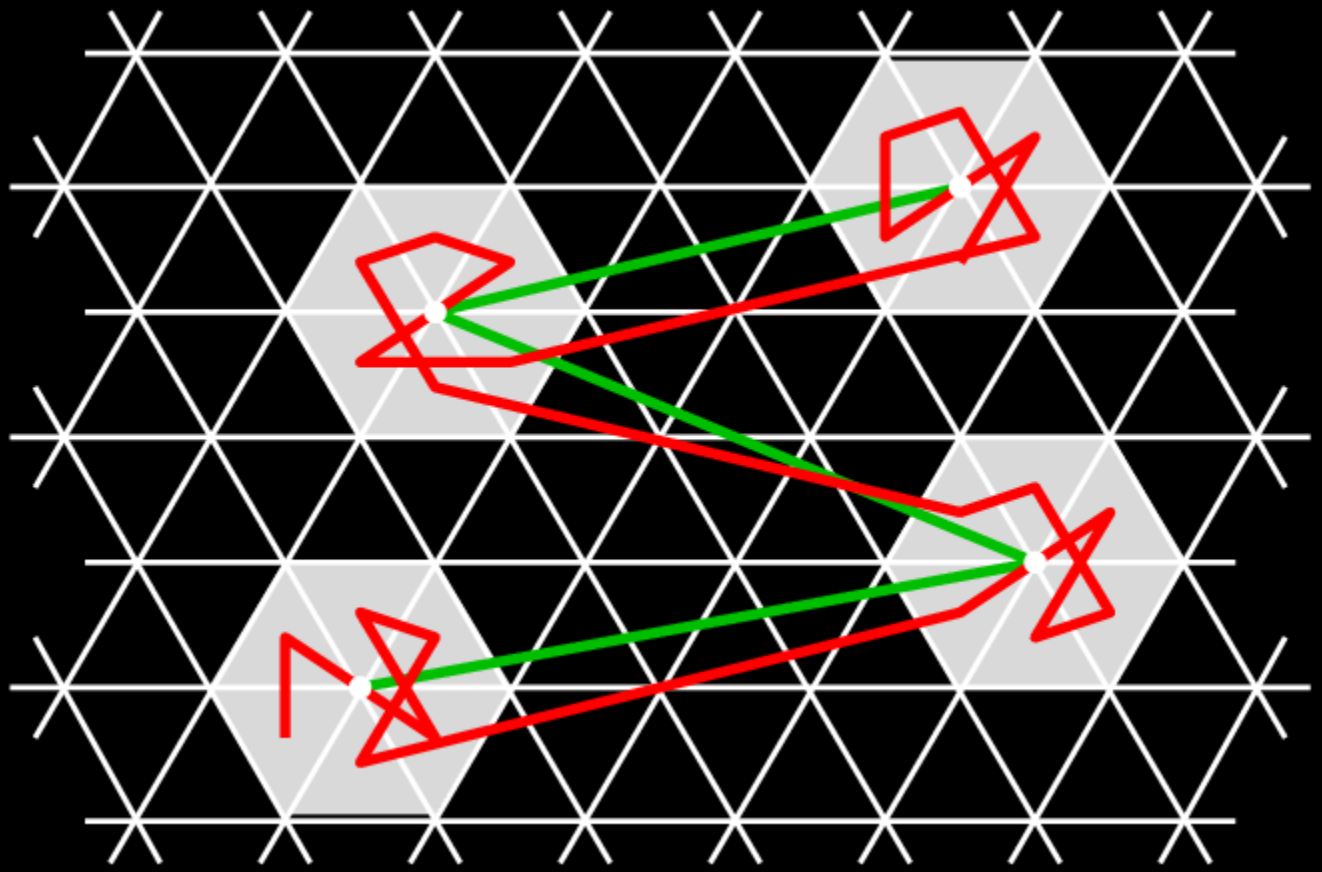
Our objective

- Optimize at run-time
  - We even have access to the *exact* cache size
- Faster than previous methods, i.e., O(t)
  - Must *not* depend on cache-size
- Should be easy to integrate
  - Run directly on index buffers
- Should be general
  - Run transparently on non-manifolds

“Triangle-triangle” adjacency unnecessary

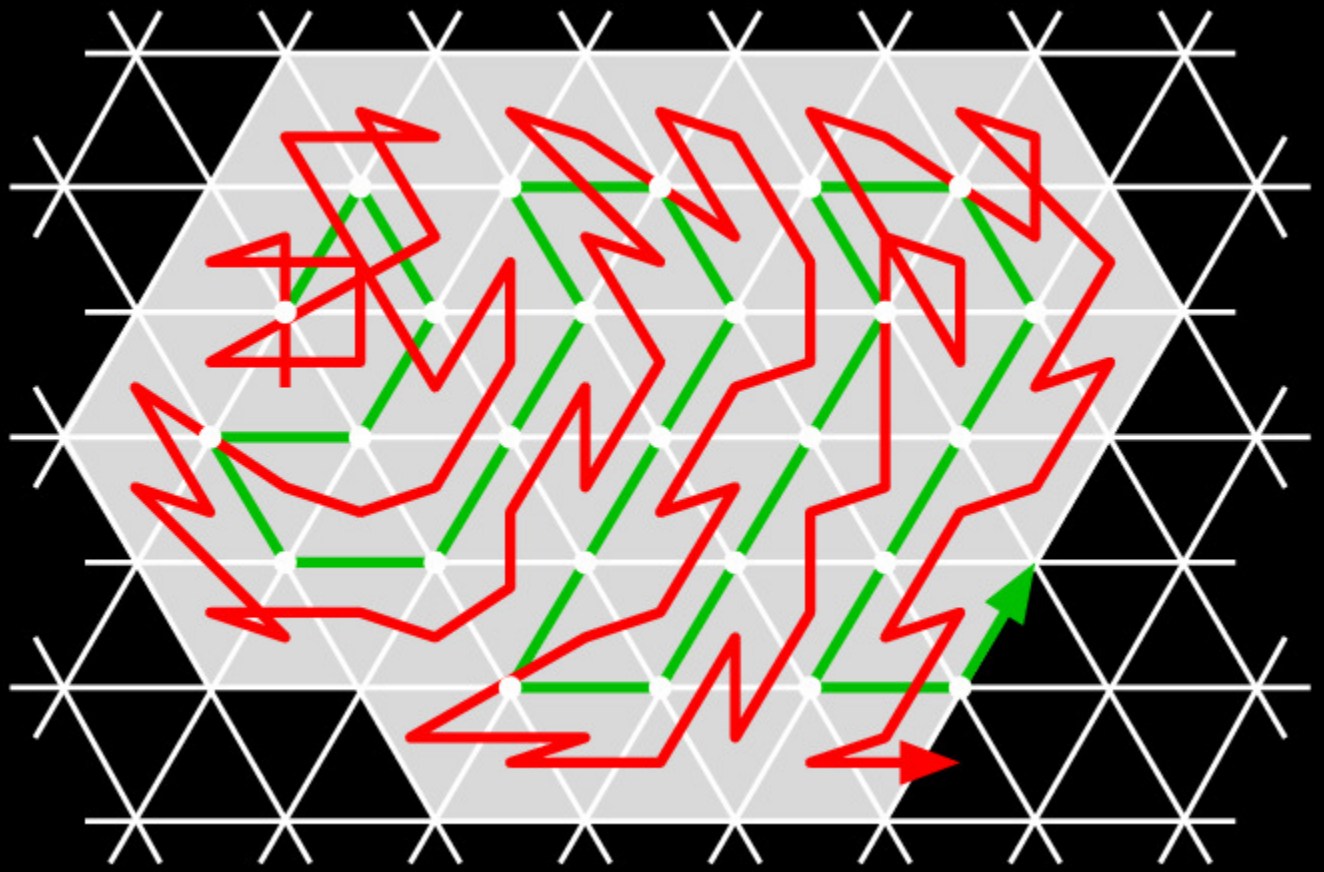
- Awkward to maintain on non-manifolds
- By the time this is computed, we should be *done*
- Use “vertex-triangle” adjacency instead
- Computed with 3 trivial linear passes

Simply output vertex adjacency lists



Tipsy (locally random) fans

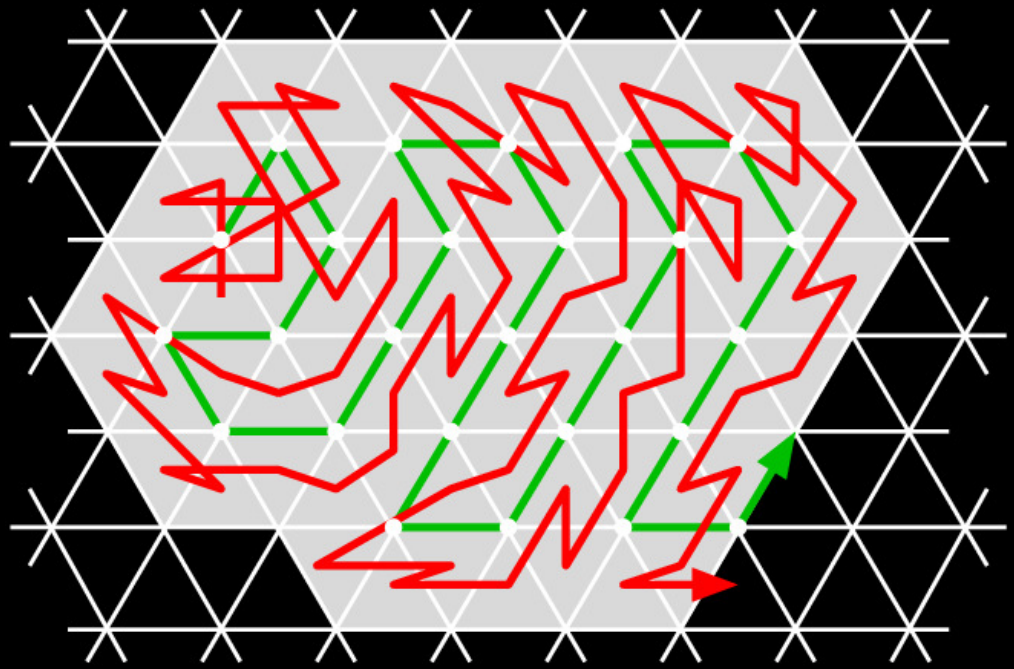
Choosing a better sequence



Tipsy strips

Selecting the next fanning vertex

- Must be a constant time operation
  - Select next vertex from 1-ring of previous
  - If none available, pick latest referenced
  - If none available, pick next in input order



