CSC 433/533
Computer Graphics

Alon Efrat
Thanks: Joshua Levine

Wrapping up distributed Ray Tracing

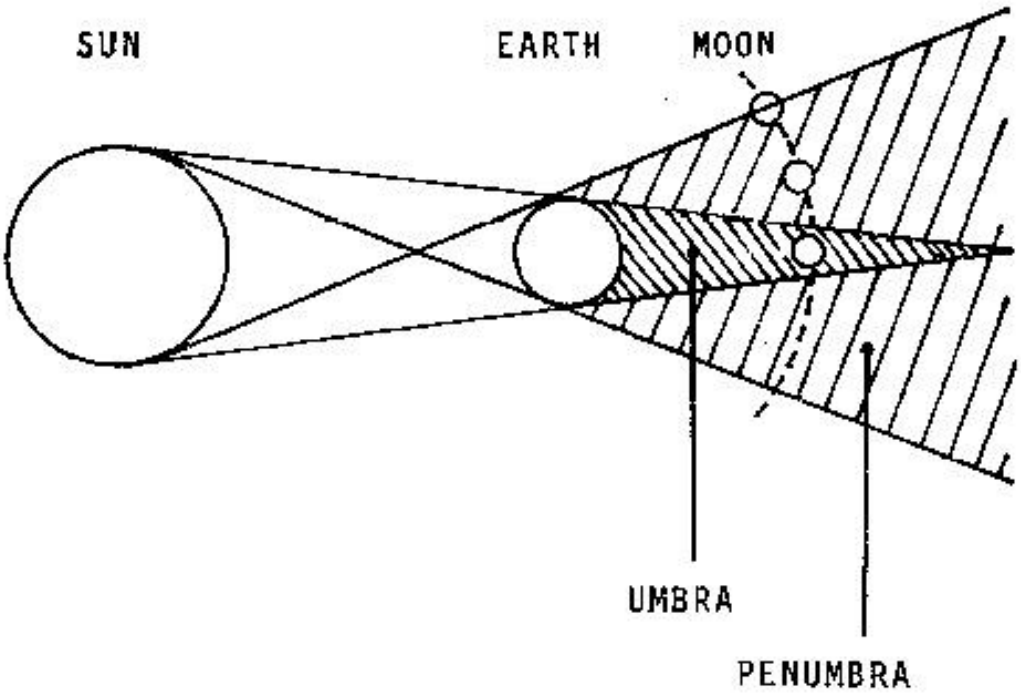
Triangle Meshes

Today’s Agenda

- Goals for today:
  - Finish distributed ray shooting
  - Discuss how complex shapes are stored and rendered

Soft Shadows

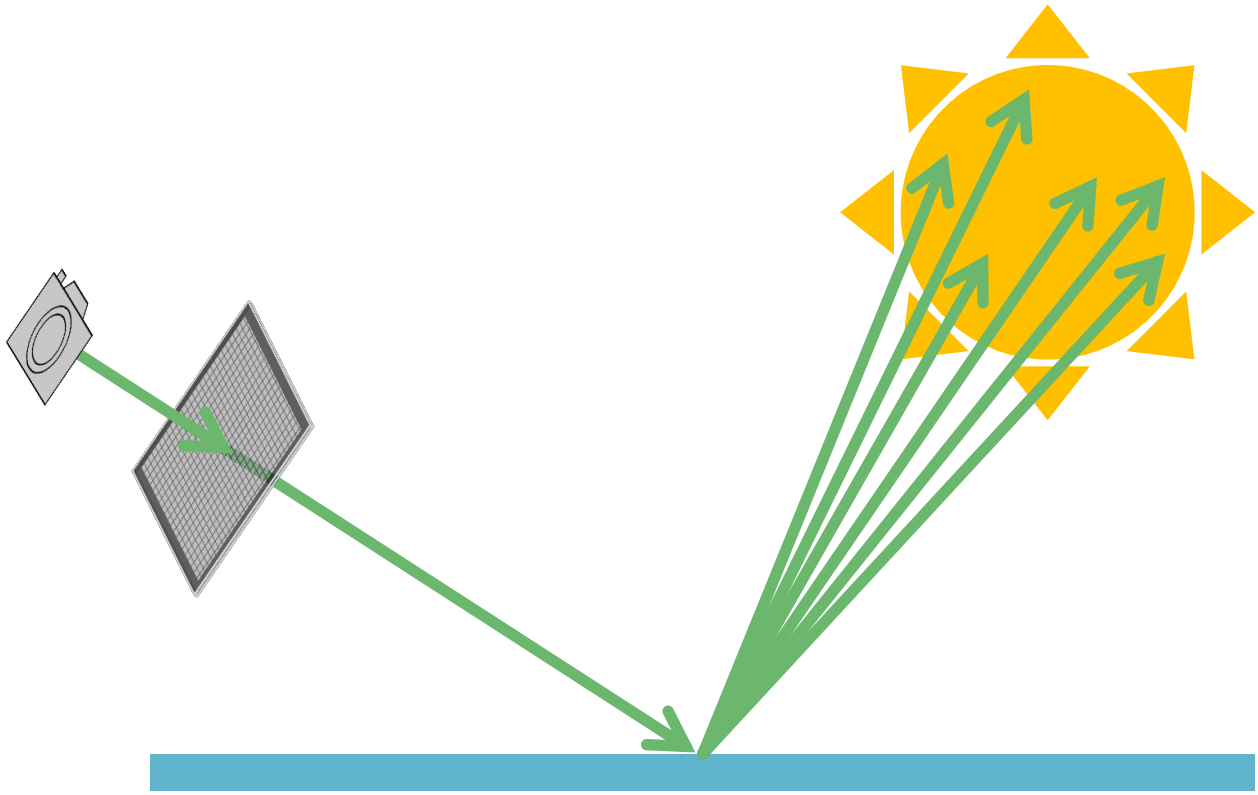
What Causes Soft Shadows



http://user.online.be/felixverbelen/lunecl.jpg

Lights aren't all point sources

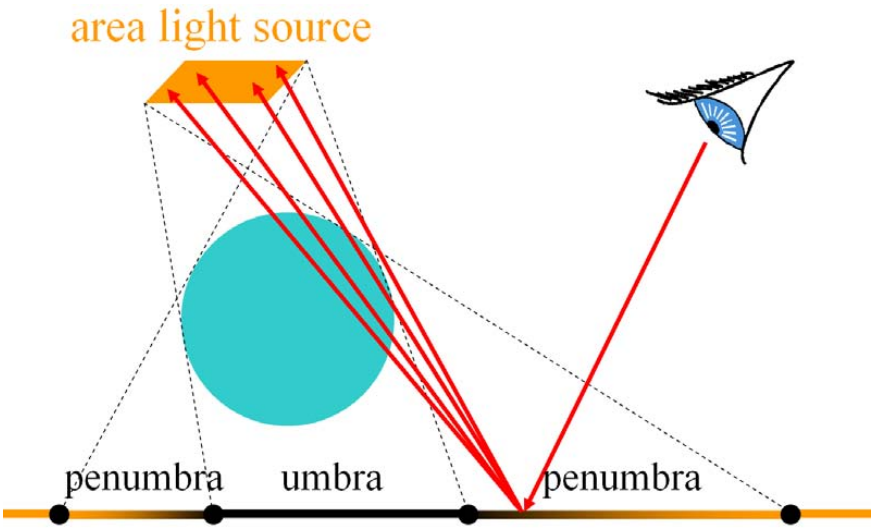
Distribution Soft Shadows

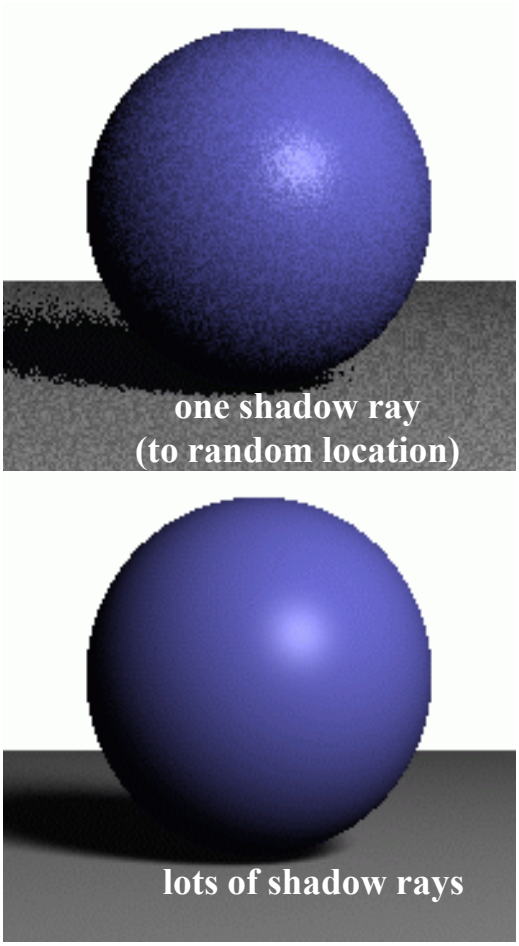


Randomly sample light rays

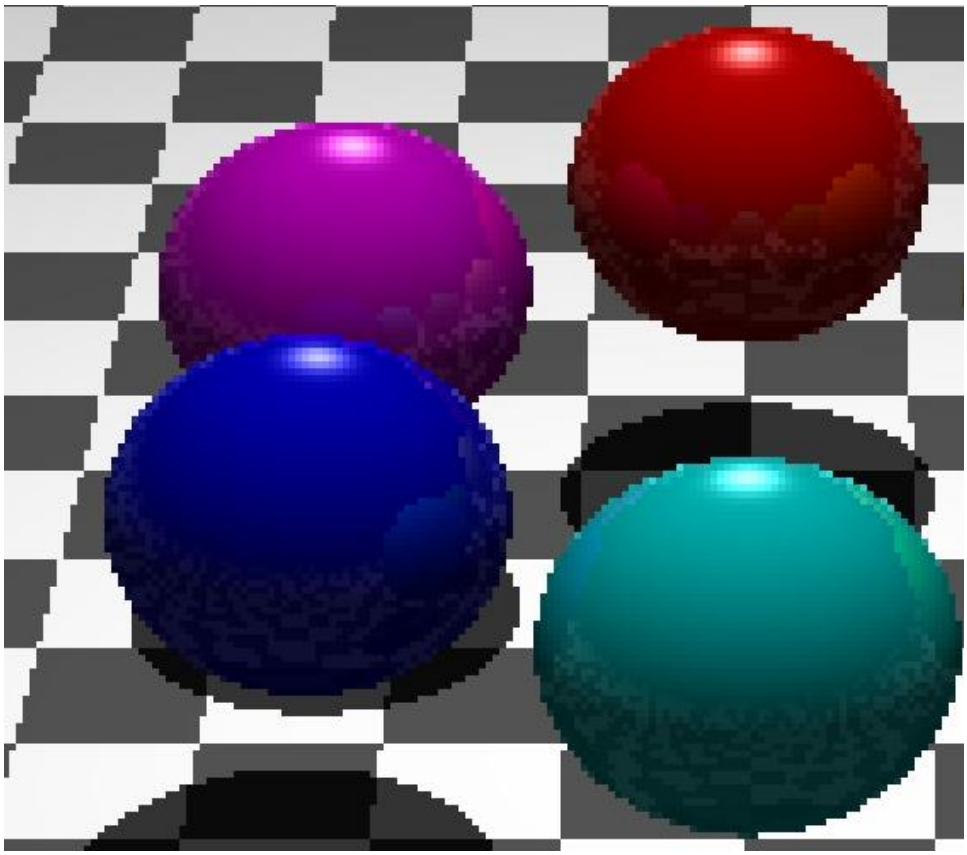
Computing Soft Shadows

- Model light sources as spanning an area
- Sample random positions on area light source and average rays





Problem: Aliasing

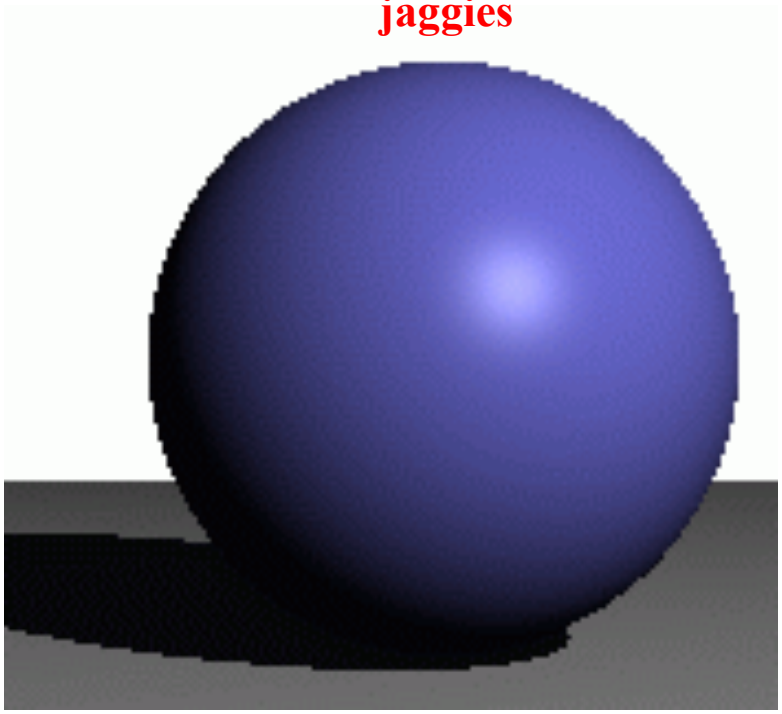


http://www.hackification.com/2008/08/31/experiments-in-ray-tracing-part-8-anti-aliasing/

