CS4620/5620: Lecture 16

Programmable Shading and Meshes

Announcements

- Prelim next Thursday
  - In the evening, closed book
  - Including material of this week

Putting it together

- Usually include ambient, diffuse, Phong in one model

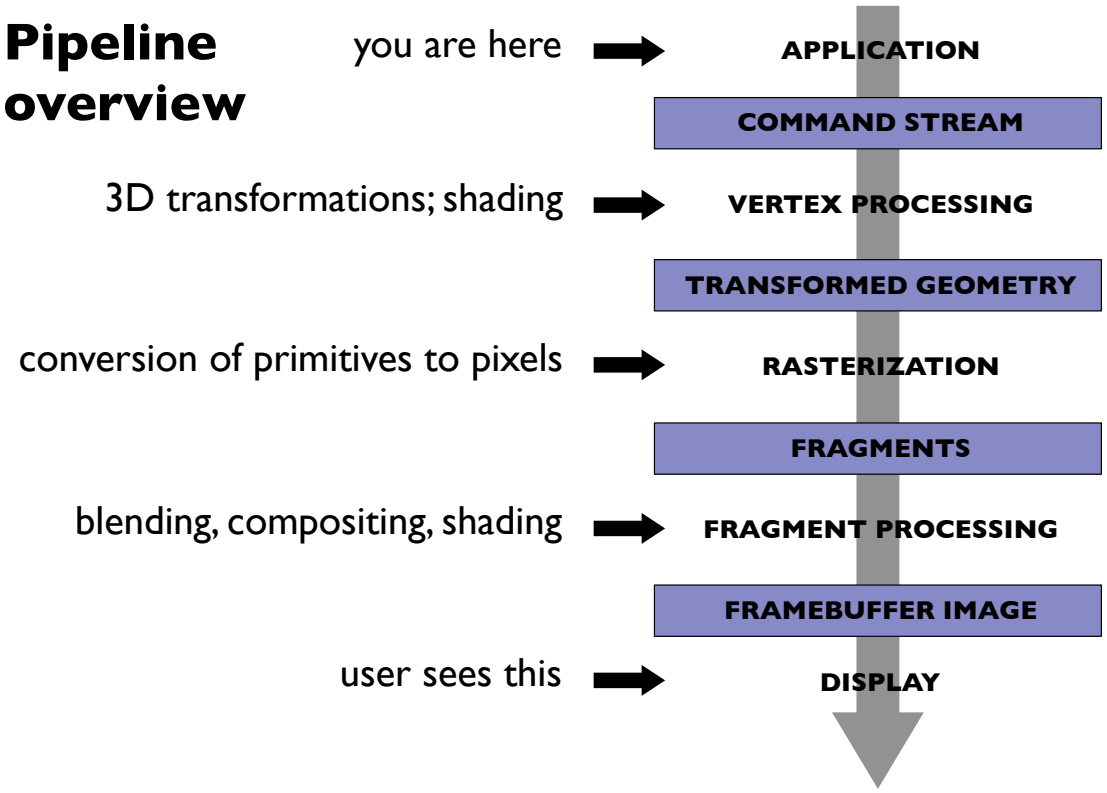
$$\begin{aligned} L &= L_a + L_d + L_s \\ &= k_a\, I_a + k_d\, I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, I \max(0, \mathbf{n} \cdot \mathbf{h})^n \end{aligned}$$

- The final result is the sum over many lights

$$L = L_a + \sum_{i=1}^{N} [(L_d)_i + (L_s)_i]$$

$$L = k_a\, I_a + \sum_{i=1}^{N} [k_d\, I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s\, I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^n]$$

Pipeline overview



Flat shading

- Shade using the real normal of the triangle
- Leads to constant shading and faceted appearance
  - truest view of the mesh geometry