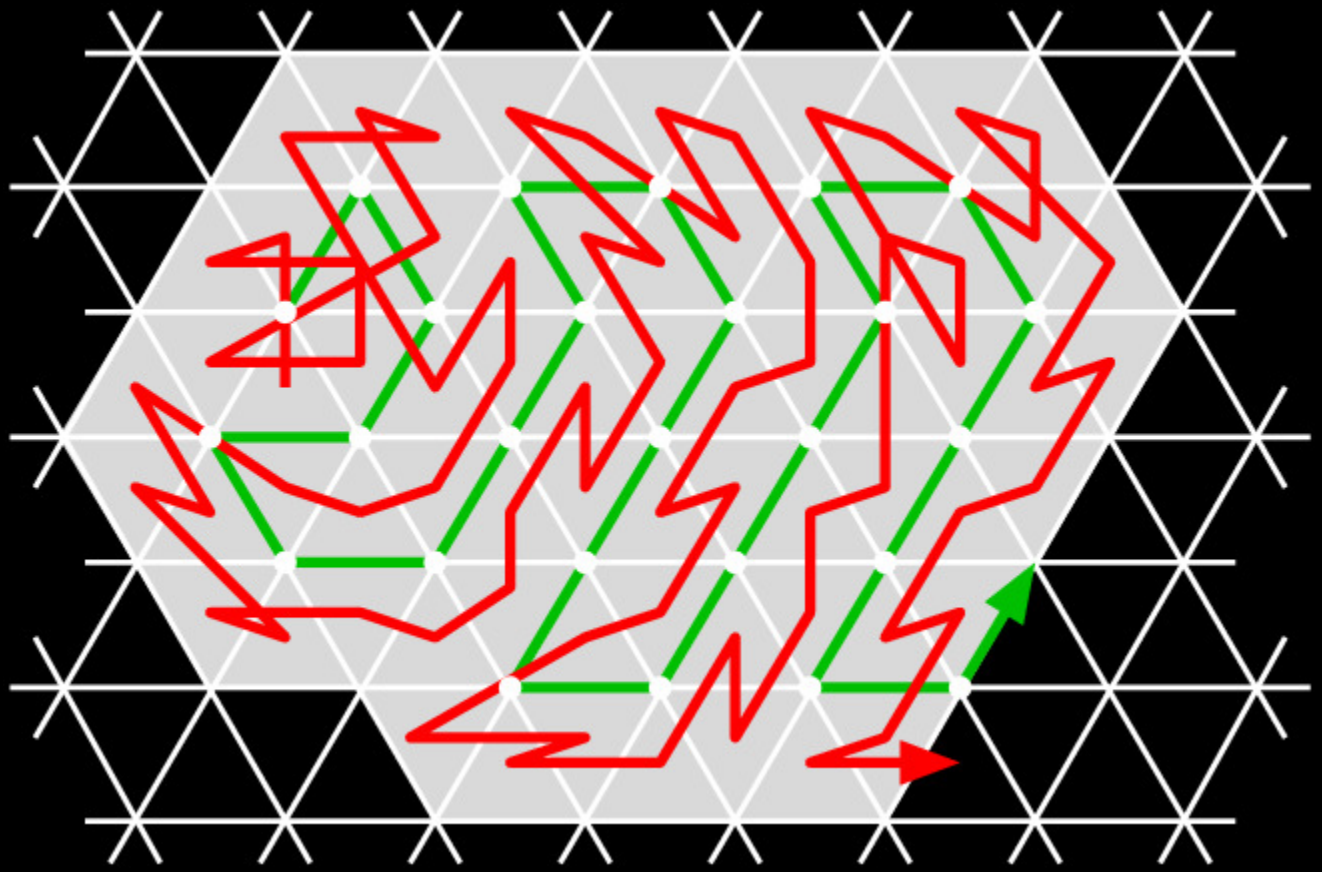
Best next fanning vertex within 1-ring

- Consider vertices referenced by emitted triangles

- Furthest in FIFO that would *remain* in cache

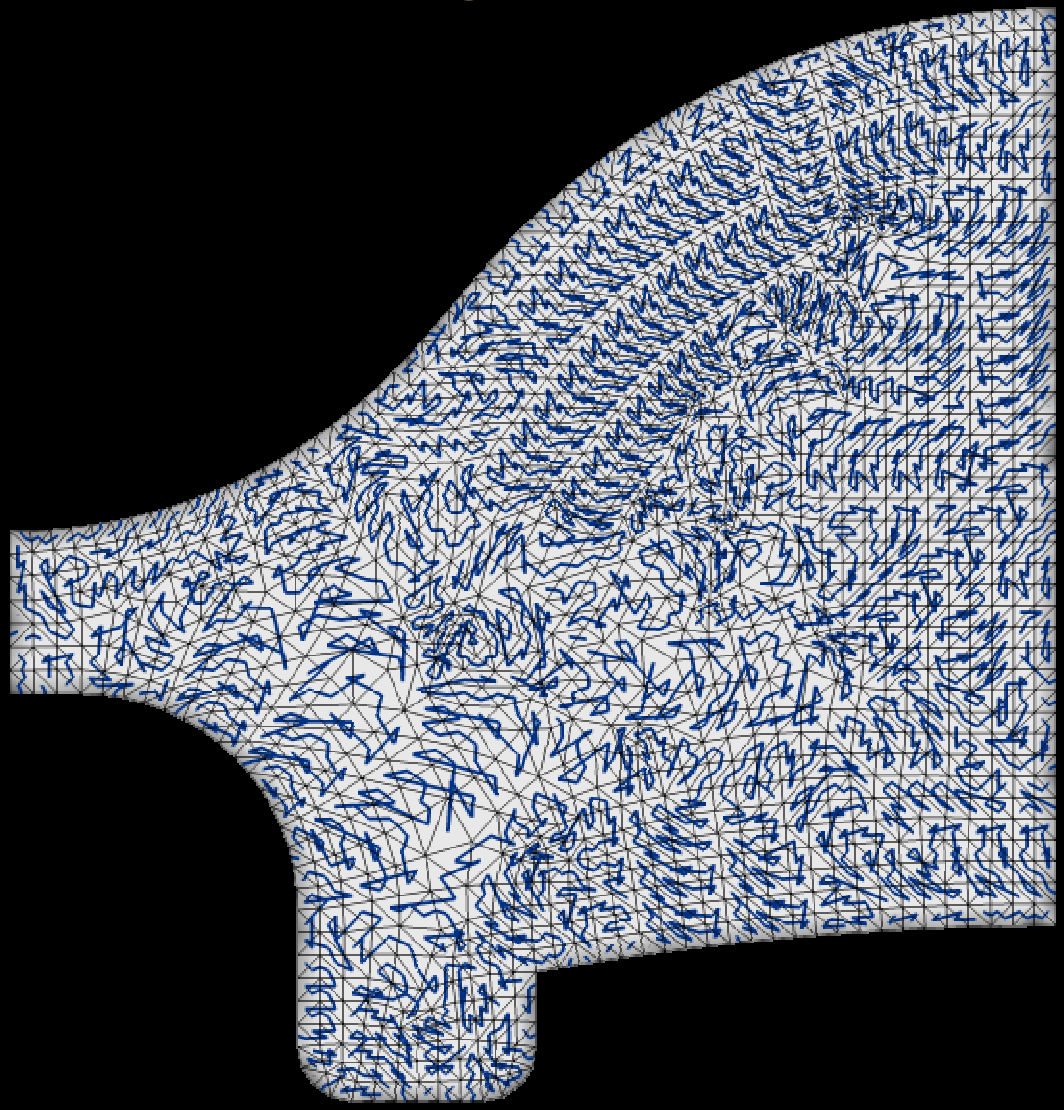
s	cache time stamp	how far is *u* in the cache?	$s - C[u]$
$C[u]$	caching time stamp of *u*	is *u* still in the cache?	$s - C[u] \leq k$
$L[u]$	# of live triangles in *u*	would *u* remain in cache?	$s - C[u] - 2L[u] \leq k$