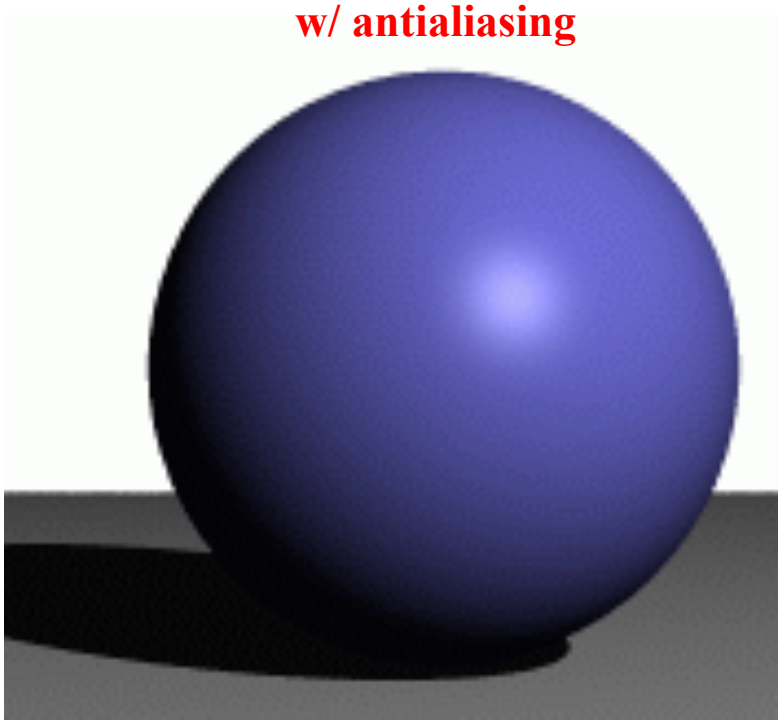
Antialiasing w/ Supersampling

- Cast multiple rays per pixel, average result

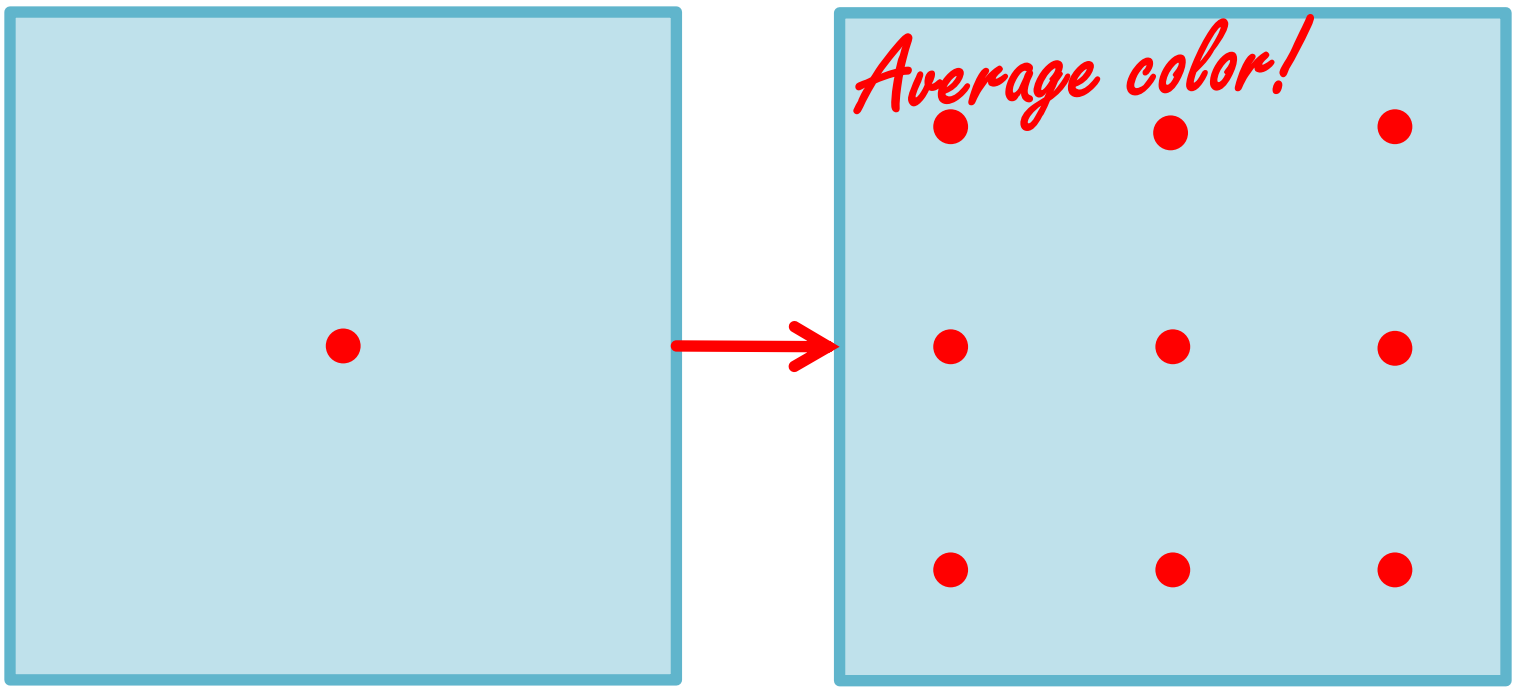
jaggies



w/ antialiasing

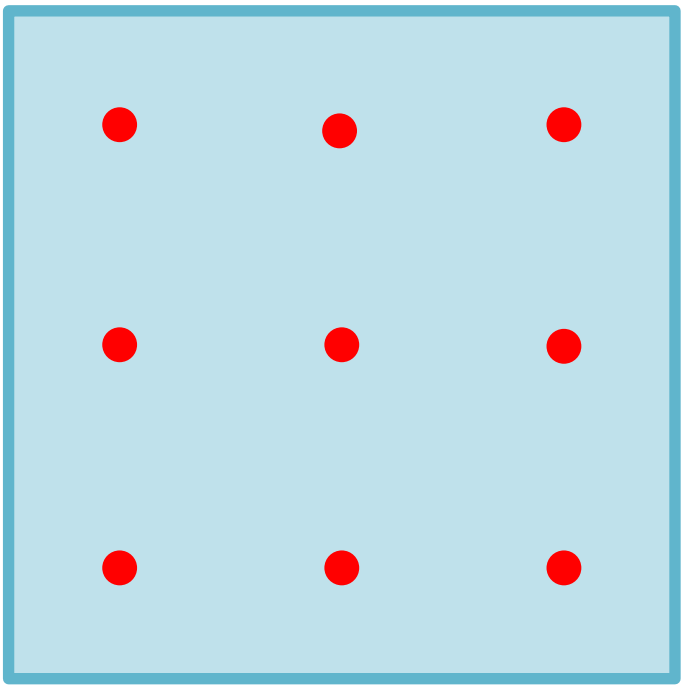


Distribution Antialiasing



Multiple rays per pixel

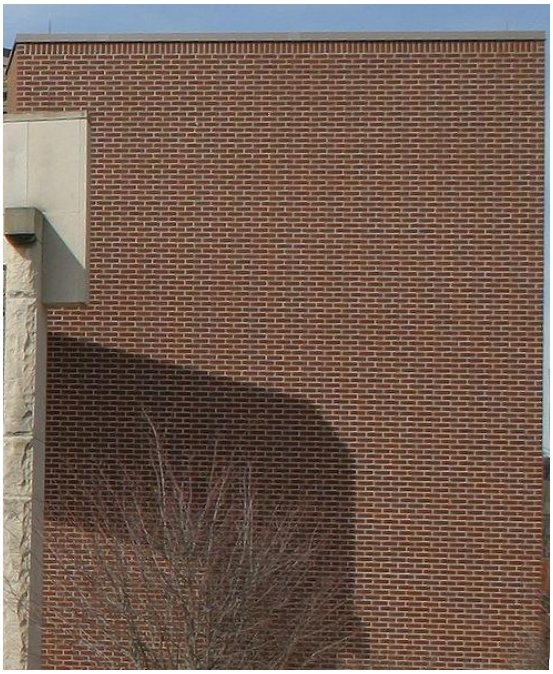
Distribution Antialiasing w/ Regular Sampling

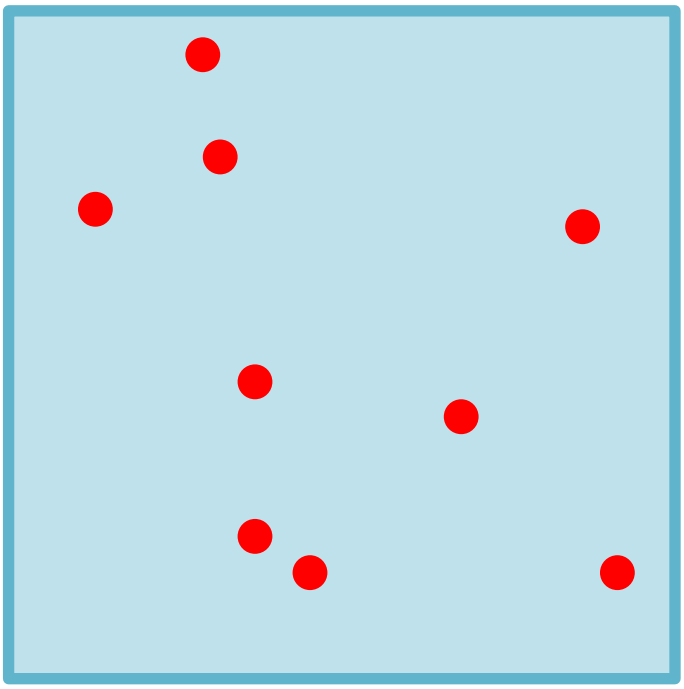


http://upload.wikimedia.org/wikipedia/commons/f/fb/Moire_pattern_of_bricks_small.jpg