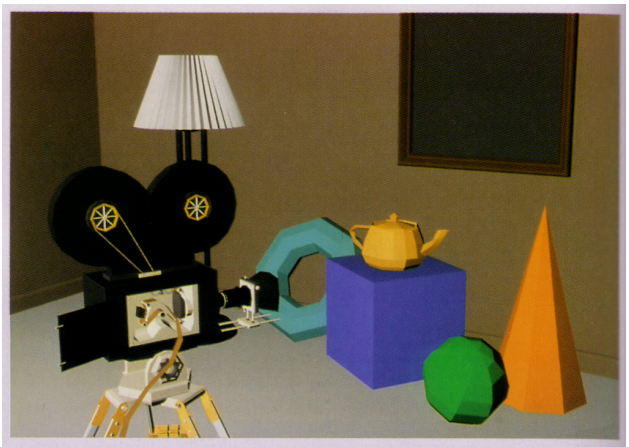


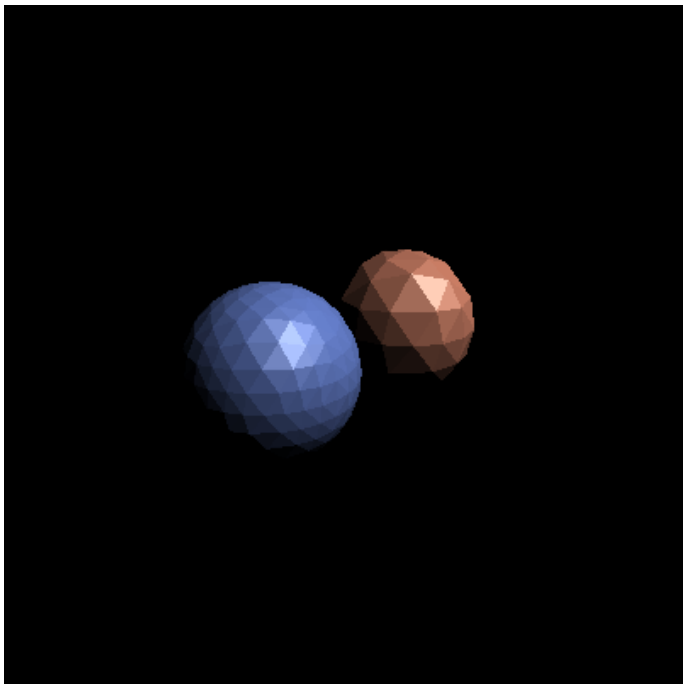
Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Pipeline for flat shading

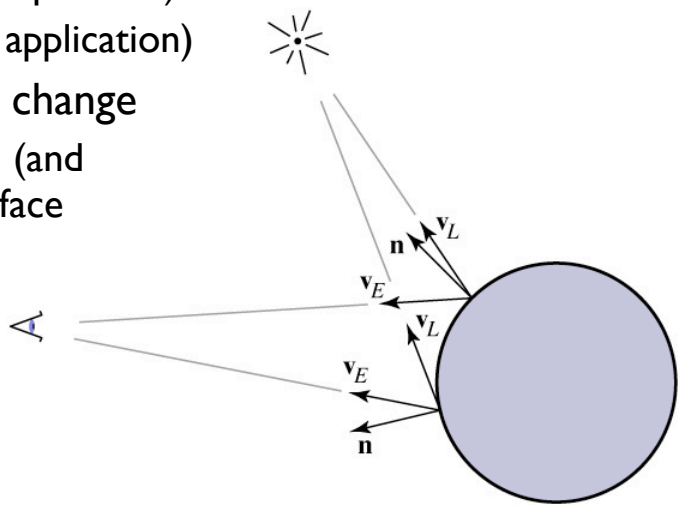
- Vertex stage (input: position / vtx; color and normal / tri)
  - transform position and normal (object to eye space)
  - compute shaded color per triangle using normal
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: z' (screen z)
  - pass through color
- Fragment stage (output: color, z')
  - write to color planes only if interpolated $z' <$ current z'

Result of flat-shading pipeline



Local vs. infinite viewer, light

- Phong illumination requires geometric information:
  - light vector (function of position)
  - eye vector (function of position)
  - surface normal (from application)
- Light and eye vectors change
  - need to be computed (and normalized) for each face