Tipsy pattern



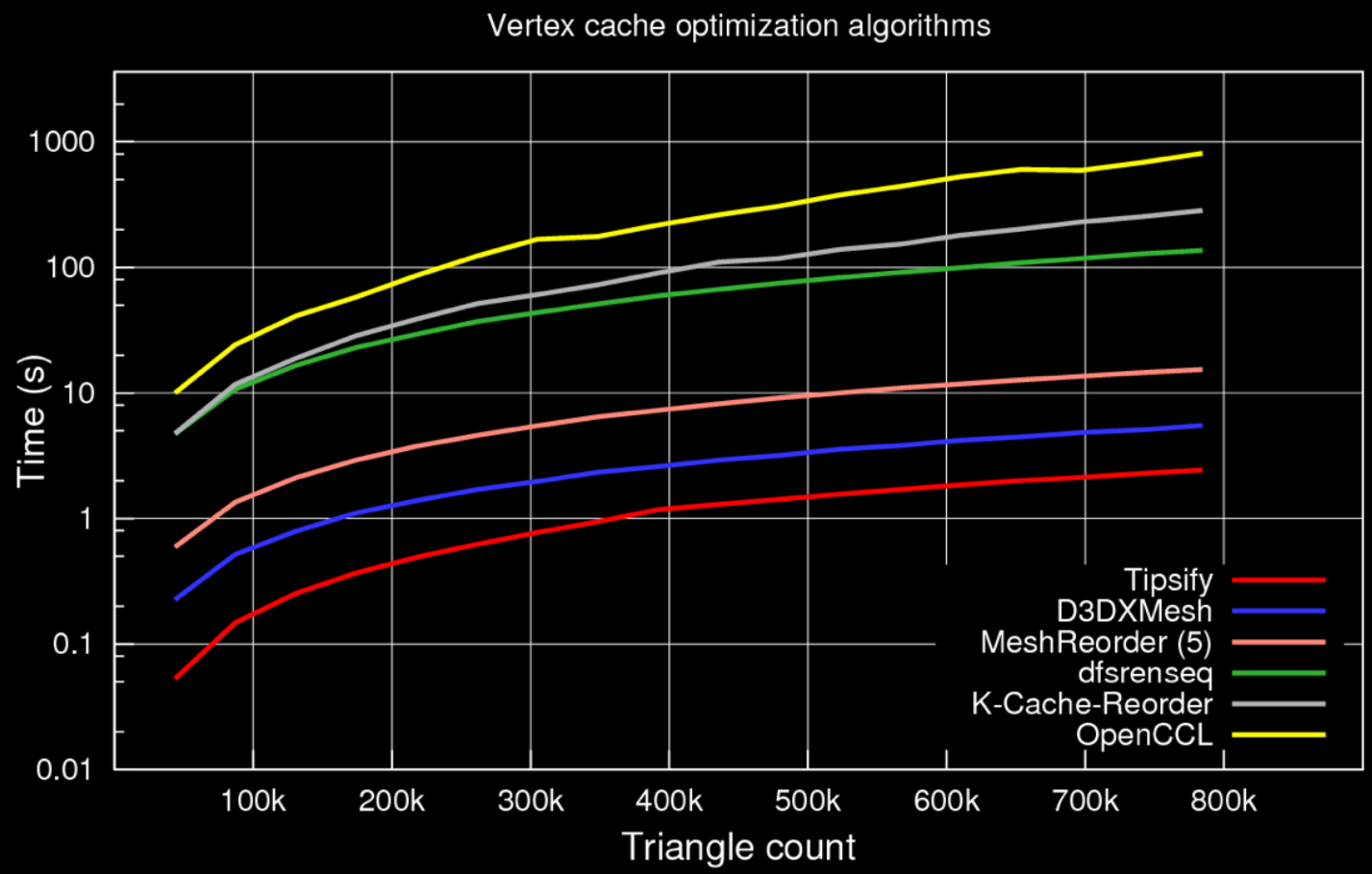
Tipsy strips

Tipsy pattern

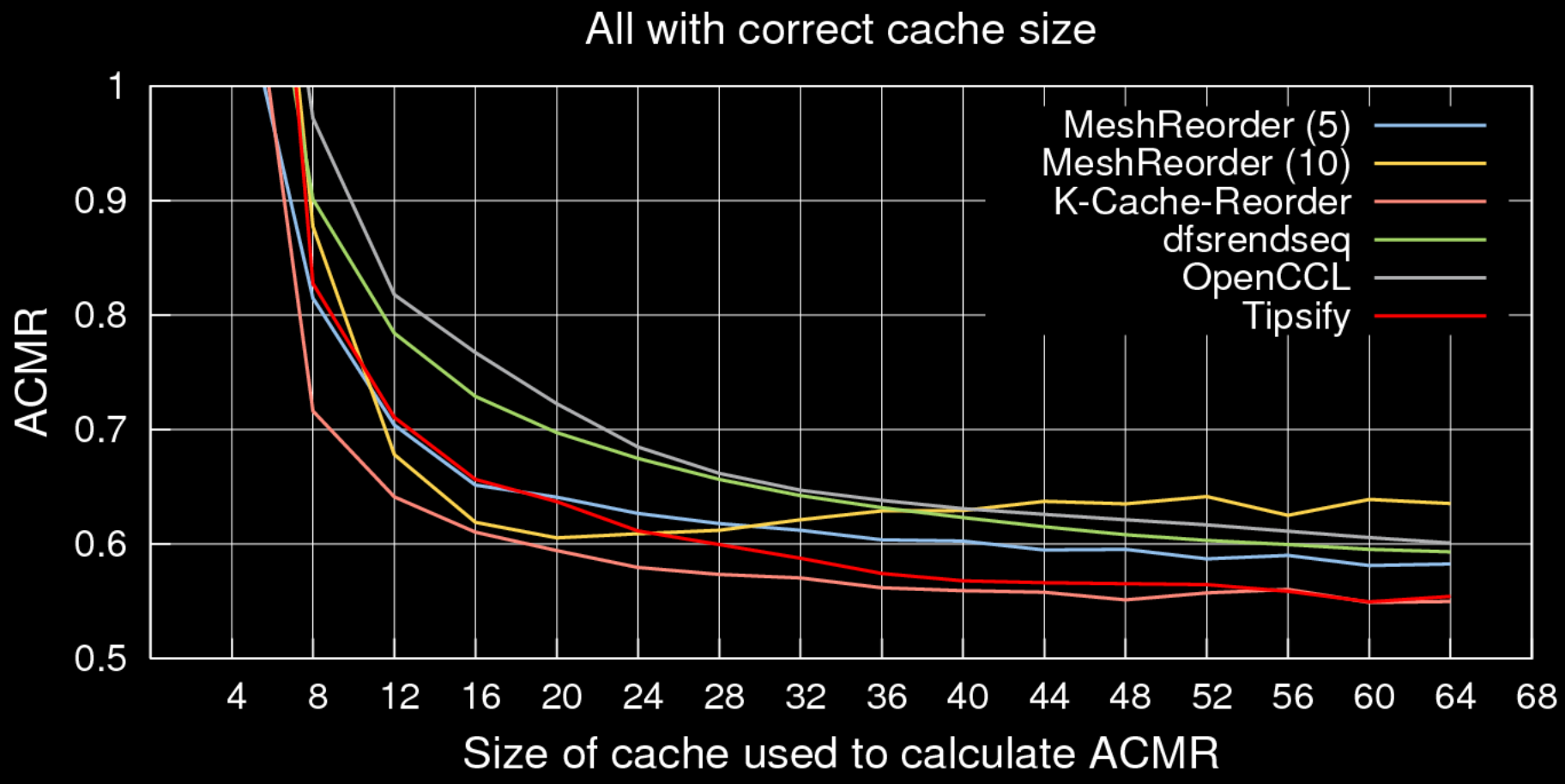


Tipsify

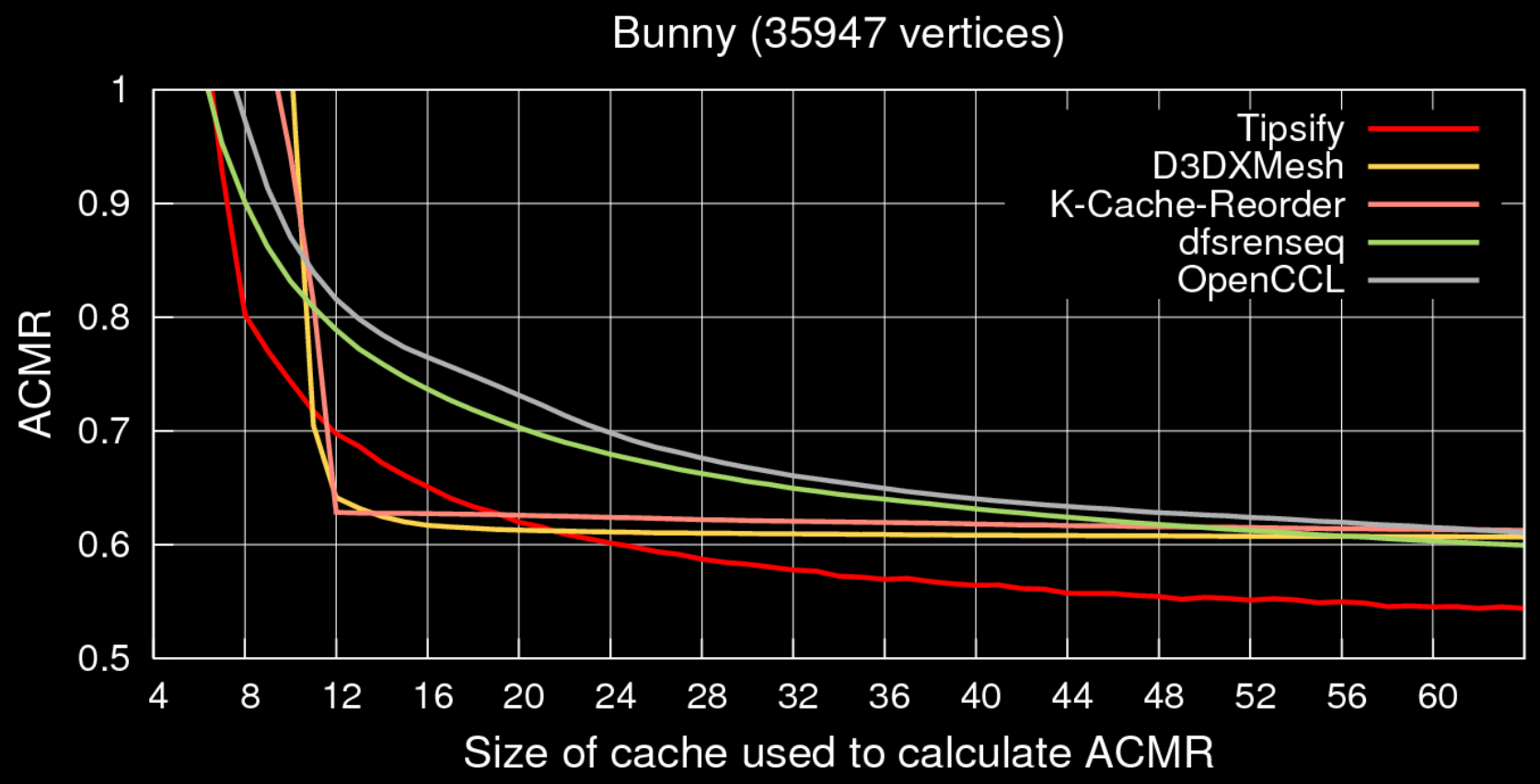
Typical running times



Preprocessing comparison

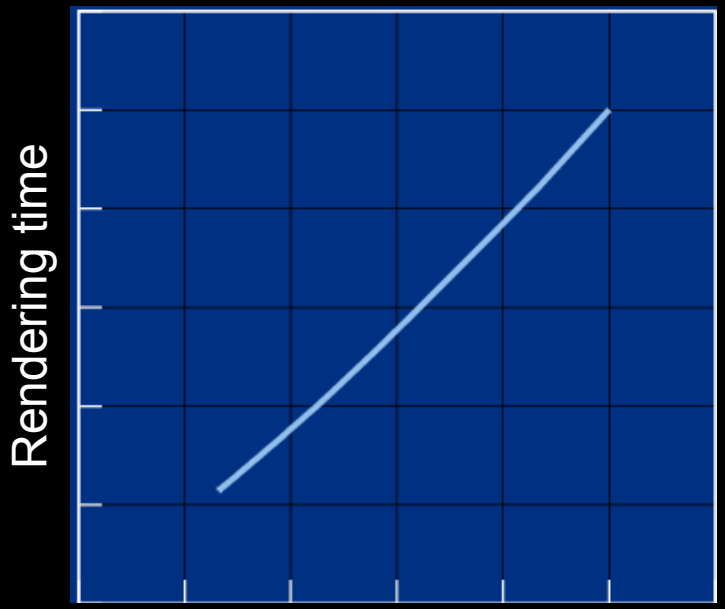


Typical ACMR comparison
Cache size of 12

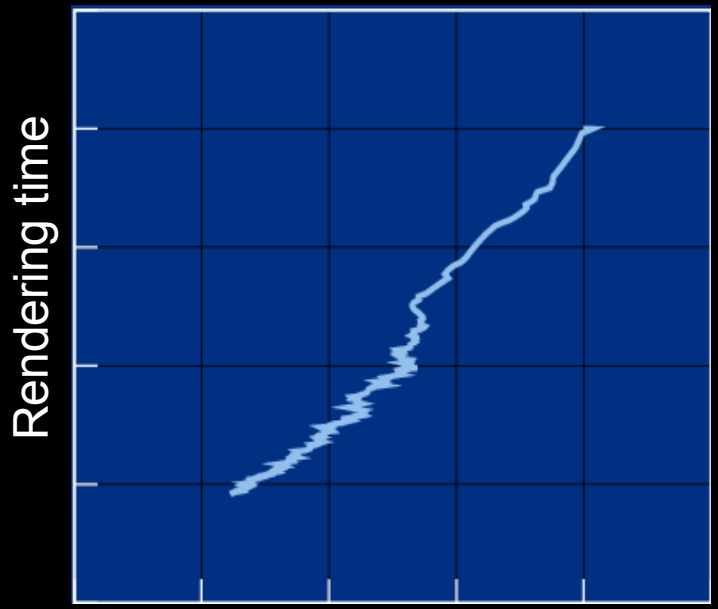


Motivation: Rendering time dependency

Vertex-bound scene



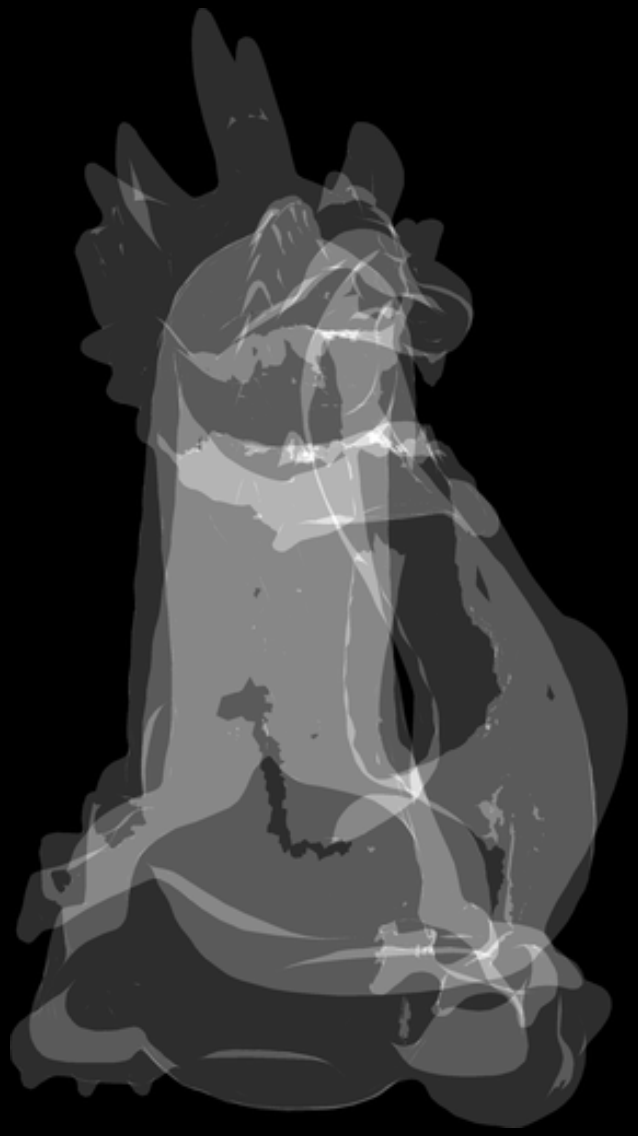