Multiple rays per pixel

Distribution Antialiasing w/ Random Sampling

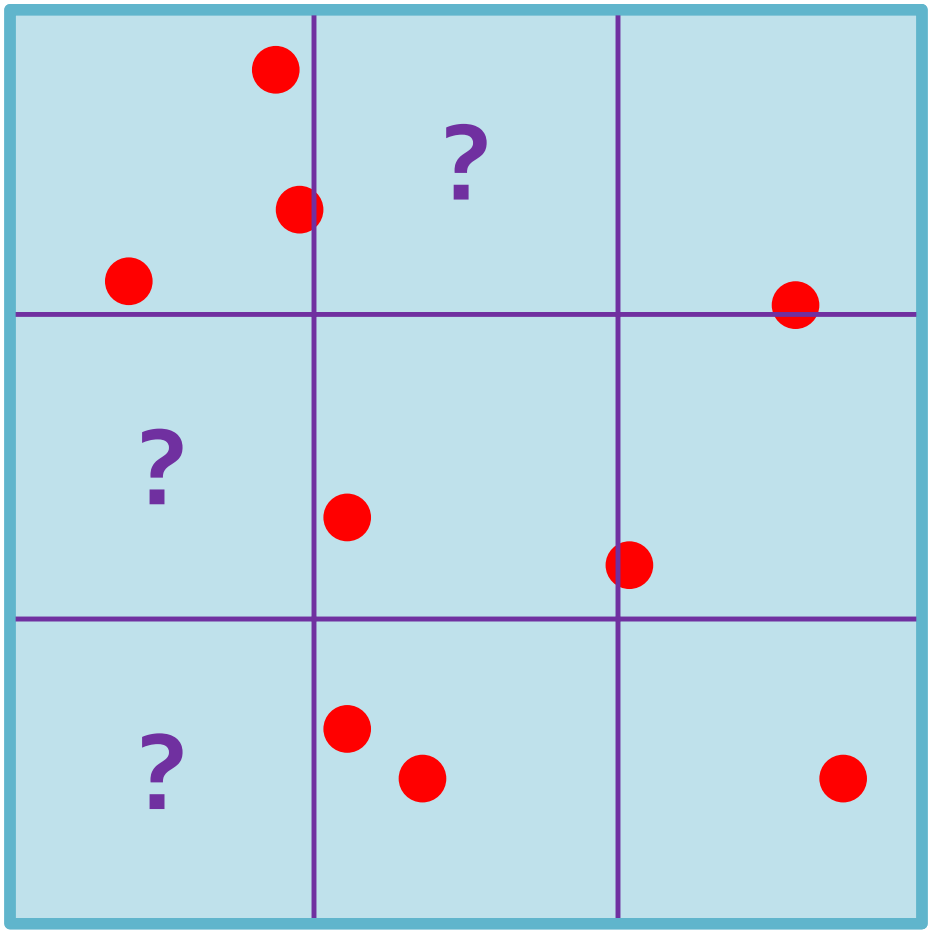




http://en.wikipedia.org/wiki/File:Moire_pattern_of_bricks.jpg

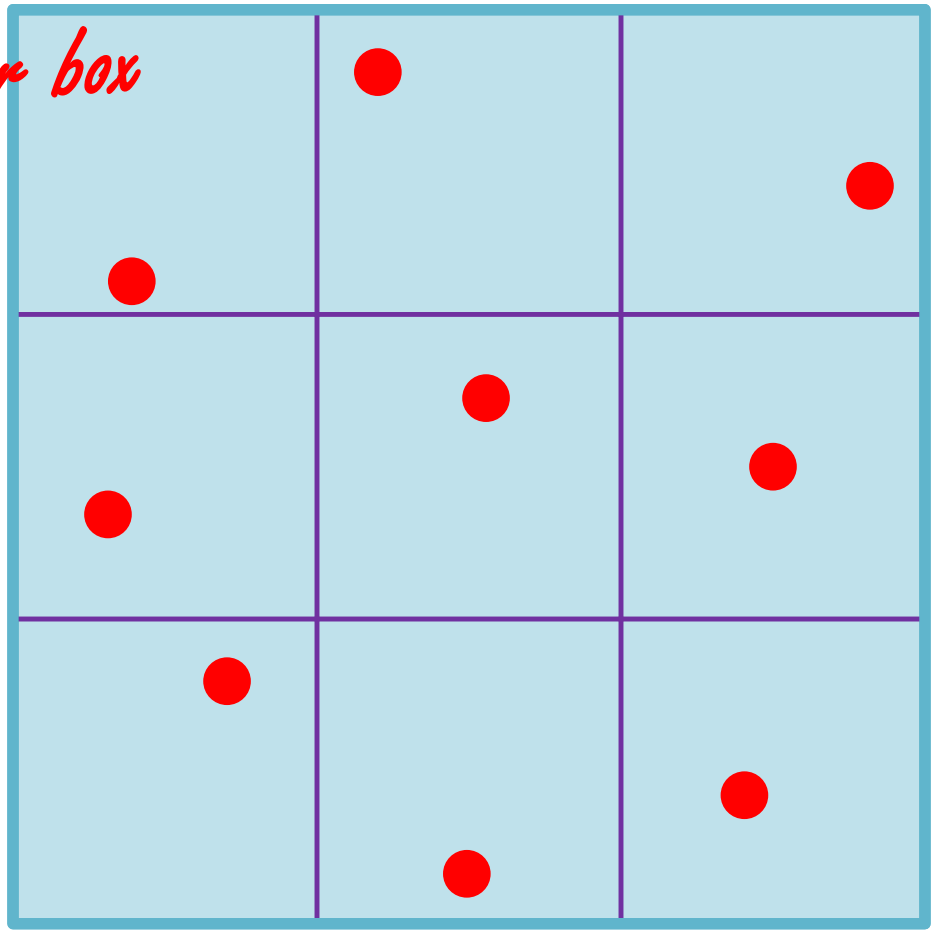
Remove Moiré patterns

Random Sampling Could Miss Regions Without Enough Sampling



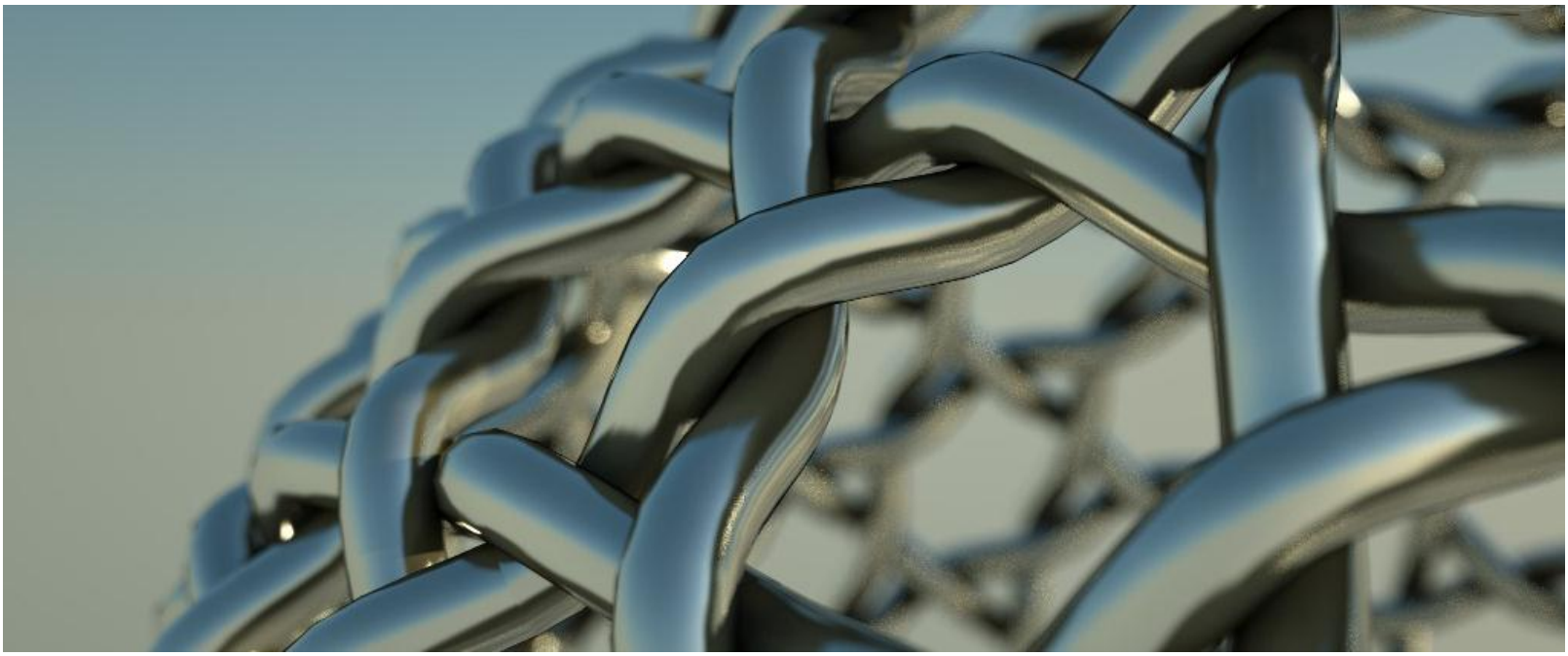
Stratified (Jittered) Sampling

One ray per box



Problem: Focus
Real Lenses Have Depth of Field

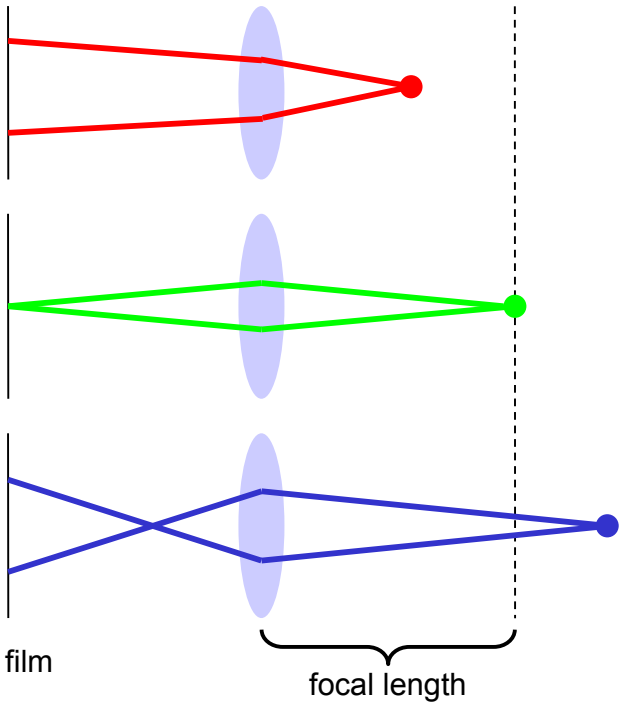
Problem: Focus
Real Lenses Have Depth of Field

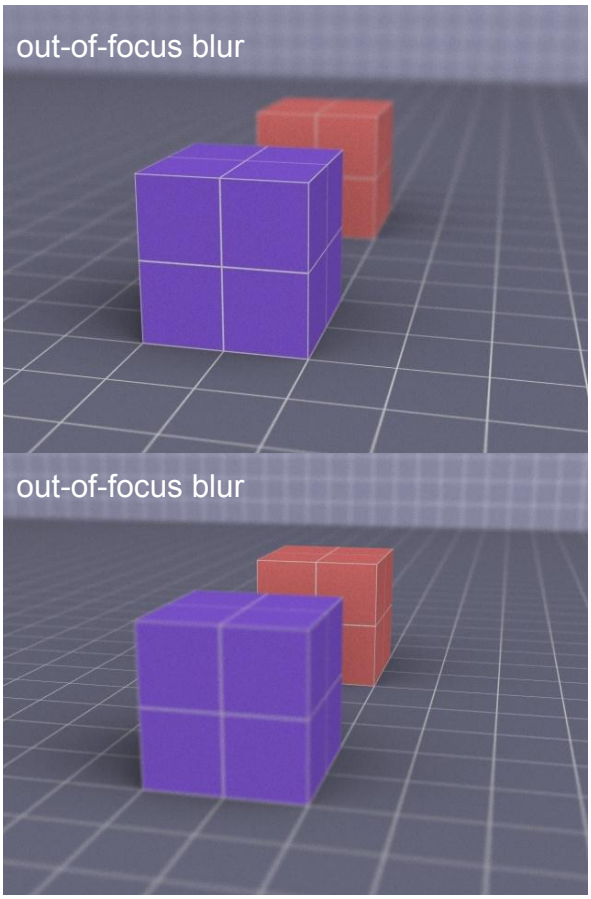


http://liam887.files.wordpress.com/2010/08/weaver.jpg

Depth of Field

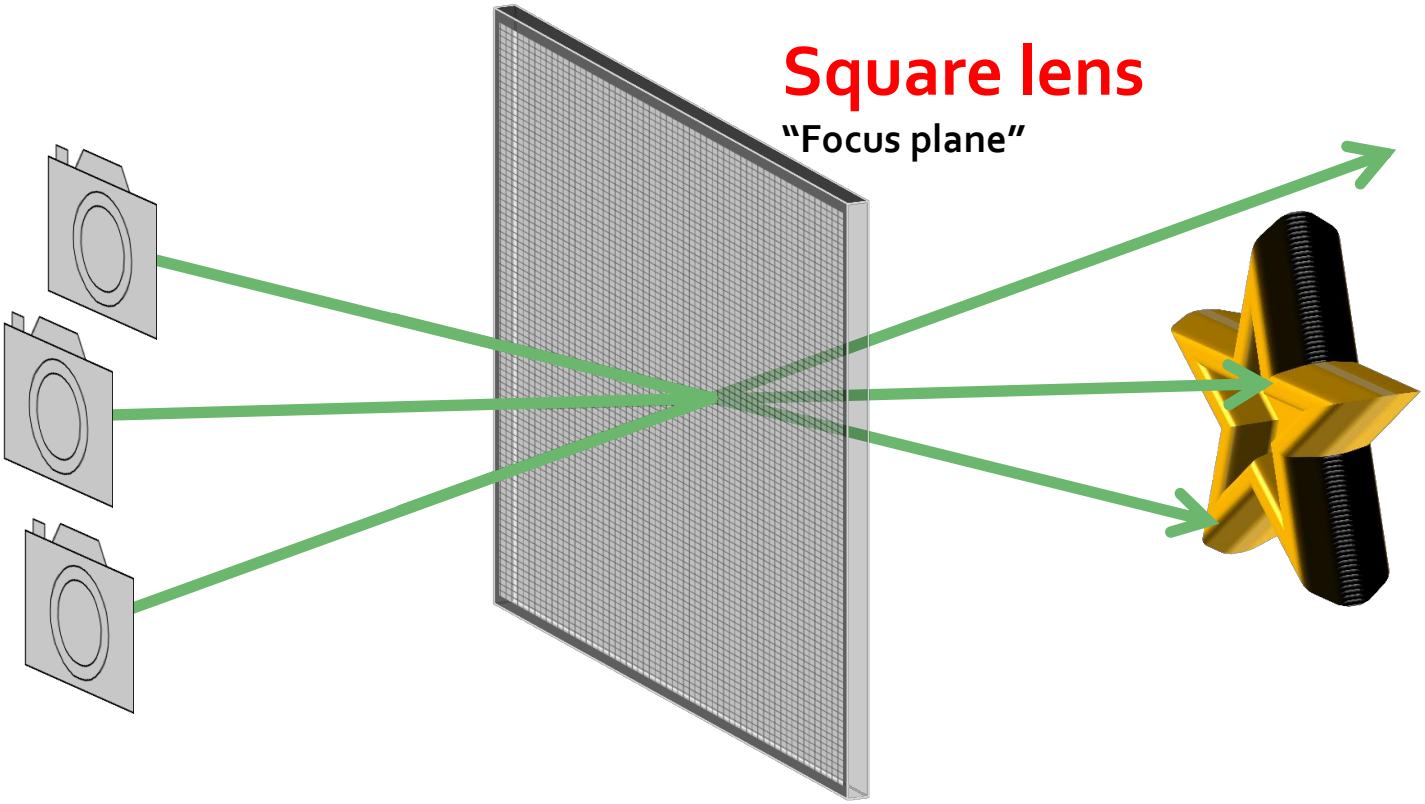
- Multiple rays per pixel, sample lens aperture
- Alternatively use normal ray tracing, but if object is not in focus (base on distance to the object), add its color to a neighboring pixels (weighted by a Gaussian)





Justin Legakis

Distribution Depth of Field



Randomly sample eye positions

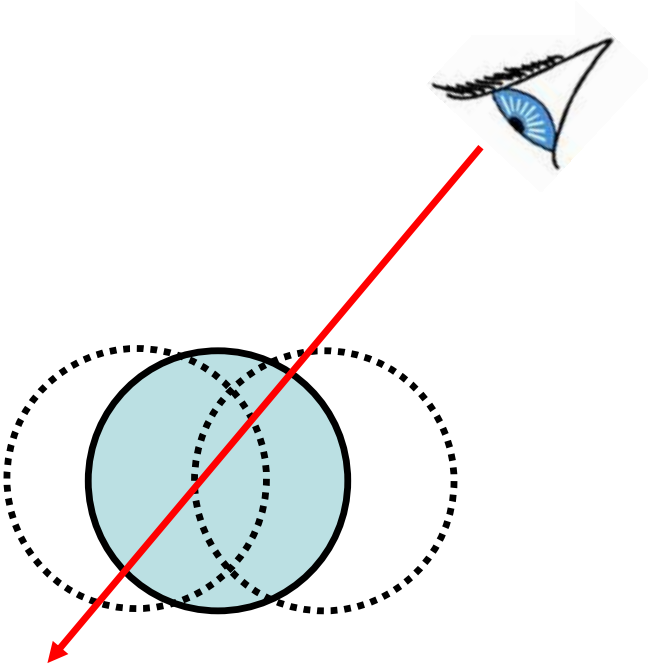
Problem: Exposure Time
Real Sensors Take Time to Acquire

Problem: Exposure Time
Real Sensors Take Time to Acquire

http://www.matkovic.com/anto/3dl-test-balls-01.jpg

Motion Blur

- Sample objects temporally over a time interval



Rob Cook

Next: Triangle Meshes and other data structures

- FOCG, Ch. 12
- Check out recommended reading for some additional references

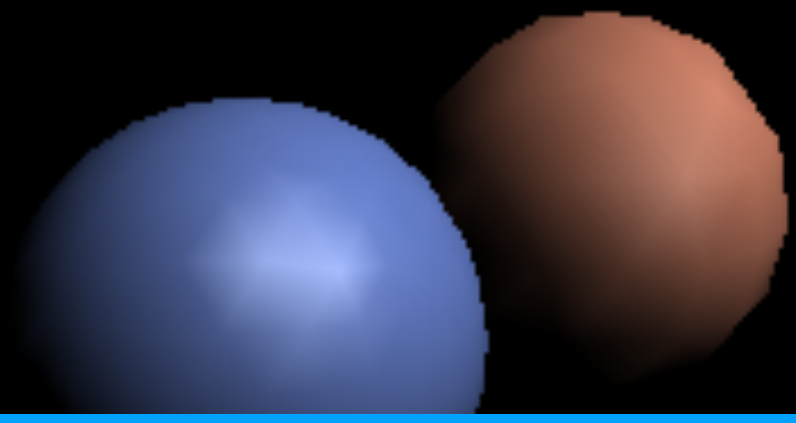
Shading on surfaces

- In practice, we have colors given either to each pixel (texture), or color for each vertex. The discussion below is only about shading
- For simplicity, assume surface has uniform color
- Problem: How could we produce the shading ? Shedding requires normal for each pixels
- If we are happy with a polyhedra surface - just compute for each face the normal.
- If on the other hand, the surface interpolates a smooth surface (e.g. a sphere), we should think about other alternative

Shading on Surfaces

- In practice, we have colors given either to each pixel (texture), or color for each vertex. The discussion below is only about shading
- For simplicity, assume surface has uniform color
- Problem: How could we produce the shading ? Shedding requires normal for each pixels
- If we are happy with a polyhedra surface - just compute for each face the normal.
- If on the other hand, the surface interpolates a smooth surface (e.g. a sphere), we should think about other alternative

Results of Gouraud Shading Pipeline



- Slightly idea. For every vertex v, compute the approximated normal.
- Compute the (shaded) color in each vertex
- Inside each triangle, use barycentric coordinates to interpolate the color.
- When interpolating inside a rectangle (billboard) use the values of α, β as discussed in hw3