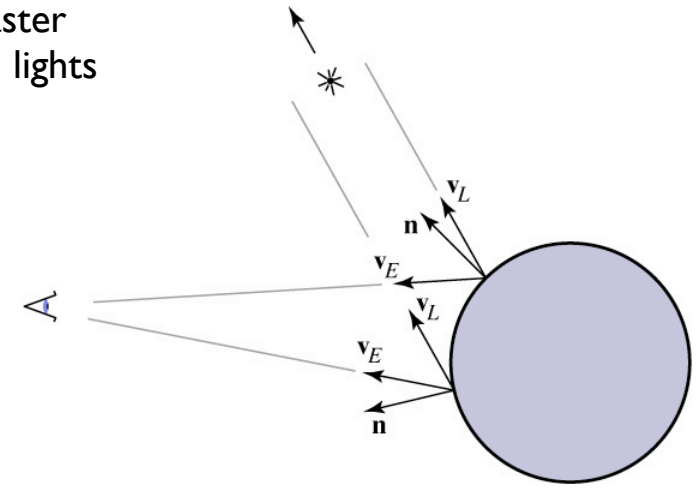


Local vs. infinite viewer, light

- Look at case when eye or light is far away:
  - distant light source: nearly parallel illumination
  - distant eye point: nearly orthographic projection
  - in both cases, eye or light vector changes very little
- Optimization: approximate eye and/or light as infinitely far away

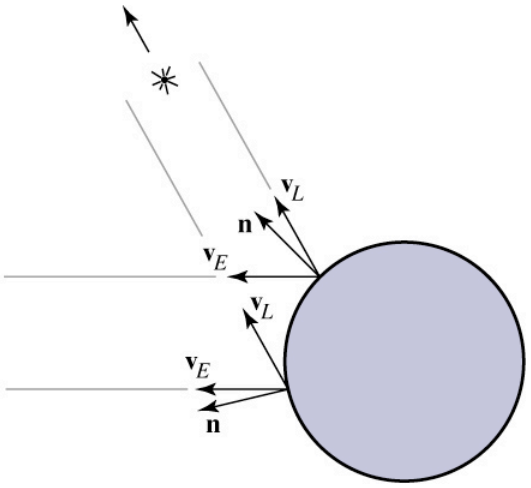
Directional light

- Directional (infinitely distant) light source
  - light vector always points in the same direction
  - often specified by [*x y z* 0]
  - many pipelines are faster if you use directional lights



Infinite viewer

- Orthographic camera
  - projection direction is constant
- "Infinite viewer"
  - even with perspective, can approximate eye vector using the image plane normal
  - can produce weirdness for wide-angle views
  - Blinn-Phong: light, eye, half vectors all constant!



Gouraud interpolation

- Often we're trying to draw smooth surfaces, so facets are an artifact
  - compute colors at vertices using vertex normals
  - interpolate colors across triangles
  - "Gouraud shading"
    - ***Gouraud interpolation***
  - "Smooth shading"
    - ***Phong interpolation***

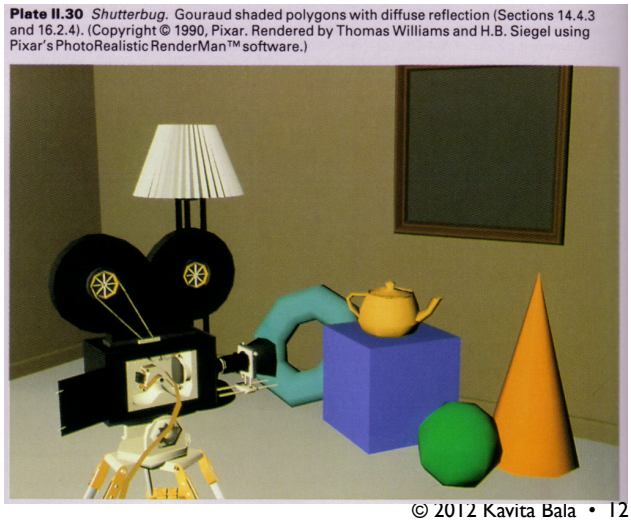


Plate II.30 *Shutterbug.* Gouraud shaded polygons with diffuse reflection (Sections 14.4.3 and 16.2.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