Pixel-bound scene



Reduce!
(transparently)

Part 2: Overdraw

- Expensive pixel shaders
- High overdraw
- Use early-z culling



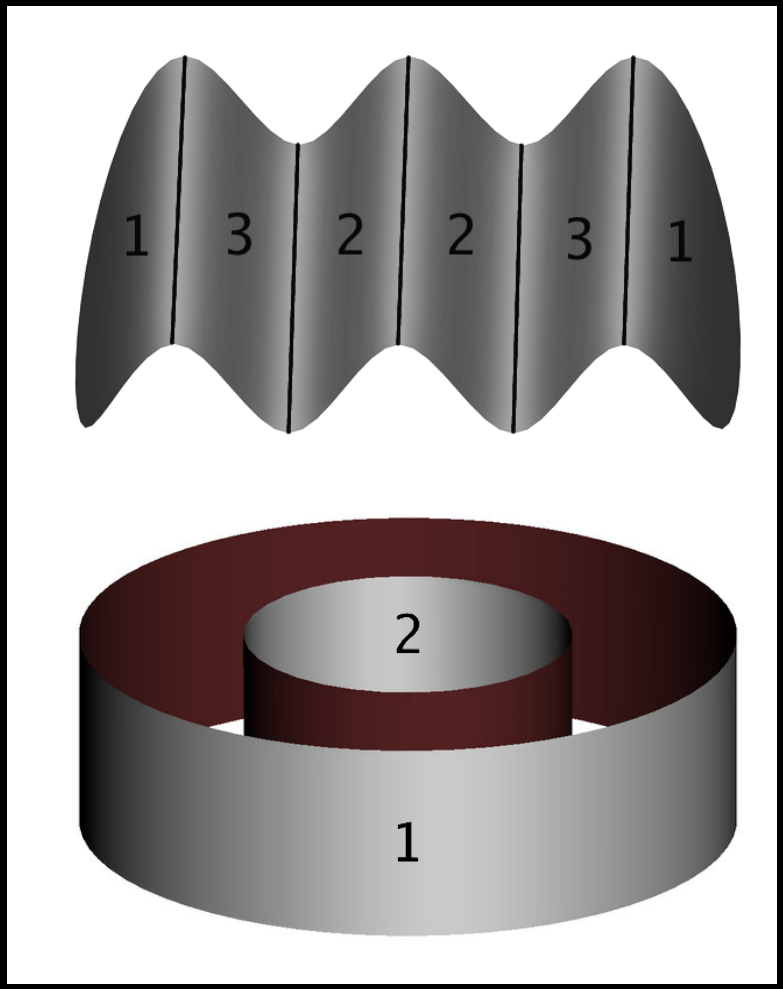
Options

- Dynamic depth-sort
  - Can be too expensive
  - Destroys mesh locality
- Z-buffer priming
  - Can be too expensive
- Sorting per object
  - E.g. GOVINDARAJU et al. 2005
  - Does not eliminate intra-object overdraw
  - Not transparent to application
  - Requires CPU work
  - Orthogonal method