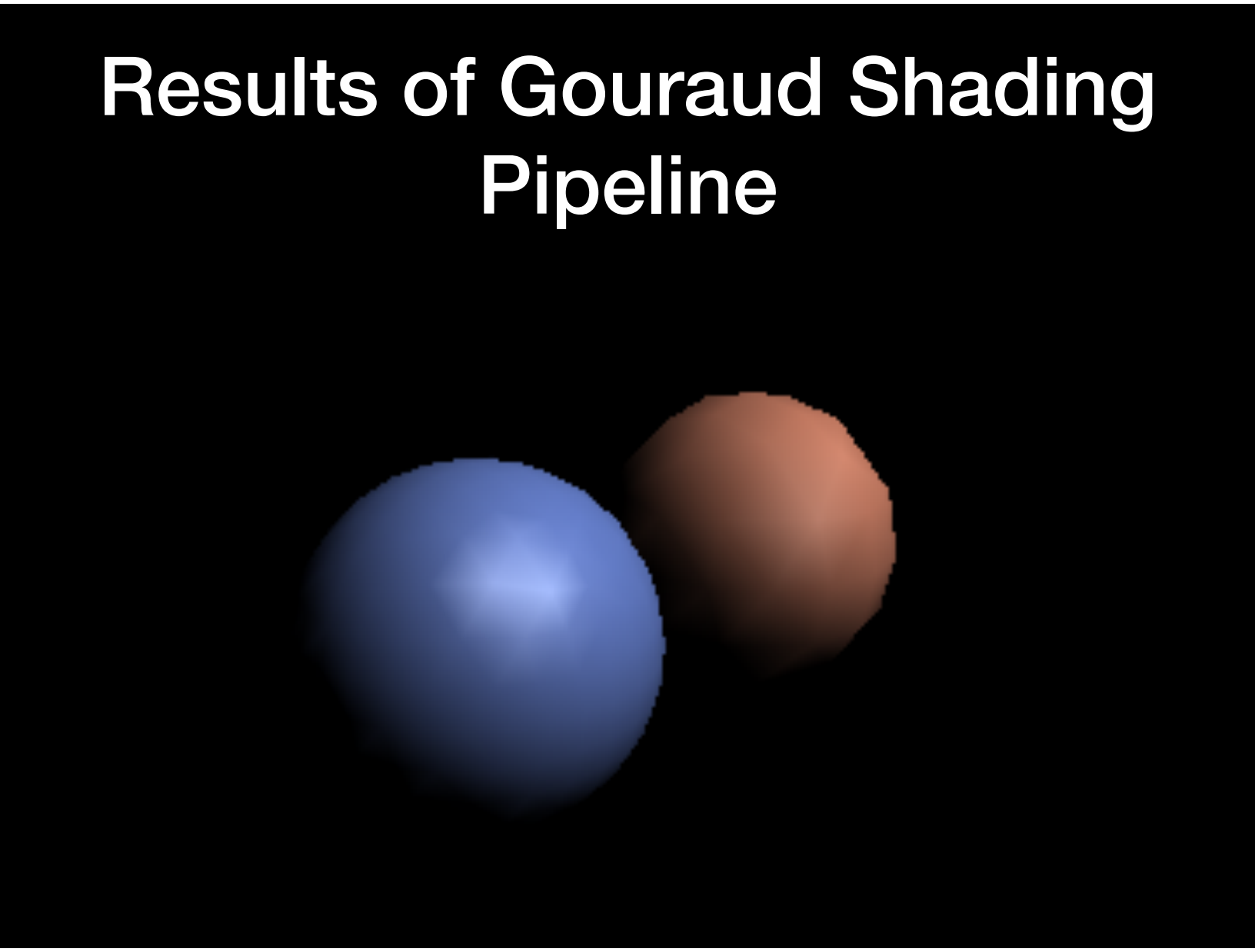
Problems w/ Gouraud Shading

- While you can use any shading model on the vertices (Gouraud shading is just an interpolation method!), typically using only diffuse color works best.
- Results tend to be poor with rapidly-varying models like specular color
    - In particular, when triangles are large relative to how quickly color is changing

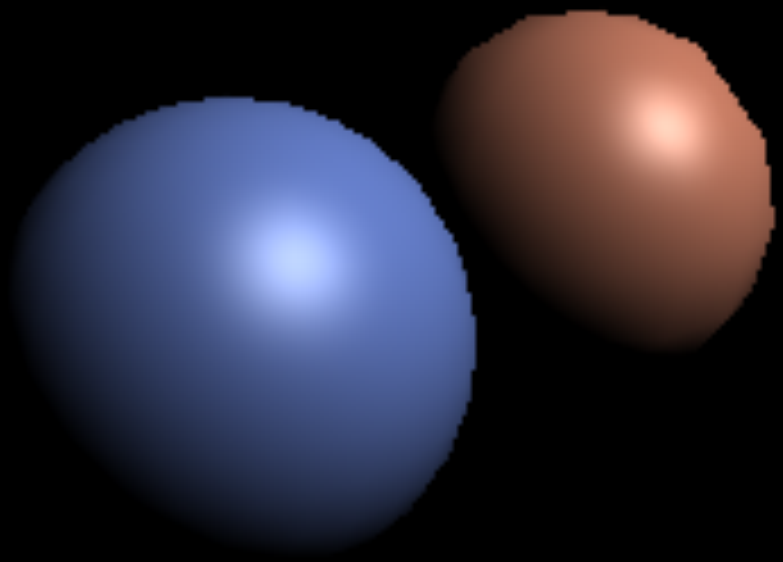
Better shading (but slower): Phong Interpolation Shading

- Think about a triangle with vertices v_1, v_2, v_3 or a billboard with corners $p_{LL}, p_{UL}, p_{LR}, p_{UR}$, compute the normals at the corners.
- For each pixel *p* on this triangle or billboard, express p as the convex combination of this corners (needs to compute the weights)
    - (for a triangle) $p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3$ (barycentric coordinates)
    - (for a recitingle)
      $p = \alpha\beta \cdot P_{UL} + (1-\alpha)\beta \cdot \quad P_{UR} \quad + \quad \alpha(1-\beta)P_{LL} + (1-\alpha)(1-\beta)P_{LR}$
- Compute its interpolated normals $\overrightarrow{n} = \alpha_1 \overrightarrow{n}_1 + \alpha_2 \overrightarrow{n}_2 + \alpha_3 \overrightarrow{n}_3$
- Normalize its length
- Use this normal (for each pixel) to compute its shading, as if it is the real normal
- See formula on whiteboard
- Caution: interpolated normals must be of unit length
- Caution: Don't confuse with Phong Specular Shading
    - (same person, two different concept)

Results of Gouraud Shading Pipeline



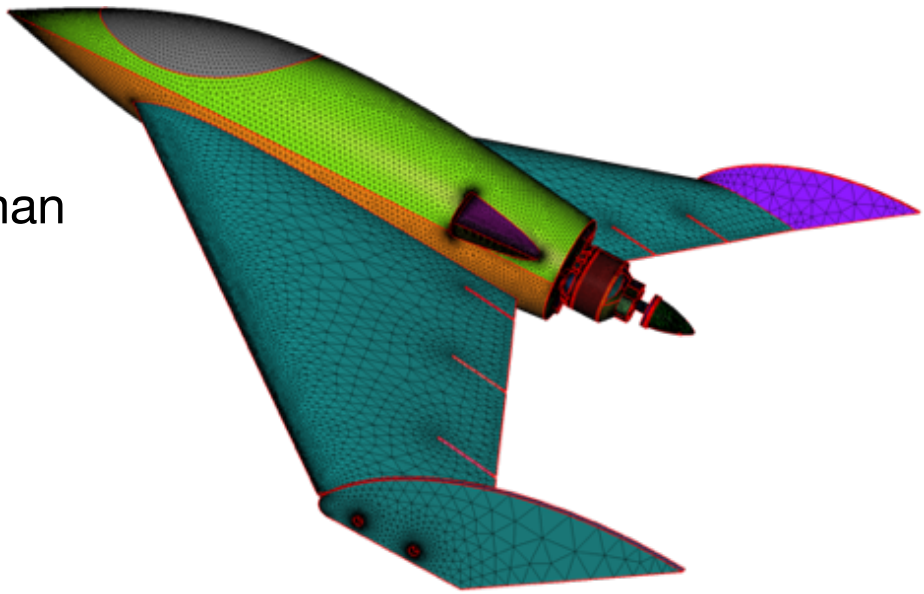
Results of Phong Shading Pipeline

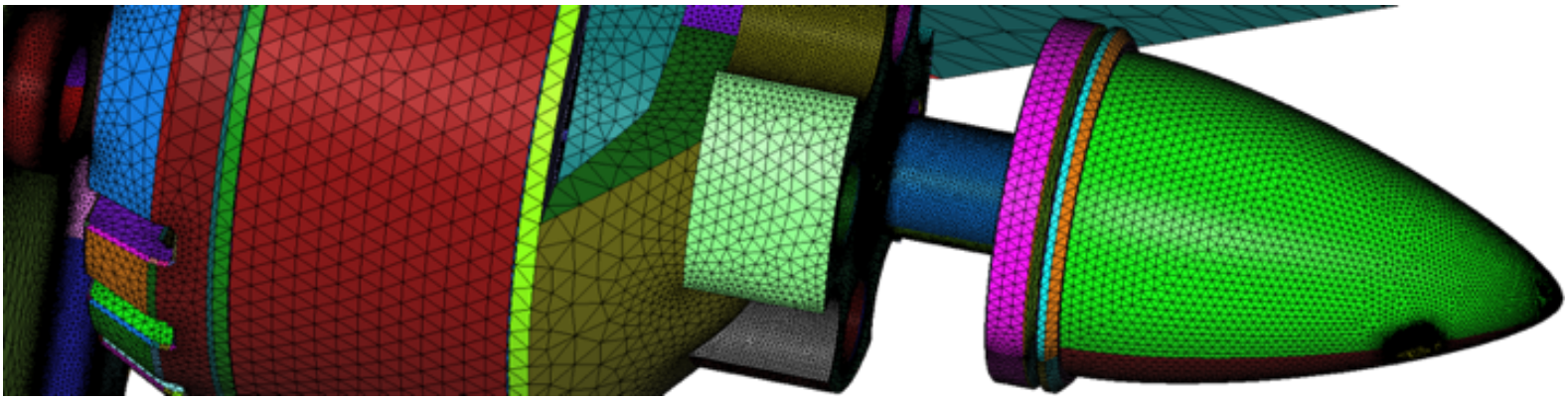


Material for midterm ends here

- Patrick Laug & Houman Borouchaki 2013

- 1,844,460 triangles

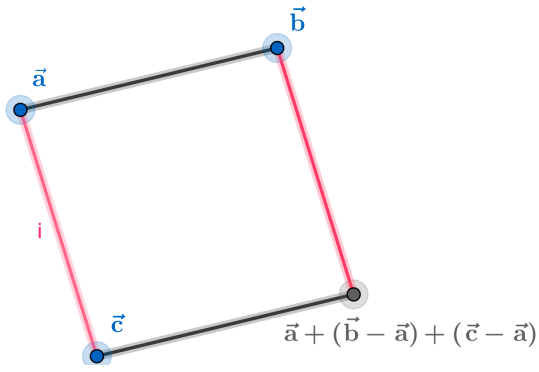




Modeling Complex Shapes

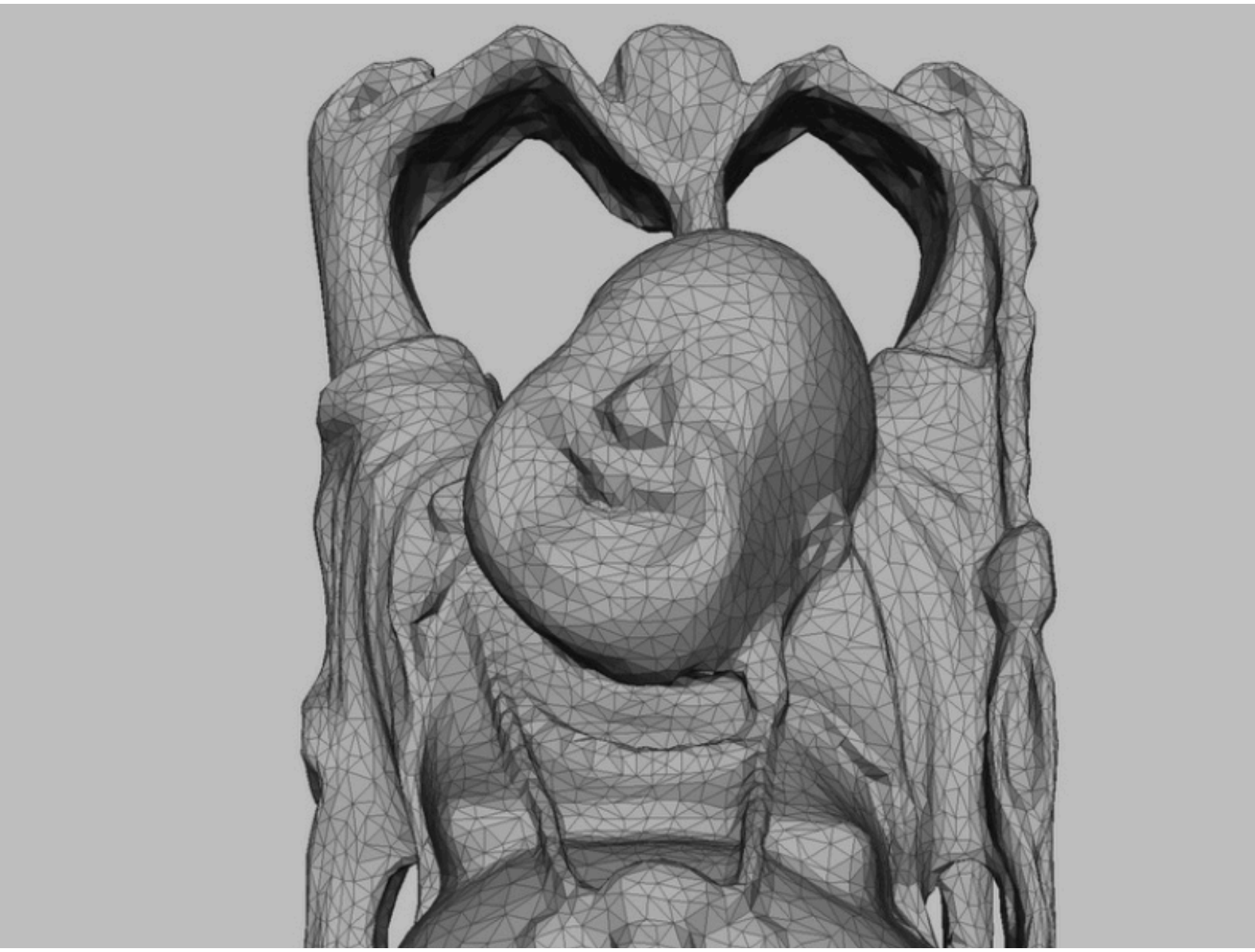
Recall: Shape Models That We Have So Far

- Implicit Shapes ($f(\mathbf{p}) = 0$ for all $\mathbf{p}$ on shape):
  - Sphere: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$
  - Plane: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$
- Parametric Shapes ($\mathbf{p}(t)$ is a point on shape for all t):
  - Rays: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Triangles:
    $p = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}, \quad 0 \leq \alpha, \beta, \gamma$ and $\alpha + \beta + \gamma = 1$
  - Triangle (second form)
    $p = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}), \quad 0 \leq \beta, \gamma$ and $\beta + \gamma \leq 1$
  - Parallelogon $p = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \qquad 0 \leq \beta, \gamma \leq 1$