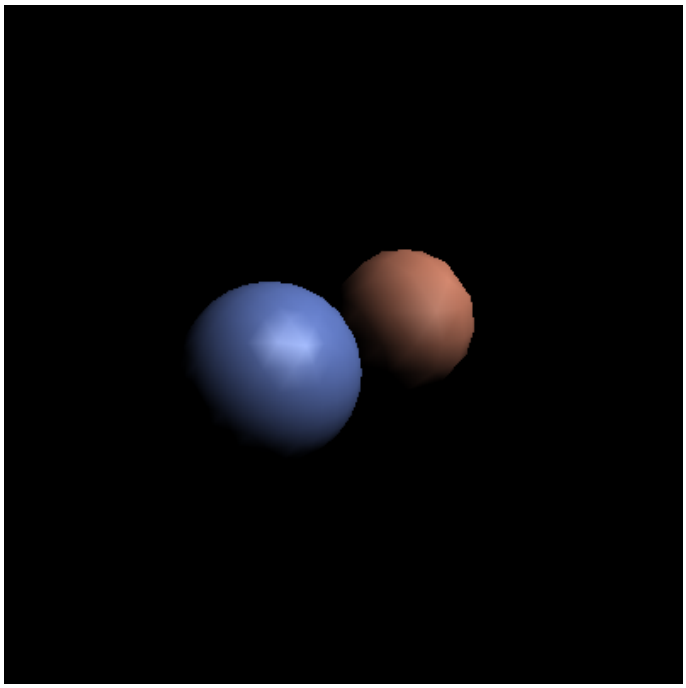
Aside: naming

- Historical
  - Gouraud interpolation, Phong interpolation
    - Different types of smooth shading
  - Phong shading
    - Actually Phong reflectance model (diffuse, specular)
- Bad naming
  - Gouraud shading: not really shading
  - Phong shading: ambiguous
- Correct
  - Gouraud interpolation/shading, per-pixel shading

Pipeline for Gouraud interpolation

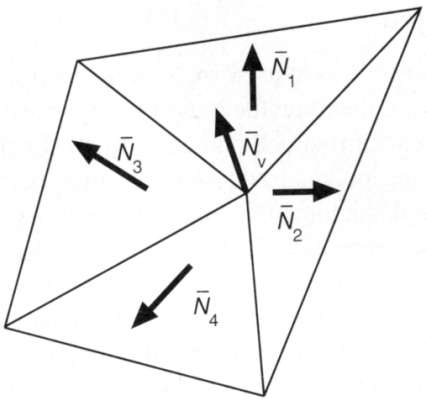
- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - compute shaded color per vertex
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: *z'* (screen *z*); *r, g, b* color
- Fragment stage (output: color, *z'*)
  - write to color planes only if interpolated *z'* < current *z'*

Result of Gouraud shading pipeline



Vertex normals

- Need normals at vertices to compute Gouraud interpolation
- Best to get vtx. normals from the underlying geometry
  - e. g. spheres example
- Otherwise have to infer vtx. normals from triangles
  - simple scheme: average surrounding face normals



[Foley et al.]

$$N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|}$$

Non-diffuse Gouraud interpolation

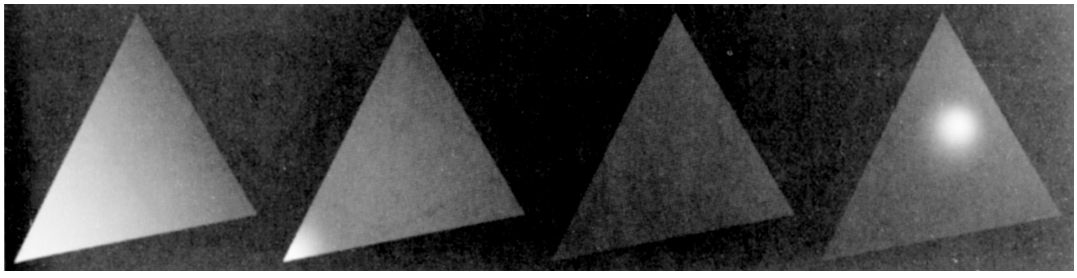
- Can apply Gouraud interpolation to any illumination model
  - it's just an interpolation method
- Results are not so good with fast-varying models like specular ones
  - problems with any highlights smaller than a triangle