Objective

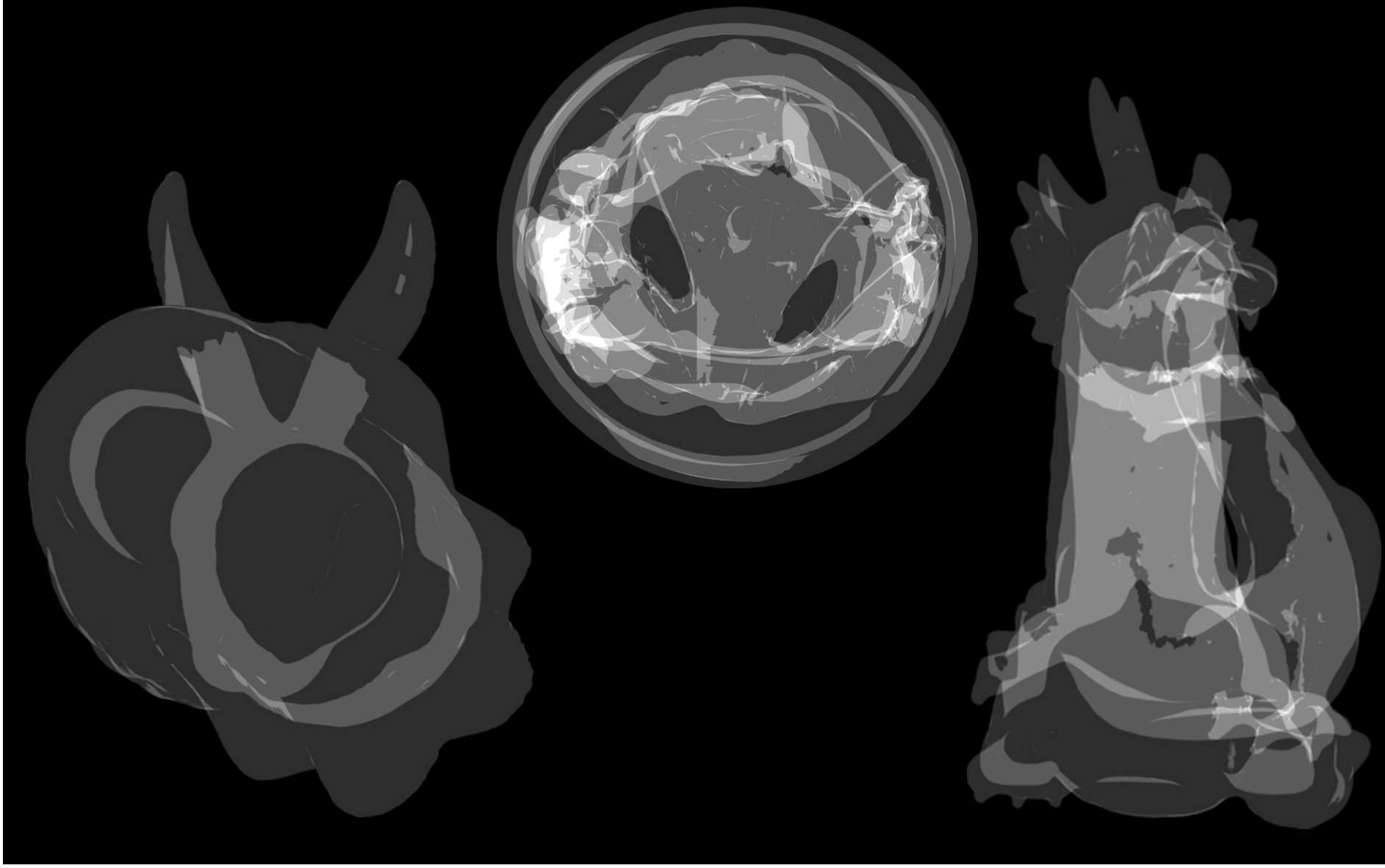
- Simple solution
- Single draw call
- Transparent to application
- Good in both vertex and pixel bound scenarios
- Fast to optimize

Insight: View Independent Ordering [Nehab et al. 06]

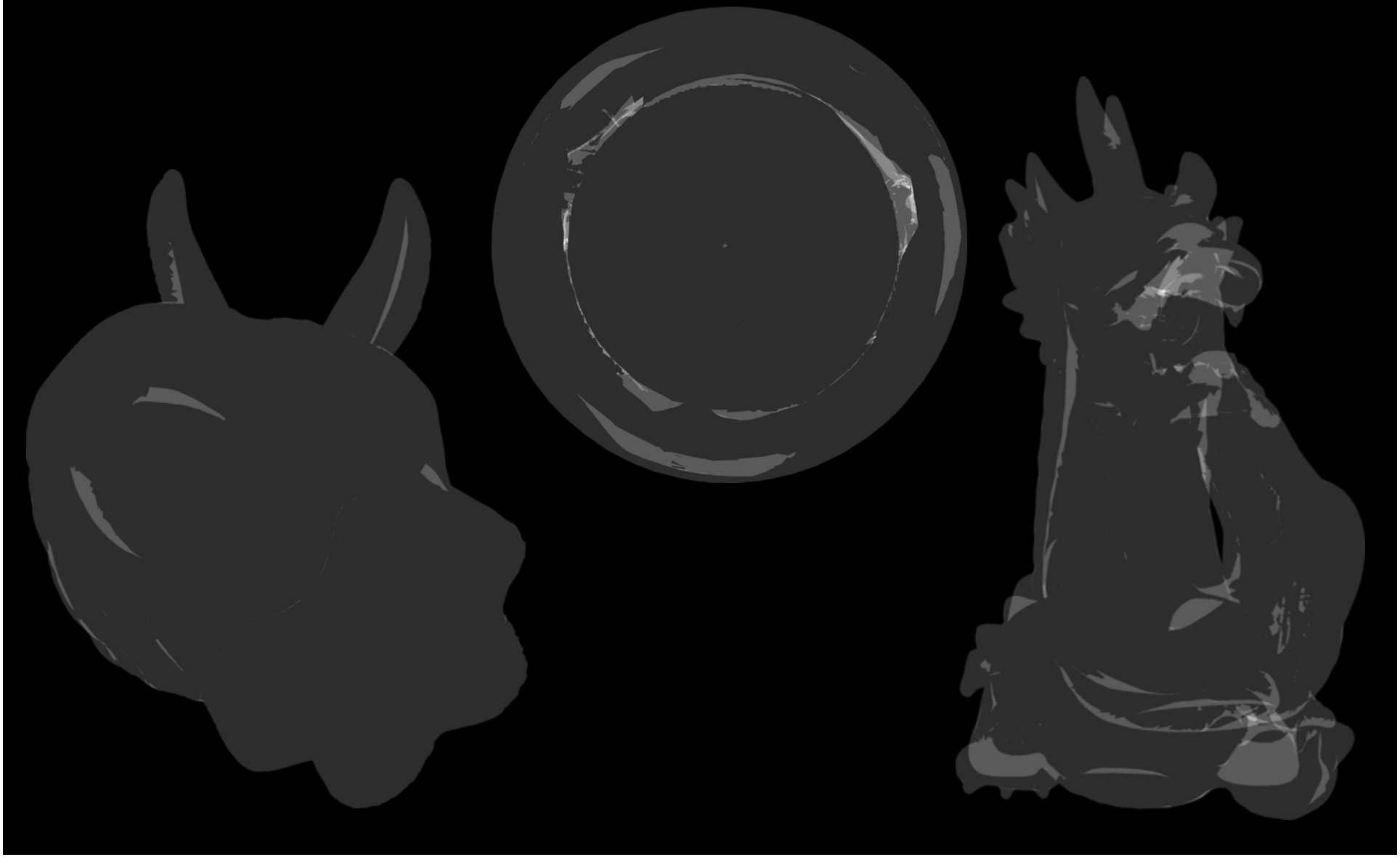


- Back-face culling is often used
- Convex objects have *no* overdraw, regardless of viewpoint
- Might be possible even for concave objects!

Overdraw (before)



Overdraw (after)



Our algorithm

- Can we do it at load-time or interactively?
- Yes! ☺ (order of milliseconds)
  - Quality on par with previous method
  - Can be immediately executed after vertex cache optimization (Part 1)
  - Like tipsy, operates on vertex and index buffers

Algorithm overview

1. Vertex cache optimization
   - Optimize for vertex cache ***first*** (Tipsify)
2. Linear clustering
   - Segment the index buffer into clusters
3. Overdraw sorting
   - Sort clusters to minimize overdraw

2. Linear clustering

During tipsy optimization:

- Maintaining the current ACMR
- Insert cluster boundary when:
  - A cache flush is detected
  - The ACMR reaches above a particular threshold λ

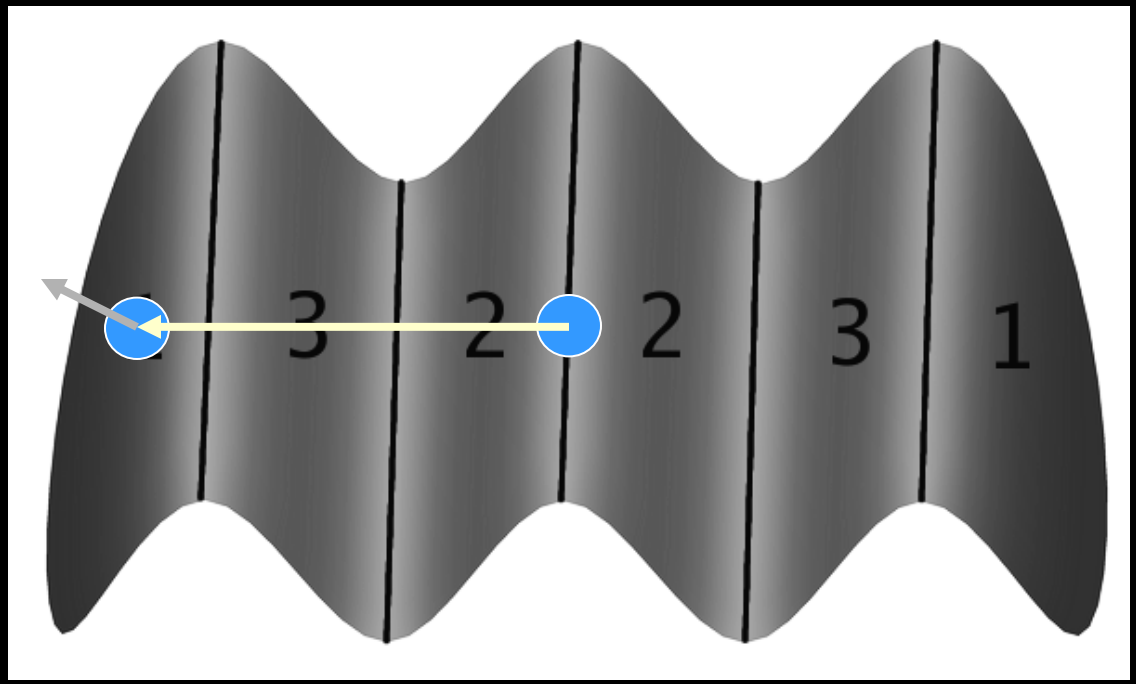
Threshold λ trades off
cache efficiency vs. ***overdraw***