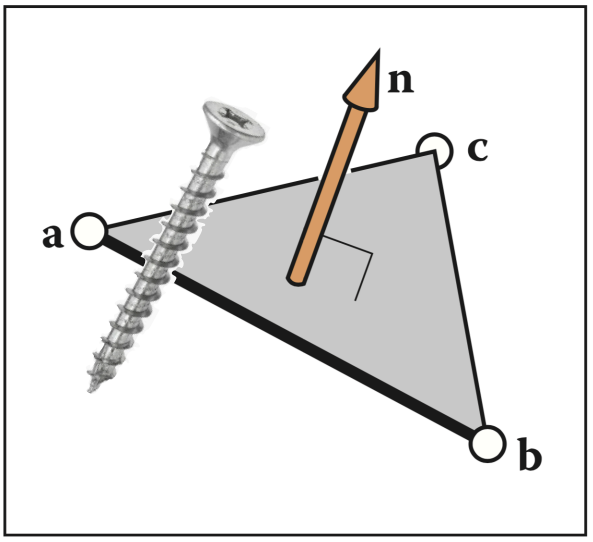
Triangle Meshes

- Are used in a huge number of applications
- Can be used to represent complex shapes by breaking them into simple (perhaps the simplest) two-dimensional elements

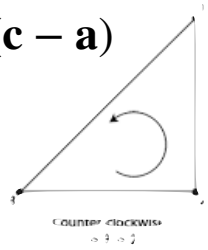


Definition of Triangles

- 3 **vertices** (points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ in 3D space)
- The normal of the triangle is a vector, $\mathbf{n}$, that points to its front side
- Convention: vertices listed in counter-clockwise order from the “front” of the triangle

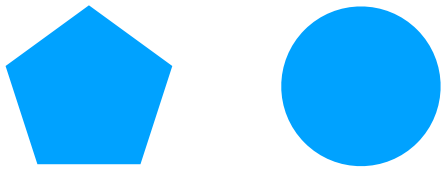


$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$



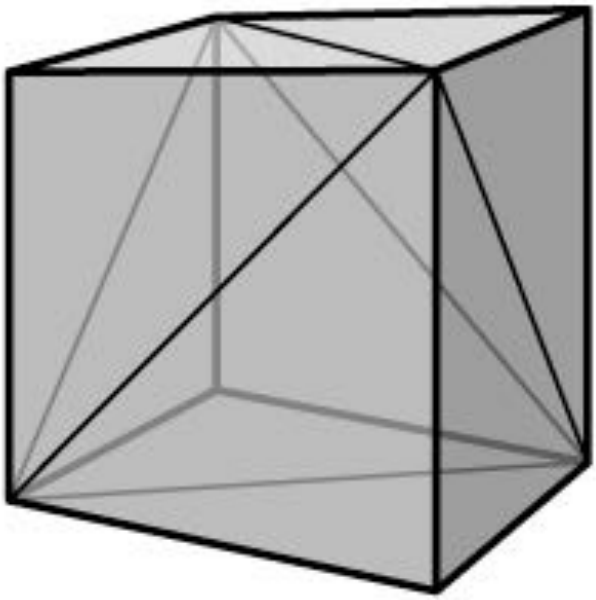
Definition of Triangle Meshes

- In short, a collection of triangles in 3D space that are connected to form a surface
  - Terminology: vertices, edges, triangles
  - Surface is piecewise planar, except where two triangle meet which forms a crease and their shared edge
  - Meshes are often a **piecewise** approximation of a smooth surface. We will study how graphics can hide the artifacts, creates the illusion of a **smooth** surface without increasing their comolexity.



A Simple Mesh

- How many vertices? How many triangles?



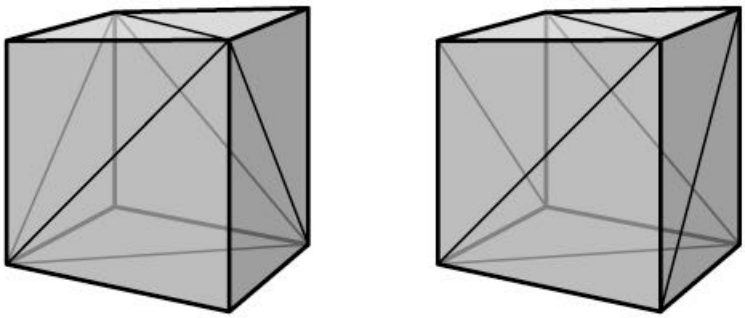
Mesh Topology

Two Considerations for Meshes

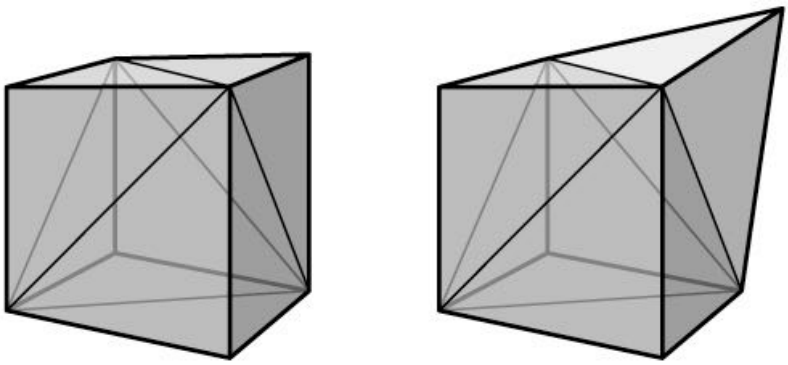
- We typically care about the mesh being a good approximation to a surface:
  - This leads to questions of **mesh geometry**, e.g.: How many triangles? where to place their vertices?
- We also care about how these triangles are connected
  - This leads to questions of **mesh topology**, e.g.: Are there holes in the mesh? How do triangles intersect?
  - Mesh topology can affect assumptions on algorithms that process meshes

Topology vs. Geometry

- Same geometry, different topology



- Same topology, different geometry



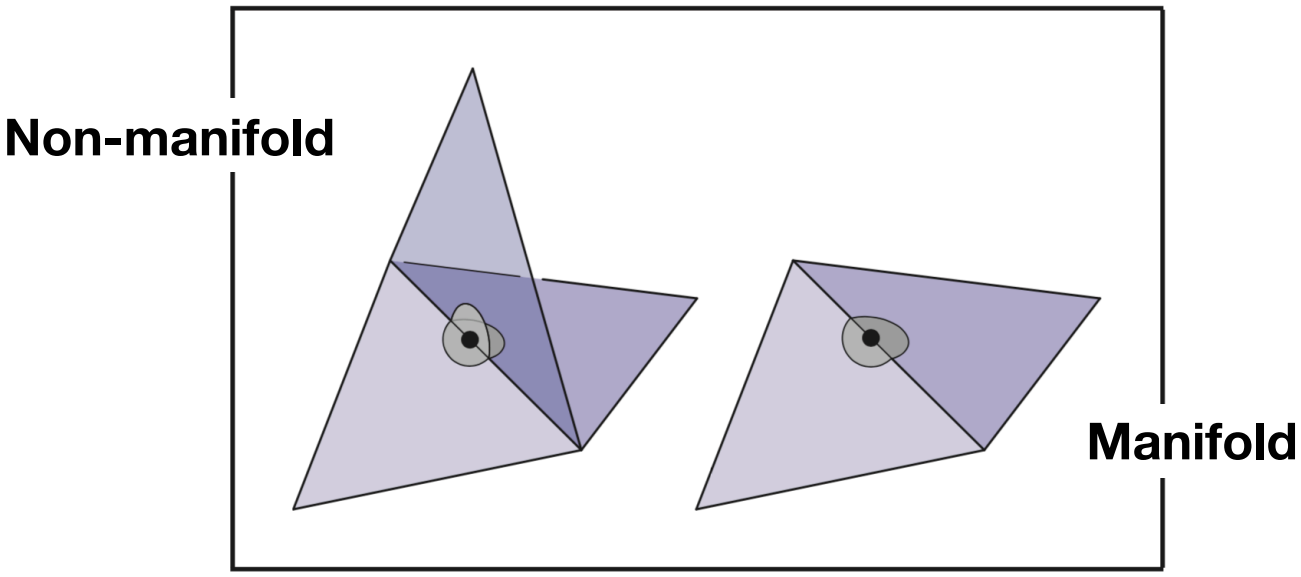
Topological Validity

- Meshes that approximate surfaces should be manifolds
- Definition: A (2-dimensional) **manifold** is a space where every point locally appears to be 2-dimensional space
    - 3 cases: points that are on edges, points that are vertices, and points that are interior to triangles.

When is a Mesh a Manifold?

- Definition: A (2-dimensional) manifold is a space where every point locally appears to be 2-dimensional space

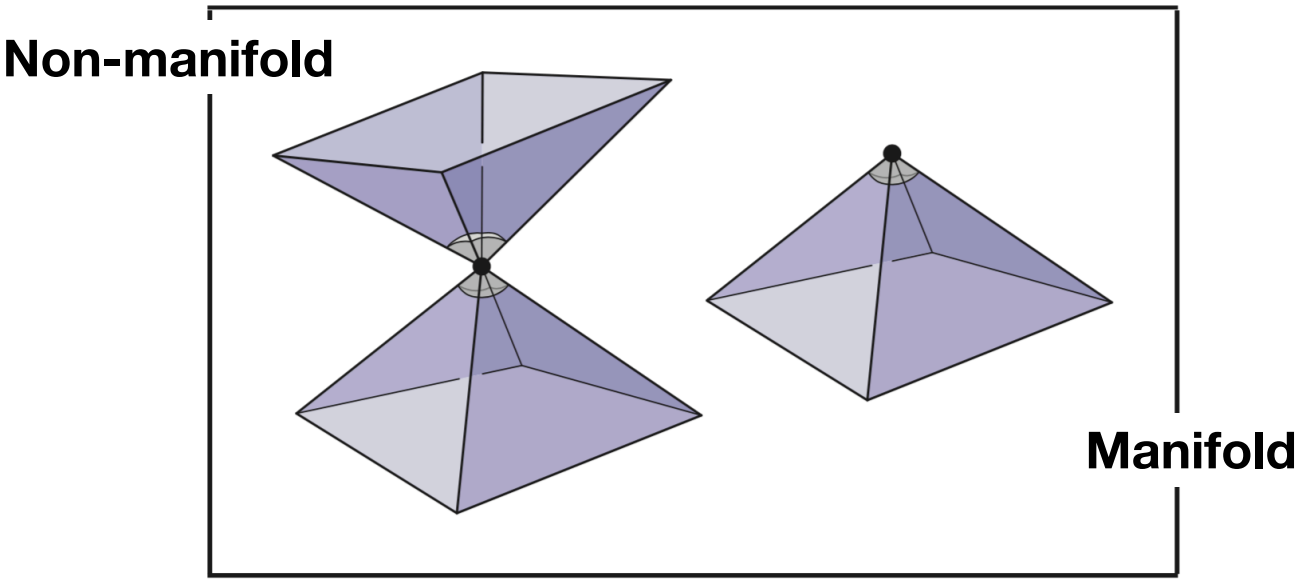
- Implication: Every edge is shared by exactly two triangles



When is a Mesh a Manifold?

- Definition: A (2-dimensional) manifold is a space where every point locally appears to be 2-dimensional space

- Implication: Every vertex has a single, complete loop of triangles around it

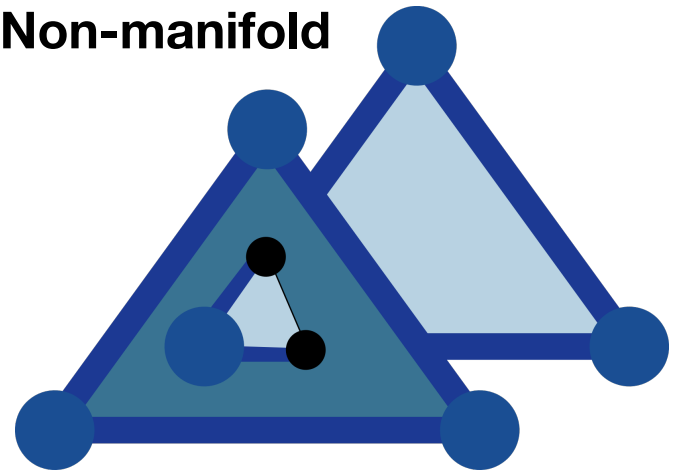


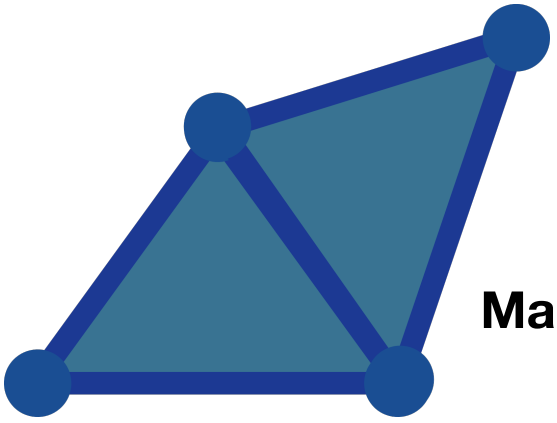
When is a Mesh a Manifold?

- Definition: A (2-dimensional) manifold is a space where every point locally appears to be 2-dimensional space

- Implication: Triangles only intersect at vertices and edges

Non-manifold

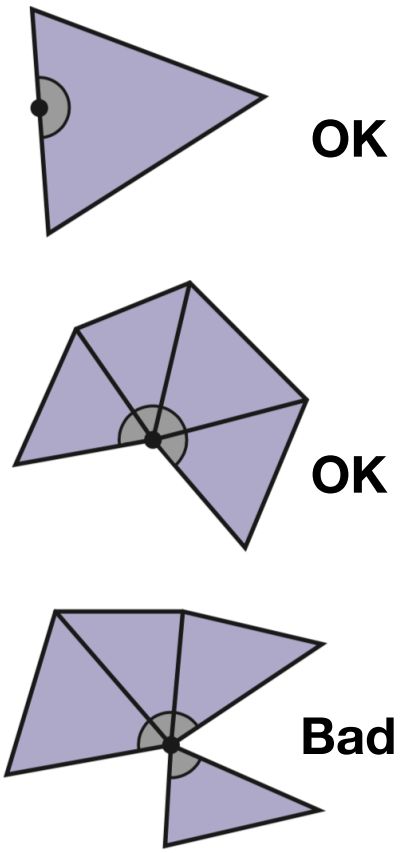




Manifold

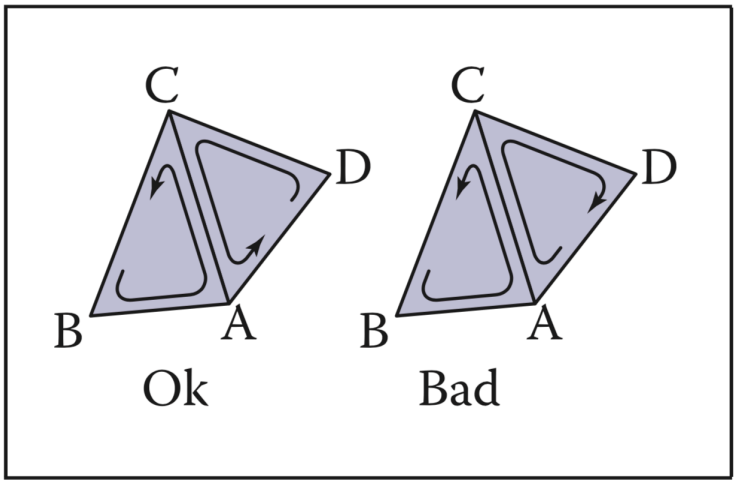
Manifolds with Boundary

- Sometimes, we relax the manifold condition to allow meshes with boundaries.
- Every point on a **manifold with boundary** either locally appears to be 2-dimensional space or 2-dimensional half-space
  - Every edge is used by either one or two triangles
  - Every vertex connects to a single edge-connected set of triangles

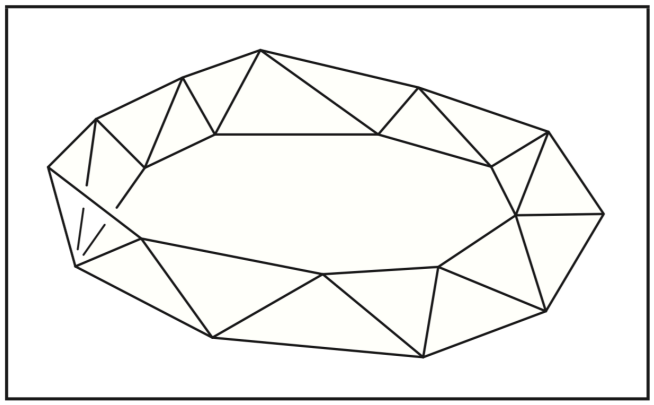


Consistent Orientation

- In many applications, all triangles facing the same way is important
  - Can be used to distinguish inside from outside.
- If consistent: neighboring triangles will appear to disagree on the order of vertices on their shared edge



Möbius strip: Non-orientable



Simple Representations of Triangle Meshes

Important Concerns w/ Representing Triangle Meshes

- Efficiency of storage size
  - Many representations store redundant information
- Efficiency of access
  - How quickly can we get the information we need for rendering?
  - How quickly can we get neighborhood information, for mesh modification?

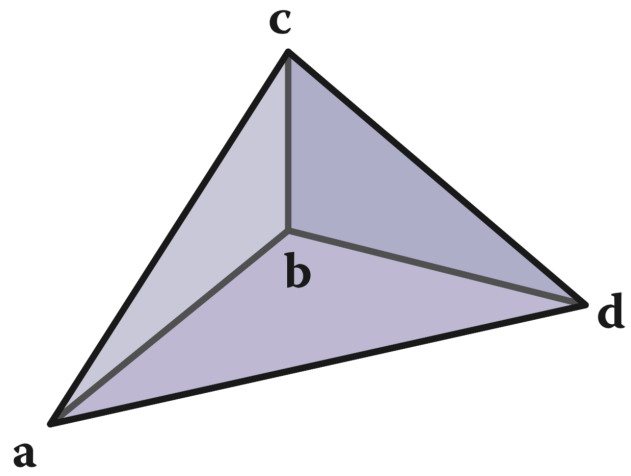
Using Separate Triangles

- Use a simple structure to store each triangle:

```
Triangle {
  vertexPositions[3];  //Vec3
};
```

- Store a triangle mesh using an array of `Triangle`

- Problems?



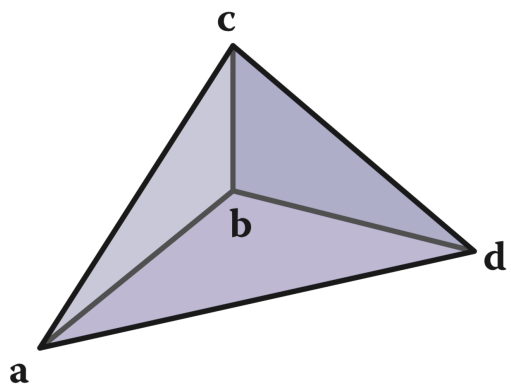
#	*Vertex* 0	*Vertex* 1	*Vertex* 2
0	(a_x, a_y, a_z)	(b_x, b_y, b_z)	(c_x, c_y, c_z)
1	(b_x, b_y, b_z)	(d_x, d_y, d_z)	(c_x, c_y, c_z)
2	(a_x, a_y, a_z)	(d_x, d_y, d_z)	(b_x, b_y, b_z)

Using Indexed Meshes

- Triangles share a common list of vertices, storing only references/ pointers:

```
Triangle {
  vertices[3]; //object reference or int
};

Vertex {
  position;    //Vec3
};
```

- Store a triangle mesh using two arrays, one of `Vertex` and the other of `Triangle`

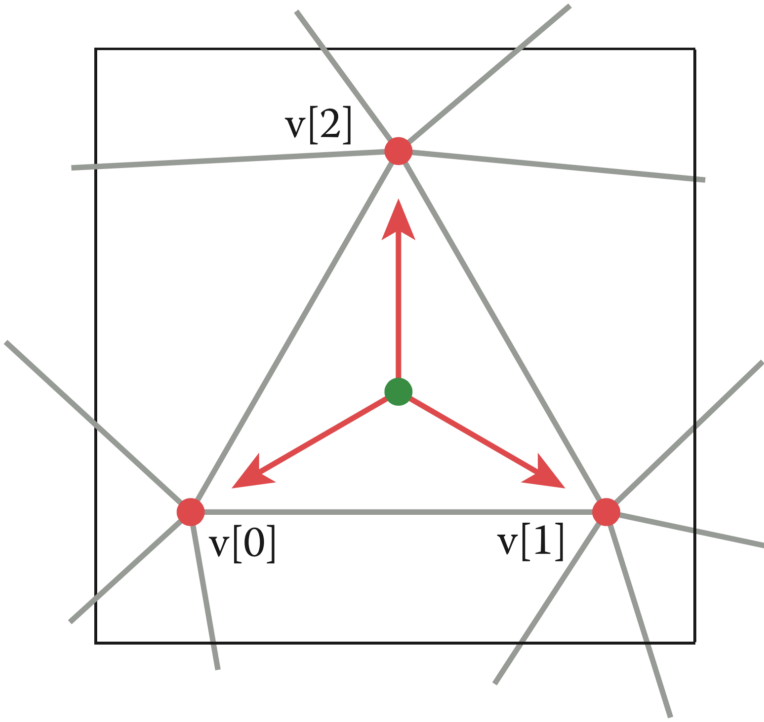


Triangles	
#	*Vertices*
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

Vertices	
#	*Position*
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Using Indexed Meshes

- Each triangle thus tracks references to the vertices associated with it

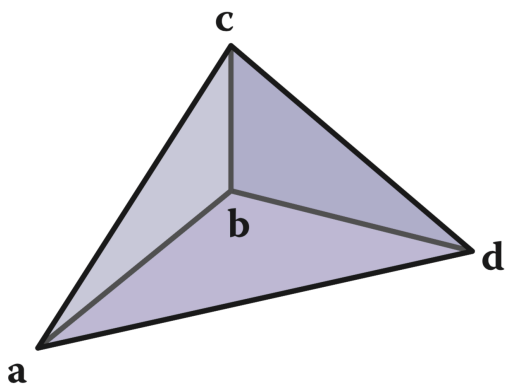


Using Indexed Meshes

- Alternatively one can store using array indices directly:

```
IndexedMesh {
  vertices[num_verts];    //Vec3
  triIndices[num_tris];   //int
};
```

- Plus, it is easy (or at least easier) to see which two triangles share an edge.



Triangles	
#	*Vertices*
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

Vertices	
#	*Position*
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Storage Requirements

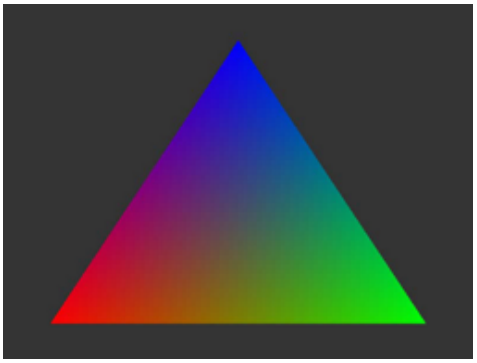
- Let n_v be the number of vertices and n_t be the number of triangles.
  - A single vertex requires 12 bytes (3 floats, 4 bytes/float)
- Separate triangles:
  - 3 vertices per triangle => $12*3*n_t = 36*n_t$ bytes
- Indexed Meshes
  - 3 integers per triangle, 3 floats per vertex => $12*n_t + 12*n_v$ bytes

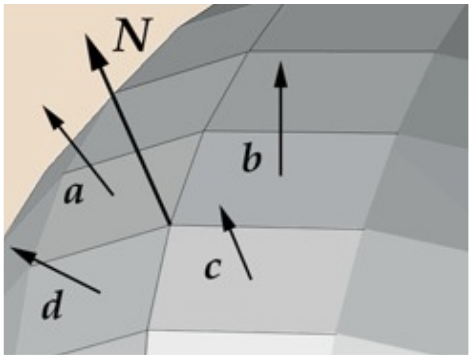
Storage Requirements

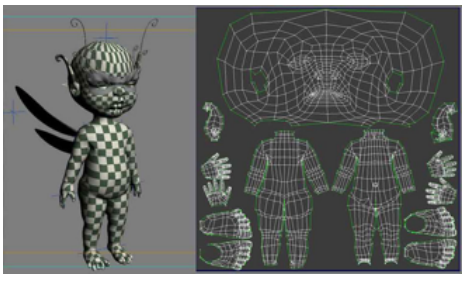
- For large meshes, $n_t \approx 2n_v$
- Separate triangles:
  - $36*n_t = 72*n_v$ bytes
- Indexed Meshes
  - $12*n_t + 12*n_v = 36*n_v$ bytes

Data on Meshes

- Typically, we store a variety of data on meshes as well
- Can store this on vertices, triangles, or even edges
- Examples:
  - Colors stored on vertices
  - Normals stored on faces
  - Texture coordinates stored on vertices
- Information stored on vertices is typically interpolated with **barycentric coordinates**

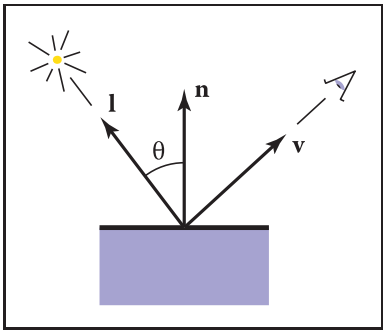






Remember Diffuse Shading

- Simple model: amount of energy from a light source depends on the direction at which the light ray hits the surface



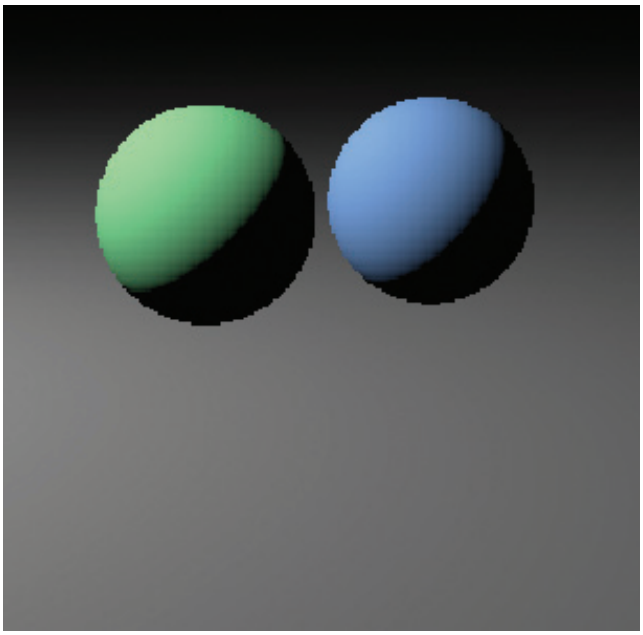
- Results in shading that is *view independent*

$$L_d = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffuse coefficient

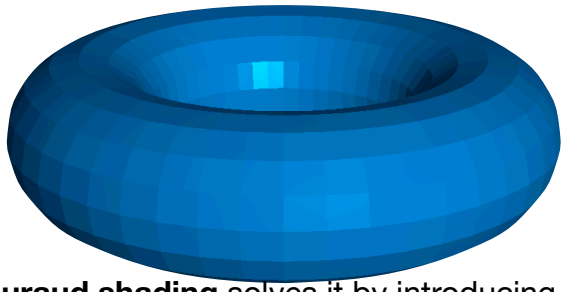
intensity/color of light

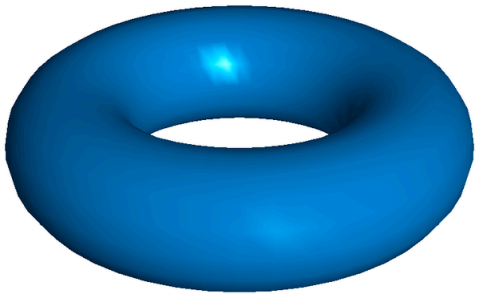
$\cos \theta$



Shading on triangle meshes. Gouraud Shading

- The shading of each triangle is determined by its normal (same normal for all points in the triangle). Edges of triangles are very noticeable. This is called ***flat shading***





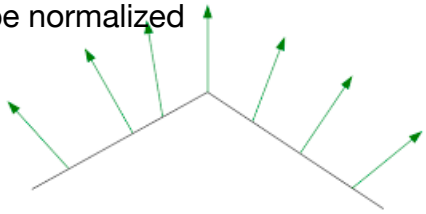
Diffuse shading formula

$$L_d = k_d \cdot I(\vec{\mathbf{n}} \cdot \vec{l})$$

$\vec{l}$ *-direction to light*

Gouraud shading solves it by introducing **fake normals.**
When computing the shading at a point, we use the fake normals
These fake normals changes continuously on the surface.
(the real normals ``jumps" at edges)

- First the normal at each vertices is computed. It is the ``average" of the normal of the adjacent face.
- Next, each pixel *p* in a triangle uses barycentric coordinates to interpolate its own normal, based on the vertices normals.
- Convex combination of unit vector is not necessarily of unit length, so the fake normals must be normalized

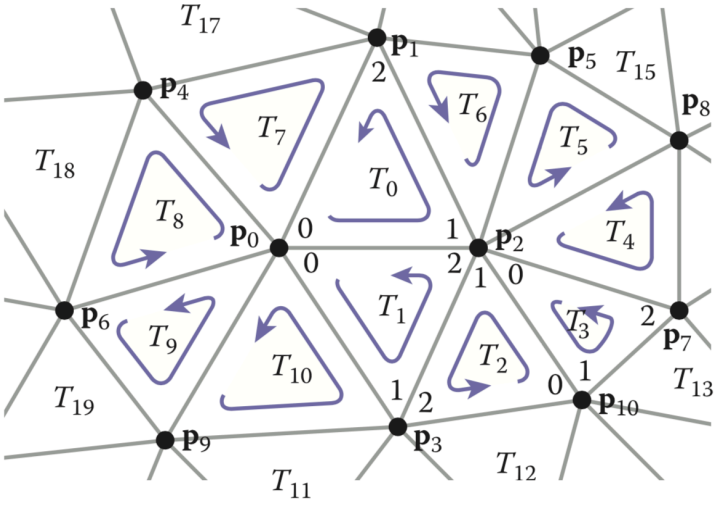


Mesh File Formats: *.obj

- Widely used format for indexed meshes
- Supports additional data stored on vertices and polygons