If we care about both, use λ **= 0.75** on all meshes

- Good enough vertex cache gains
- More than enough clusters to reduce overdraw

3. Sorting: The DotRule

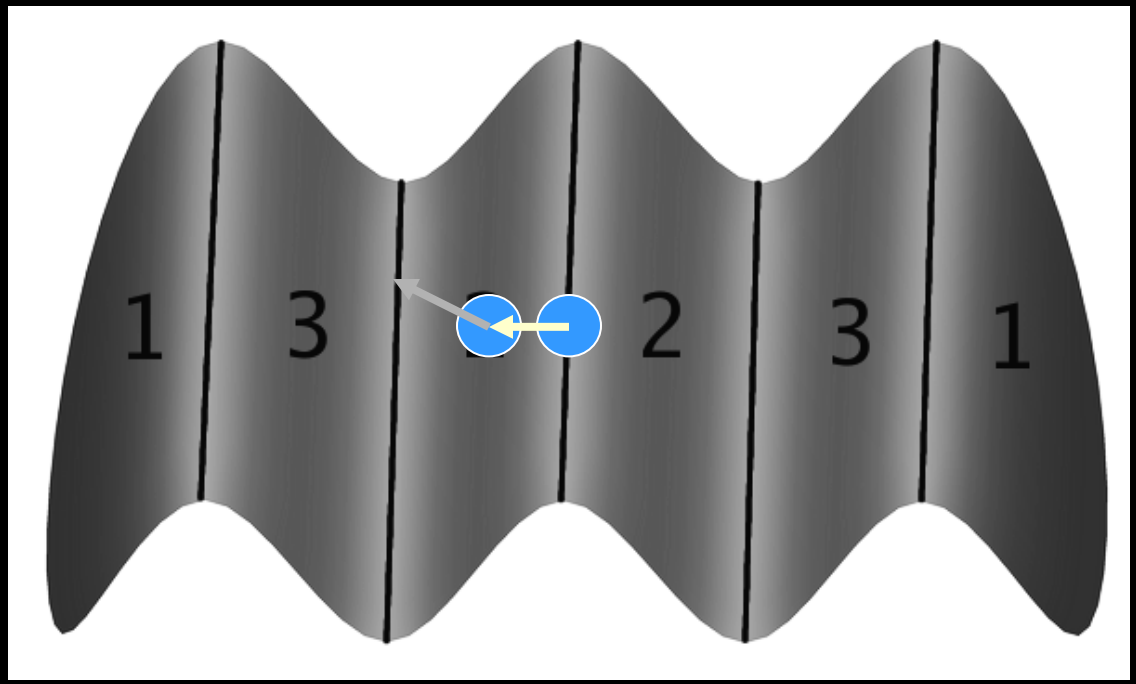
- How do we sort the clusters?
- Intuition: Clusters *facing out* have a higher *occluder potential*



$(C_p - M_p) \cdot C_n$

3. Sorting: The DotRule

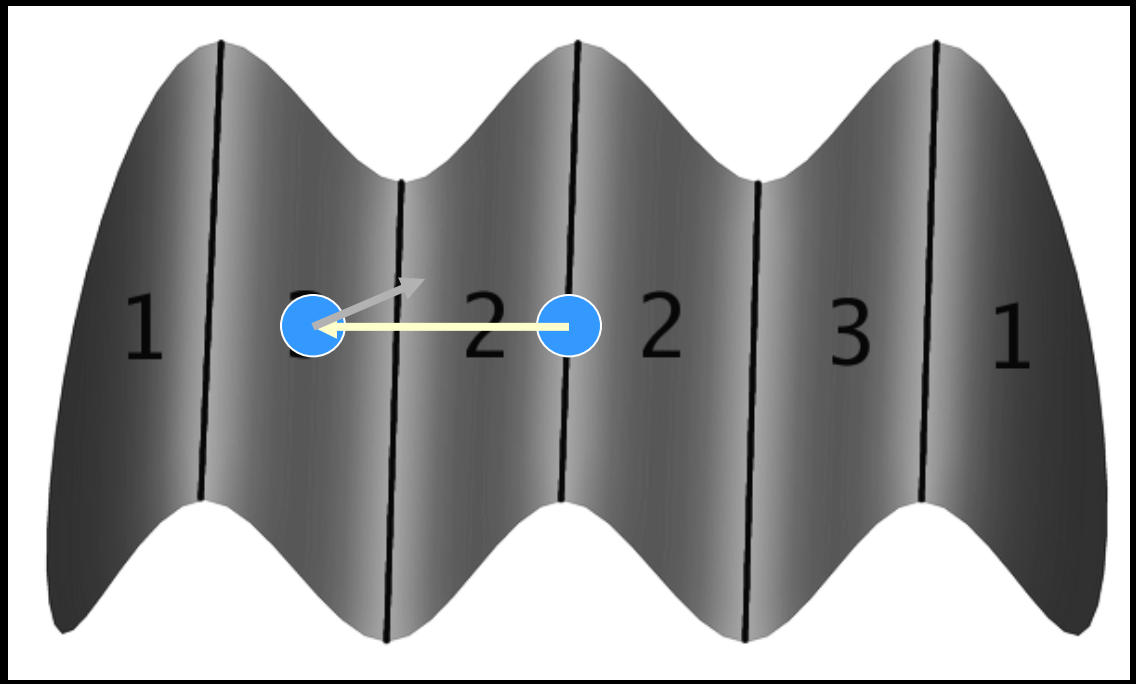
- How do we sort the clusters?
- Intuition: Clusters *facing out* have a higher *occluder potential*



$(C_p - M_p) \cdot C_n$

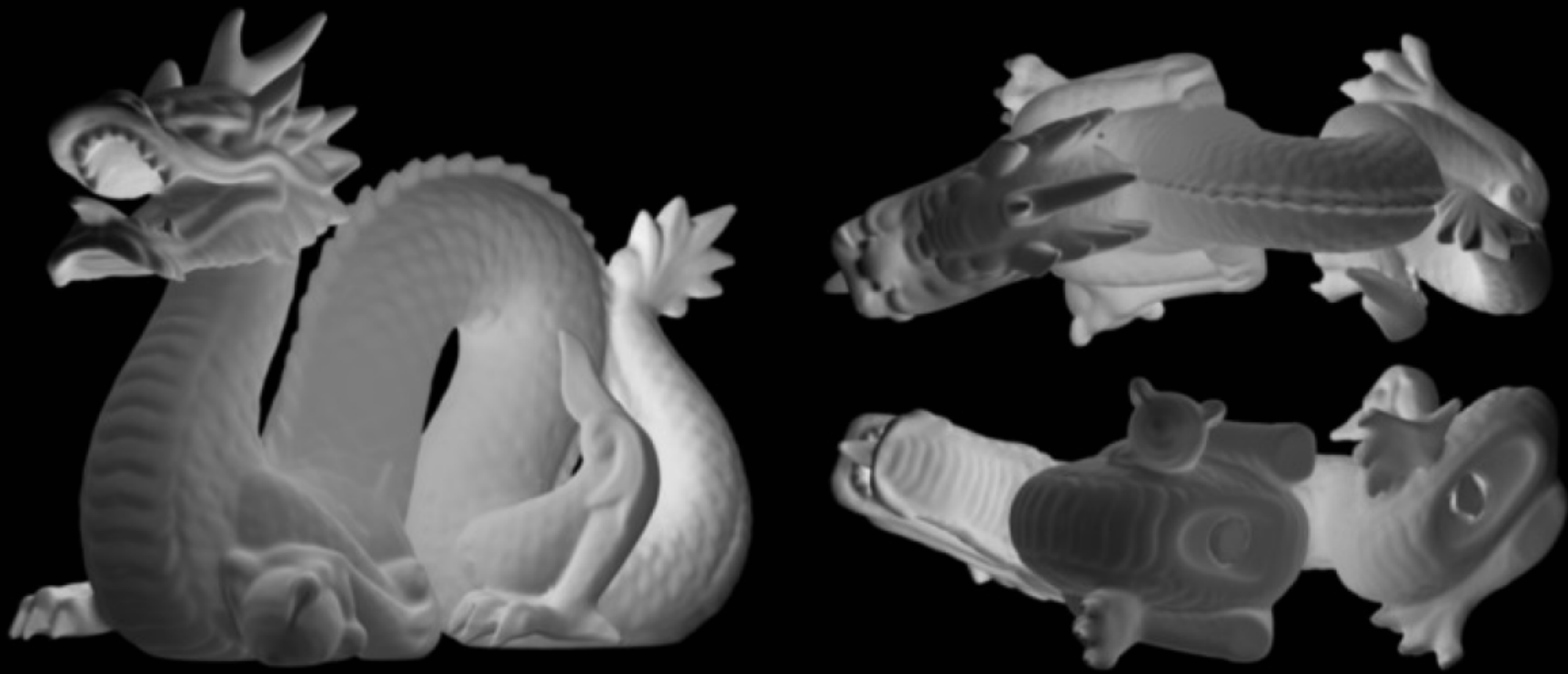
3. Sorting: The DotRule

- How do we sort the clusters?
- Intuition: Clusters *facing out* have a higher *occluder potential*