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
verts[2]	x_2, y_2, z_2
verts[3]	x_3, y_3, z_3
	⋮

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
tInd[2]	10, 2, 3
tInd[3]	2, 10, 7
	⋮

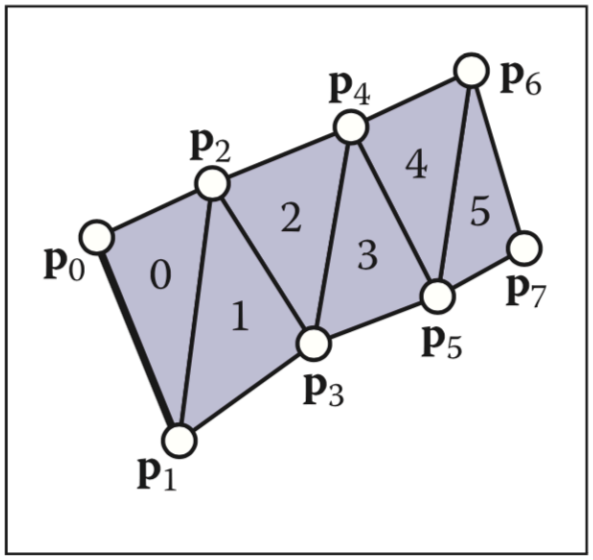



```
#sample .obj file

v 0.000 0.000 0.000
v 0.500 0.809 0.309
v 1.000 0.000 -0.309
v 0.583 -0.720 0.225
v -0.630 0.750 0.025
...

f 1 3 2
f 1 4 3
f 11 3 4
f 3 11 7
...
```

More Efficient Representations

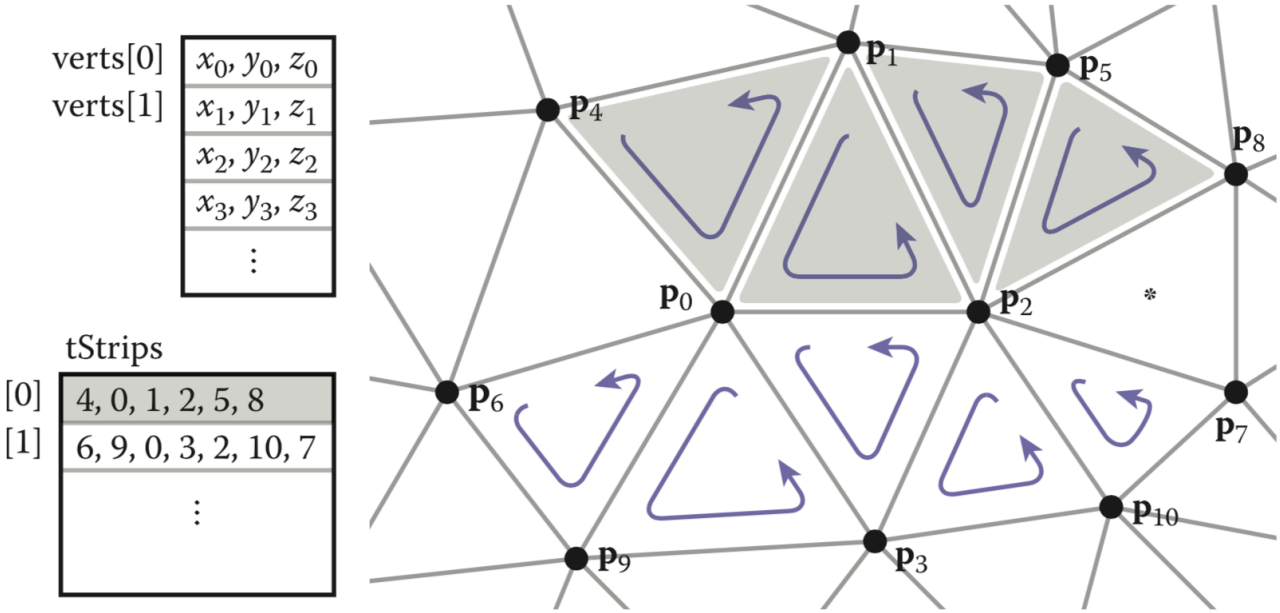
Triangle Strips

- Idea: Rely on the mesh property and group triangles that share common vertices
- Create a new triangle by reusing the last two vertices in the strip
- [0,1,2,3,4,5,6,7] specifies the sequence on the right with triangles (0,1,2), (1,2,3), (2,3,4) …
- Have to invert every other for consistent orientation



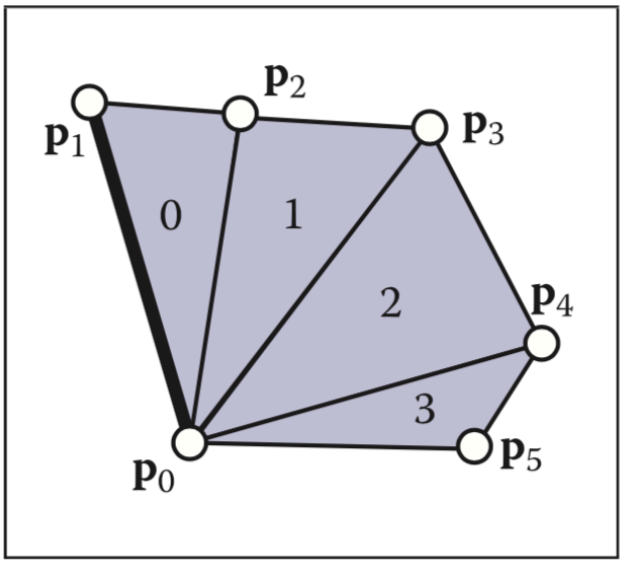
Triangle Strips

- Complex meshes store list of strips
- How long of a strip to use?



Triangle Fans

- Same idea as triangle strips, but keep the earliest vertex in the list instead of the last two
- [0,1,2,3,4,5] specifies the sequence on the right with triangles (0,1,2), (0,2,3), (0,3,4), …



Mesh Data Structures and Queries

Queries on Meshes

- For face, find all:
  - Vertices
  - Edges
  - Adjacent faces
- For vertex, find all:
  - Incident edges
  - Incident triangles
  - Neighboring vertices
- For edge, find:
  - Two adjacent faces
  - Two adjacent vertices

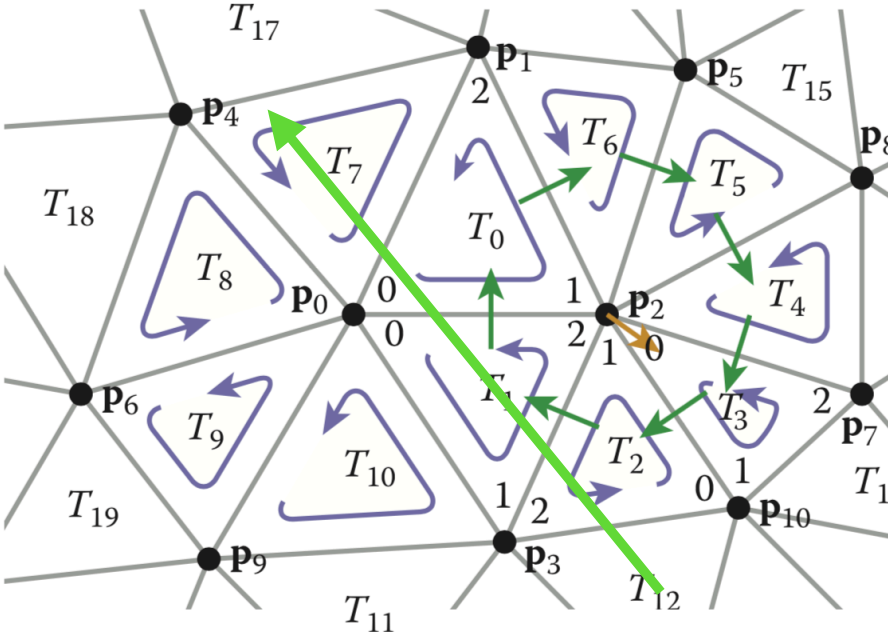
Can we do better?

```
Triangle {
  v[3];   //Vertex
  e[3];   //Edge
  adj[3]; //Triangle
}

Vertex
  t[];    //Triangle
  e[];    //Edge
  adj[];  //Vertex
}

Edge {
  v[2];   //Vertex
  t[2];   //Triangle
}
```

Typical Operations on the Data Structure

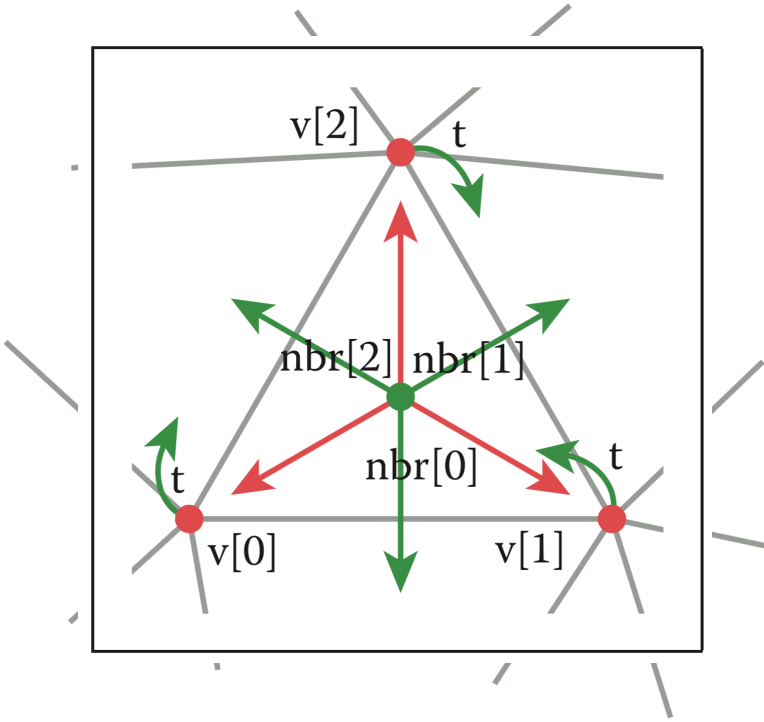
- Once a triangle T_0 is given which triangles are neighboring T_0 ?
- Given a ray r, which triangles intersect r?

- (Older material) is a point q=(x,y,z) inside T_0 ? (solved with barycentric coordinates.
- Who are the triangles that are adjacent to a vertex p_0 ?



Triangle-Neighbor Structure

- Let's try first extending the indexed mesh structure for sharing vertices
- Add pointers, `nbr[]`, to 3 neighboring triangles
- Add a single pointer, `t`, for each vertex to one of its adjacent triangles
- Can now enumerate triangles adjacent to vertices

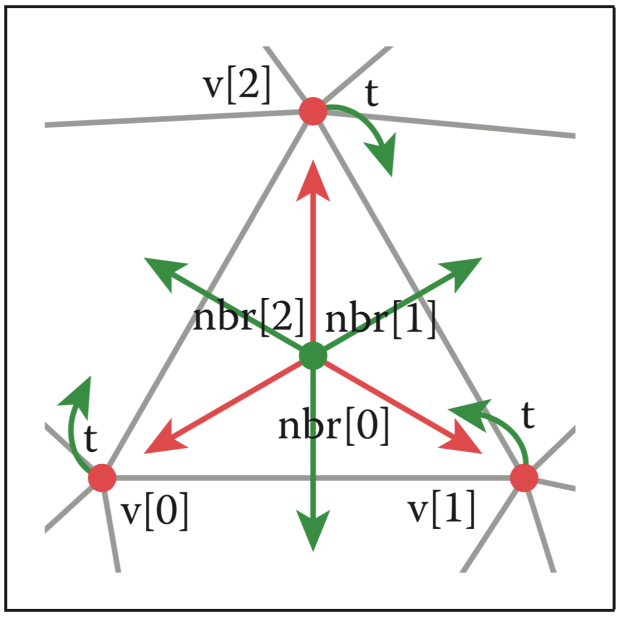


Triangle-Neighbor Structure

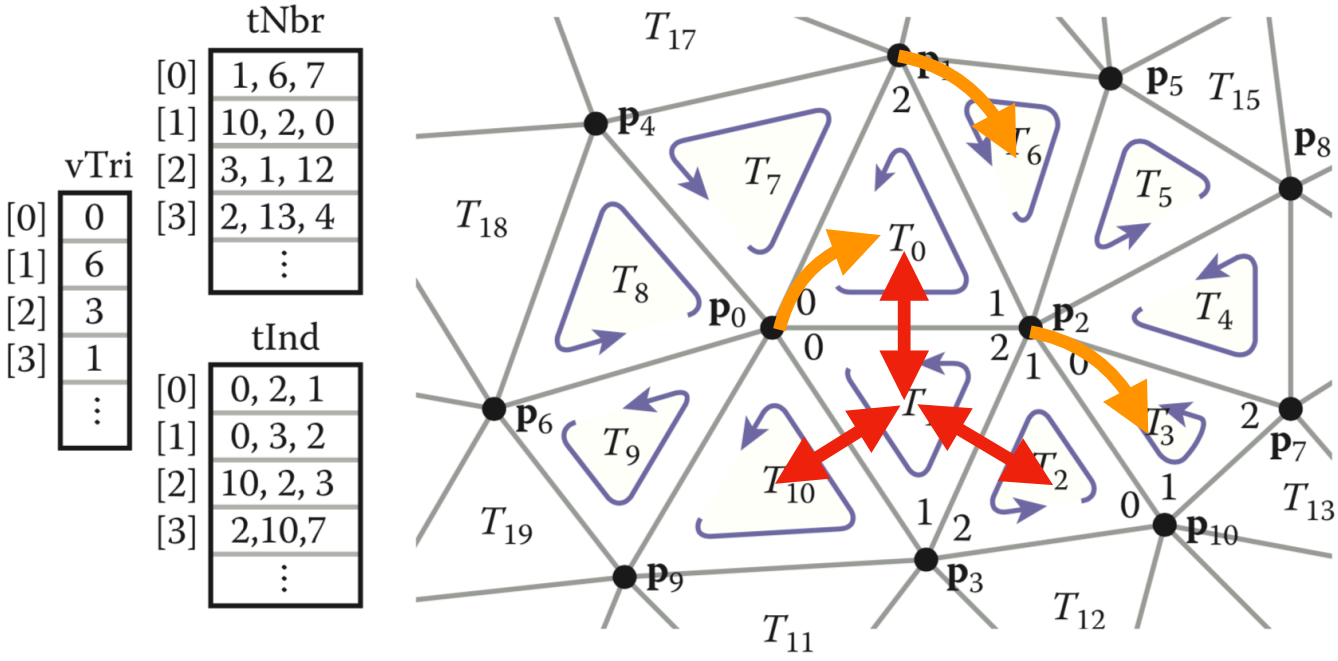
```
Triangle {
  v[3];     //Vertex
  nbr[3];   //Triangle
}

Vertex {
  ...
  t;        //Triangle
}
```