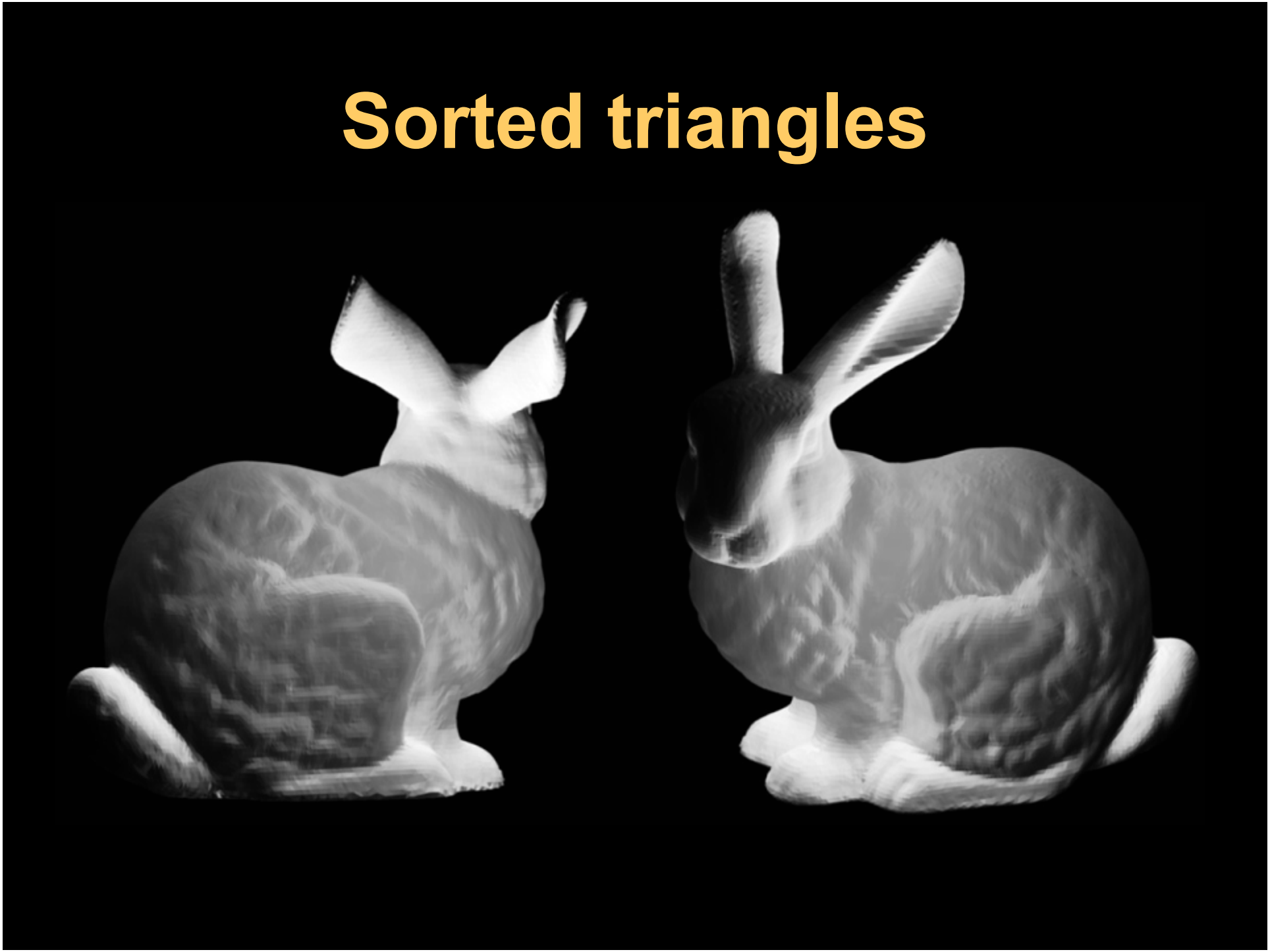


$(C_p - M_p) \cdot C_n$

Sorted triangles



Sorted triangles



Sorted clusters

Comparison to Nehab et al. 06

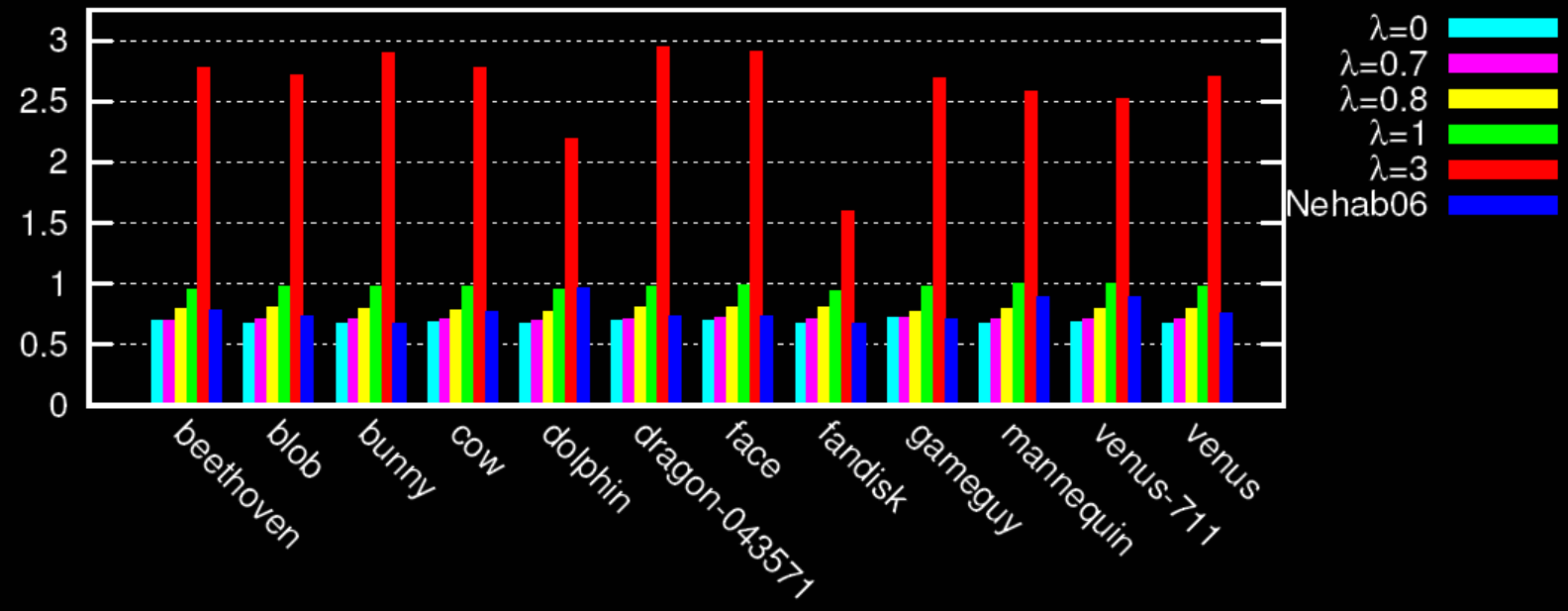
- We optimize for vertex cache first
- Allows for ***significantly*** more clusters
- Clusters not as planar, but we can afford more
- New heuristic to sort clusters very fast
- Tradeoff vertex vs. pixel processing at runtime

Timing comparisons

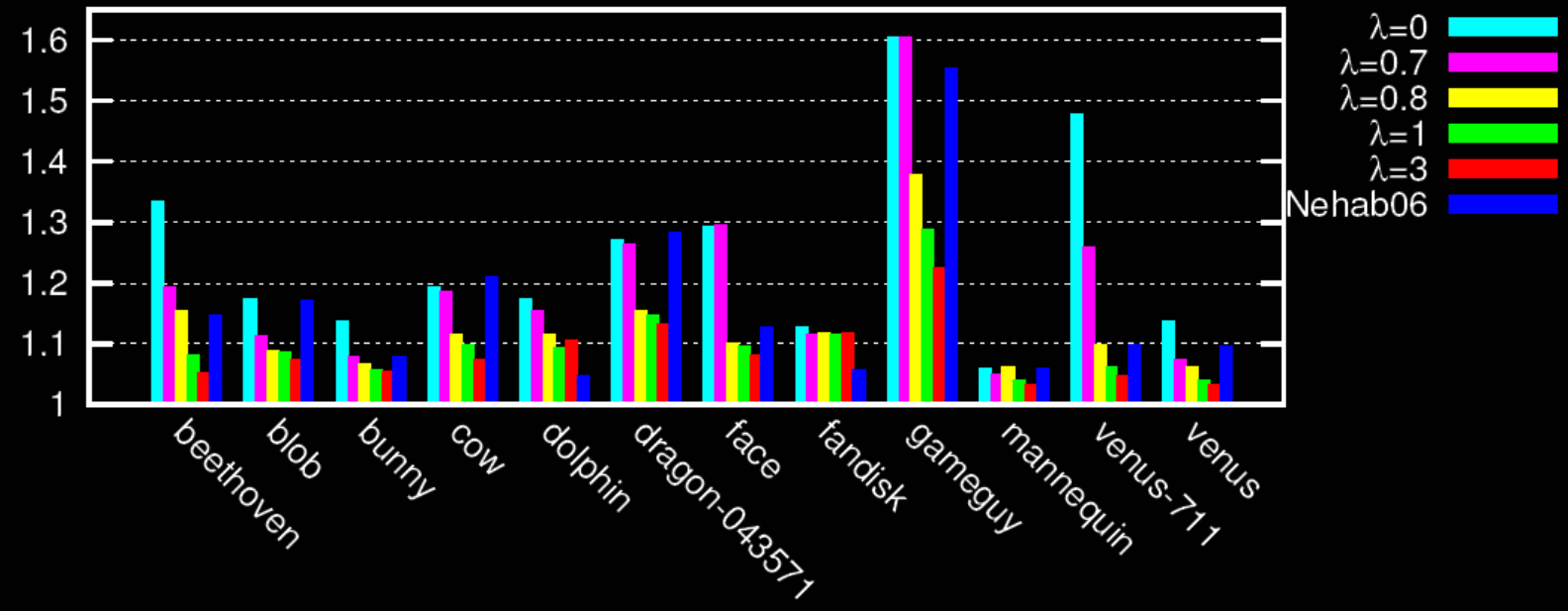
Mesh	Sander 07 (s)		New:Nehab 06
beethoven.m	0.0030		2712x
blob.m	0.0125		1359x
bunny.m	0.0749		321x
cow.m	0.0047		641x
dolphin.m	0.0003		10054x
dragon-043571.m	0.0434		253x
face.m	0.0199		251x
fandisk.m	0.0098		1024x
gameguy.m	0.0424		354x
mannequin.m	0.0007		13699x
venus.m	0.0033		1529x
Average			**3129x**

Overdraw comparison

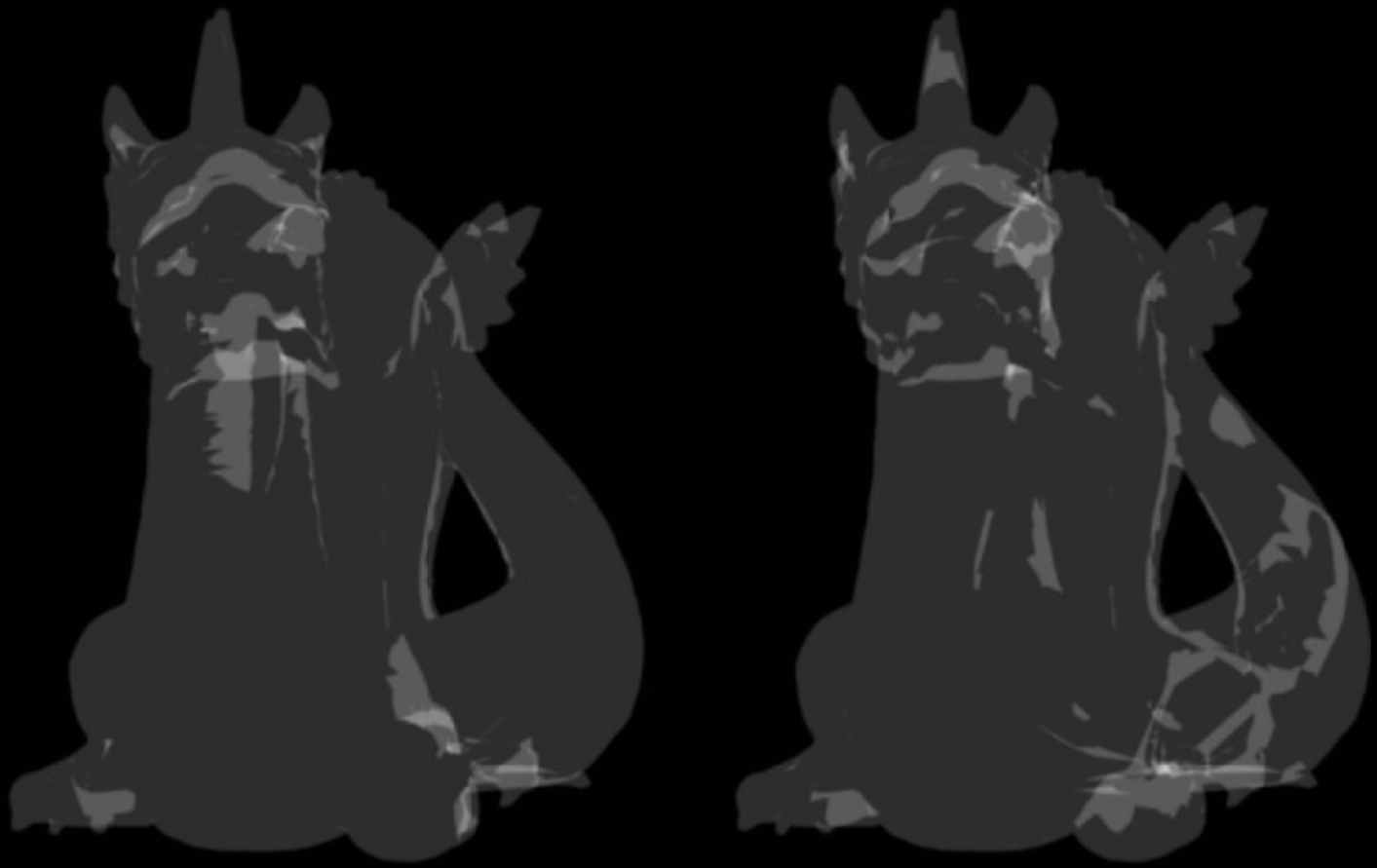
ACMR



MOVR



Comparison



Nehab et al. 06
40sec

Tipsy + DotRule
0.076sec

Summary

- Run-time vertex cache optimization
- Run-time overdraw reduction

- Operates on vertex and index buffers directly
- Works on non-manifolds
- Orders of magnitude faster
- Allows for varying cache sizes and animated models
- Quality comparable with previous methods
- About 500 lines of code!
- Extremely easy to incorporate in a rendering pipeline

- Expect most game rendering pipelines will incorporate such an algorithm
- Expect CAD applications to use and re-compute ordering interactively as geometry changes

```
void OverdrawOrder(int *piIndexBufferIn,
            int *piIndexBufferOut,
            int iNumFaces,
            float *pfVertexPositionsIn,
            int iNumVertices,
            int *piClustersIn,
            int iNumClusters,
            int *piScratch,
                  int *piRemap = NULL)
{
  int i, j;
  int c=0, cstart=0;
  int cnext=piClustersIn[1];
  int *p = piIndexBufferIn;
  Vector *pvVertexPositionsIn = (Vector *)pfVertexPositionsIn;
  Vector vMeshPositions = Vector(0,0,0);
  float fMArea = 0.f;

  int *piScratchBase = piScratch;
  Vector *pvClusterPositions = (Vector *)piScratch;
  piScratch += iNumClusters * 3;

  Vector *pvClusterNormals = (Vector *)piScratch;
  piScratch += iNumClusters * 3;

  ClusterSort *cs = (ClusterSort *)piScratch;
  piScratch += iNumClusters * 2;

  for(i = 0; i < iNumClusters; i++)
  {
    pvClusterPositions[i] = Vector(0,0,0);
    pvClusterNormals[i] = Vector(0,0,0);
  }
  float fCArea = 0.f;
  for(i = 0; i <= iNumFaces; i++)
  {
    if(i == cnext)
    {
      pvClusterPositions[c] /= fCArea * 3.f;
      pvClusterNormals[c].normalize();
      c++;
      if(c == iNumClusters)
        break;
      cstart = i;
      cnext = piClustersIn[c+1];
      fCArea = 0.f;
    }

    Vector vNormal = cross(pvVertexPositionsIn[p[2]] -
                           pvVertexPositionsIn[p[0]],
                           pvVertexPositionsIn[p[1]] -
                           pvVertexPositionsIn[p[0]]);
    float fArea = vNormal.length();
    if(fArea > 0.f)
    {
      vNormal /= fArea;
    }
    else
    {
      fArea = 0.f;
      vNormal = Vector(0,0,0);
    }

    for(j = 0; j < 3; j++)
    {
      Vector *vp = (Vector *)&pfVertexPositionsIn[(*p) * 3];
      vMeshPositions += *vp * fArea;
      pvClusterPositions[c] += *vp * fArea;
      p++;
    }
    pvClusterNormals[c] += vNormal;

    fMArea += fArea;
    fCArea += fArea;
  }
  vMeshPositions /= fMArea * 3.f;

  for(i = 0; i < iNumClusters; i++)
  {
    cs[i].dp = dot(pvClusterPositions[i]-vMeshPositions,
                   pvClusterNormals[i]);
    if(cs[i].dp < -2e20 || cs[i].dp > 2e20)
    {
      cs[i].dp = 0.f;
    }
    cs[i].i = i;
  }

  std::sort(cs, cs+iNumClusters, sortfunc);

  int jj=0;
  for(i = 0; i < iNumClusters; i++)
  {
    for(j = piClustersIn[cs[i].i]*3; j < piClustersIn[cs[i].i+1]*3; j++)
      piIndexBufferOut[jj++] = piIndexBufferIn[j];
  }

  if(piRemap != NULL)
  {
     for(i = 0; i < iNumClusters; i++)
     {
        piRemap[i] = cs[i].i;
     }
  }

  memset(piScratchBase, 0, (piScratch - piScratchBase) * sizeof(int));
}
```

Summary

- Run-time triangle order optimization
- Run-time overdraw reduction

- Operates on vertex and index buffers directly
- Works on non-manifolds
- Allows for varying cache sizes and animated models
- Orders of magnitude faster
- Quality comparable with state of the art
- About 500 lines of code!
- Extremely easy to incorporate in a rendering pipeline

- Hope game rendering pipelines will incorporate such an algorithm
- Hope CAD applications to use and re-compute ordering interactively as geometry changes

Thanks

- Phil Rogers, AMD
- 3D Application Research Group, AMD

?