...or...

```
IndexedMesh {
  ...
  tInd[num_tris];   //int[3]
  tNbr[num_tris];   //int[3]
  vTri[num_verts]; //int
};
```



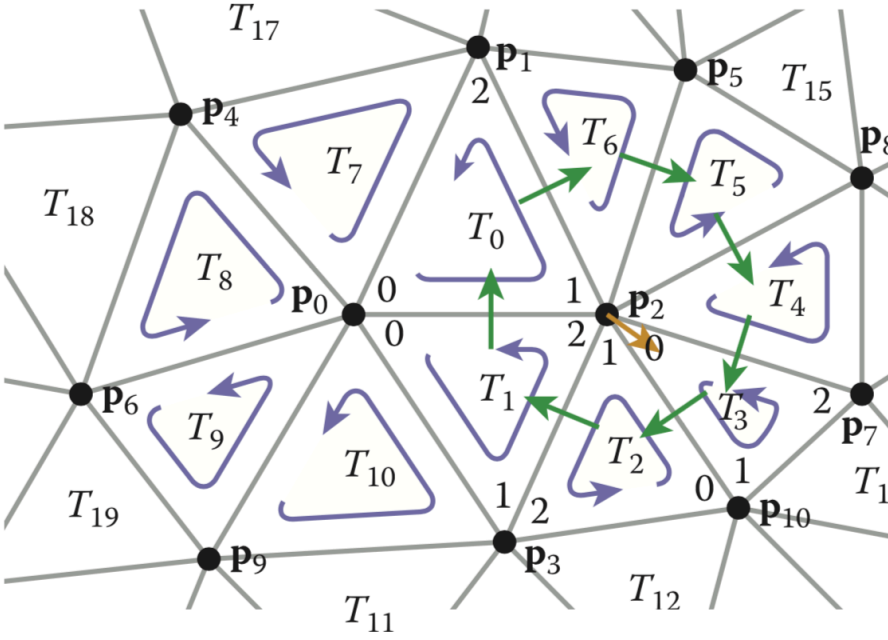
Triangle-Neighbor Structure



Triangle-Neighbor Structure

```
TrianglesOfVertex(v) {
  t = v.t
  do {
    find i where (t.v[i] == v)
    t = t.nbr[i]
  } while (t != v.t);
}
```

- Can optimize by storing pointers to neighboring edges



Triangle-Neighbor Structure

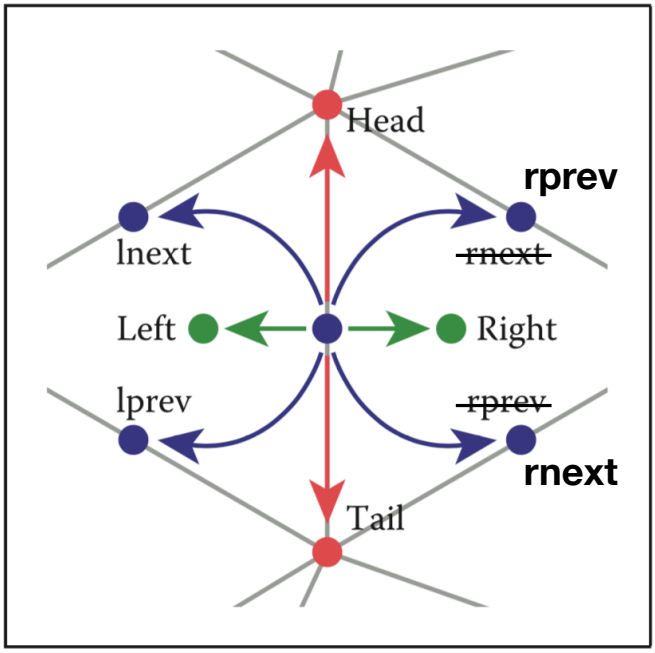
- Recall that indexed meshes needed $36*n_v$ bytes and $n_t \approx 2n_v$
- We added an array of triples of indices (per triangle)
  - This increases storage by $3*4*n_t$ or $24*n_v$ bytes
- We also added an array of representative triangle per vertex
  - This increases storage by $4*n_v$ bytes
- Total storage: 36 + 24 + 4 = 64 bytes per vertex
  - Still not as much as separate triangles

Winged-Edge Structure

- Widely used mesh structure that focuses on edges instead of triangles

- Edges store pointers to:
  - Head/Tail vertices
  - Left/Right triangles
  - Left/Right “next” edges
  - Left/Right “previous” edges

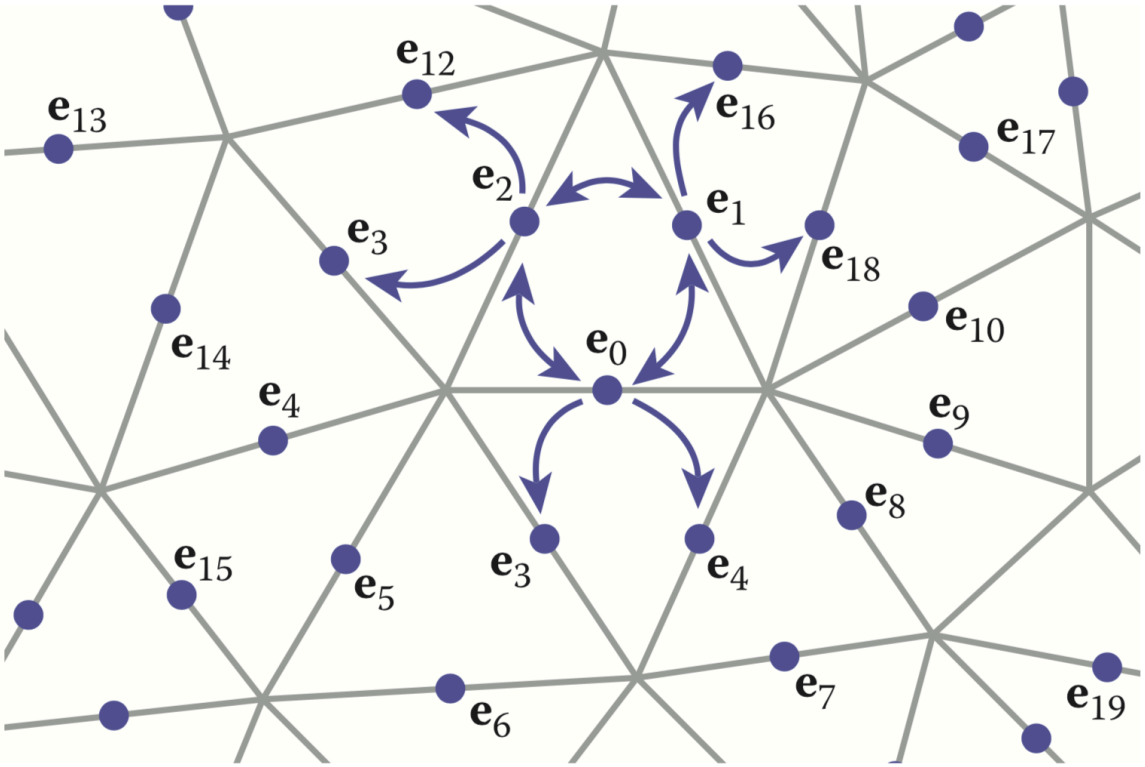
- Each vertex/triangle stores one pointer to some edge



Winged-Edge Structure

Winged-edge table

	ln	rp	lp	rn
[0]	1	4	2	3
[1]	18	0	16	2
[2]	12	1	3	0
	⋮			



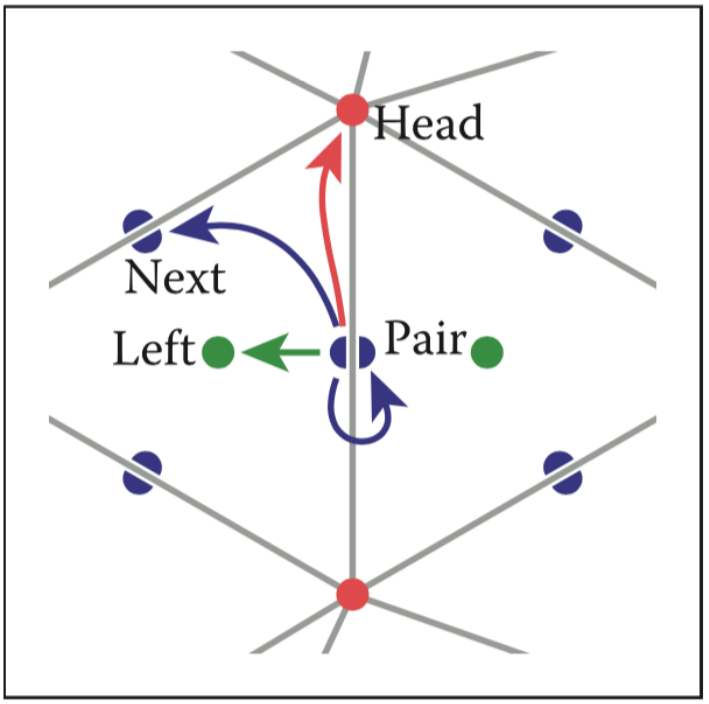
Half-Edge Structure

(sometimes called Doubly connected Edge List -DCEL)

- Simplifies winged-edge, removes awkwardness of checking which way edges are oriented

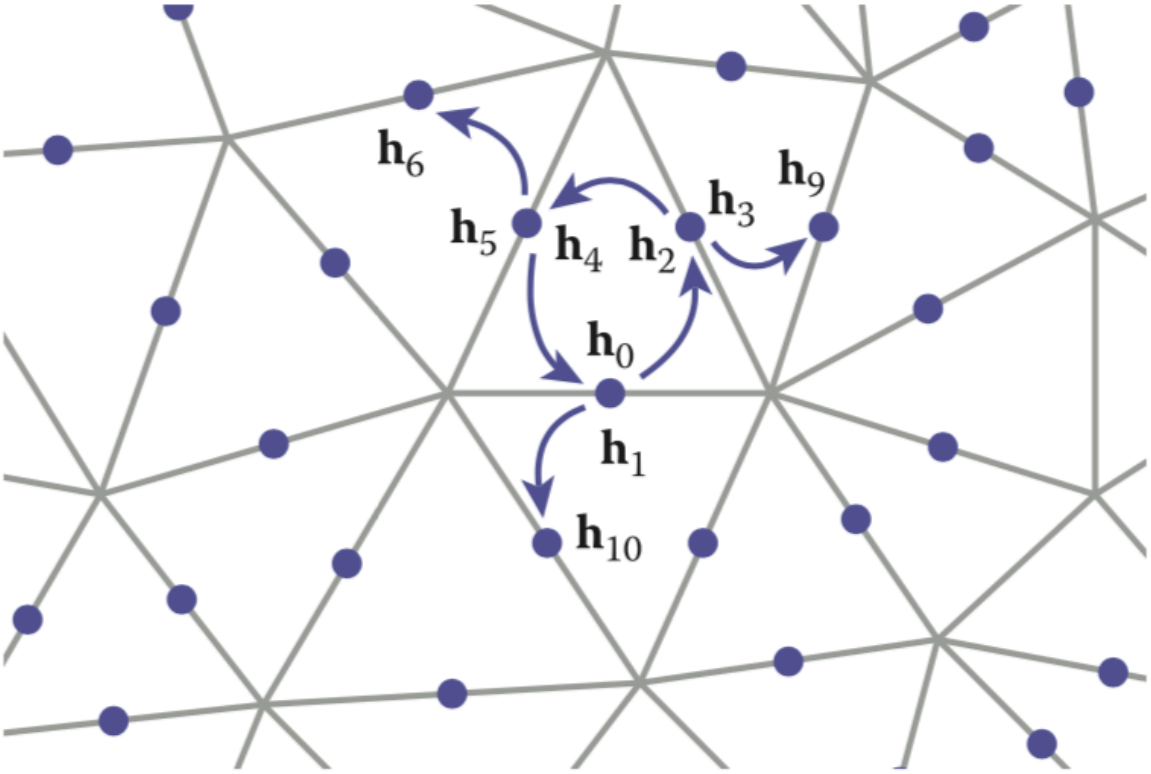
- Each **half-edge** store pointers to:
  - Head vertex
  - Left triangle
  - Left “next” edge
  - The opposite “pair” half-edge (the twin edge)

- Each vertex/triangle stores one pointer to a half-edge



Half-Edge Structure

	Pair	Next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



Half-Edge Structure

```
HEdge {
  pair, next;  //HEdge
  v;           //Vertex
  f;           //Face
};
```

```
EdgesOfVertex(v) {
  h = v.h;
  do {
    h = h.next.pair;
  } while (h != v.h);
}
```

```
EdgesOfVertex(v) {
  e = v.e;
  do {
    if (e.tail == v) {
      e = e.lprev;
    } else {
      e = e.rprev;
    }
  } while (e != v.e);
}
```

Winged-Edge Implementation

Half-Edge Storage Requirements

- Vertex data: 3 floats for position, 1 int for edge reference
  - $4*4 = 16n_v$ bytes
- Face data: 1 int for edge reference
  - $4*1 = 4*n_t = 8n_v$ bytes.
- Edge data, 4 ints for references, but store a pair of half edges for each edge
  - $n_h \approx 6n_v$
  - $8*4*6 = 96n_v$ bytes.
- In total, $120n_v$ bytes.

Secure | https://www.openmesh.org

OpenMesh

Visual Computing Institute | RWTH AACHEN UNIVERSITY

Home

Introduction
Documentation
FAQ

Download
Git

License
Participating
Bugtracking

OpenFlipper

Contact

OpenMesh

A generic and efficient polygon mesh data structure

OpenMesh is a generic and efficient data structure for representing and manipulating polygonal meshes. For more information about OpenMesh and its features take a look at the Introduction page.

On top of OpenMesh we develop OpenFlipper, a flexible geometry modeling and processing framework.

News

- **OpenMesh 6.3 released** — Oct. 4, 2016

  OpenMesh 6.3 is still fully backward compatible with the 2.x to 5.x branches. We marked some functions which should not be used anymore as deprecated and added hints which should be used instead.

  This will be the last release officially supporting C++98 compilers and building the integrated applications with Qt 4.

  The update adds a workaround for an gcc optimizer bug causing segfaults when optimizing with '-O3'. If your gcc is affected (gcc 4.x and 5.x) OpenMesh will fallback to