Plate II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

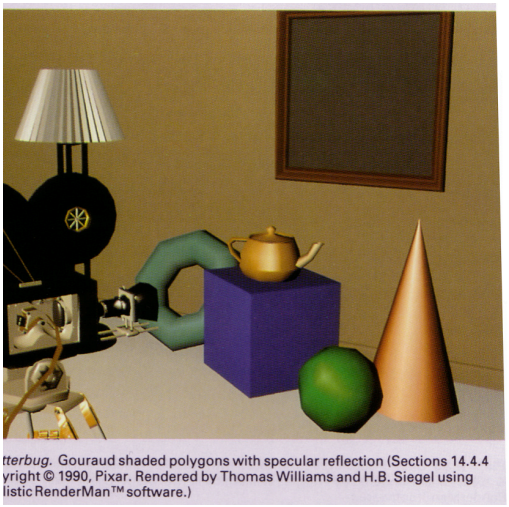
Per-pixel (Phong) interpolation

- Get higher quality by interpolating the normal
  - just as easy as interpolating the color
  - but now we are evaluating the illumination model per pixel rather than per vertex (and normalizing the normal first)
  - in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage



Phong (per-pixel) interpolation

- Bottom line: produces much better highlights



tterbug. Gouraud shaded polygons with specular reflection (Sections 14.4.4 yright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using listic RenderMan™ software.)

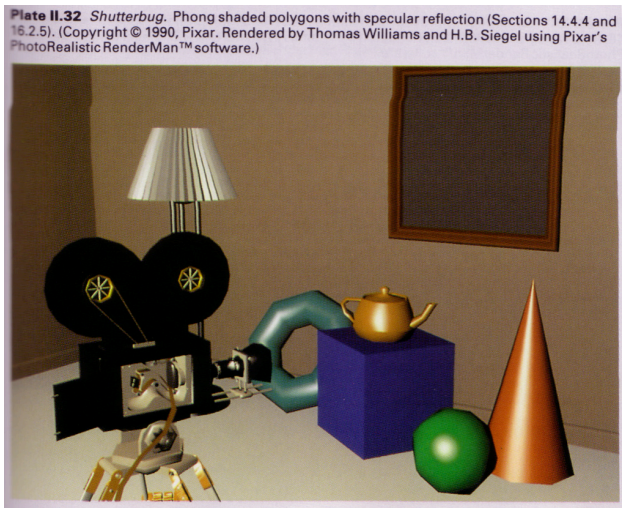


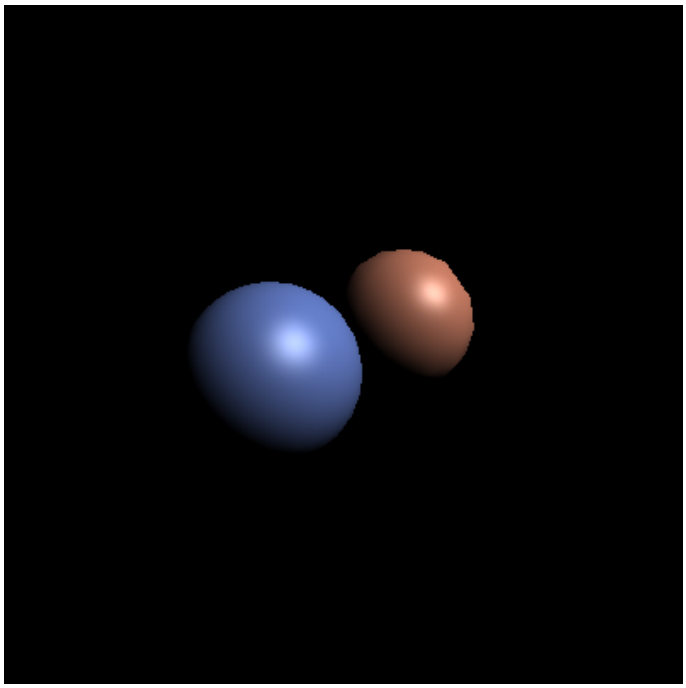
Plate II.32 *Shutterbug.* Phong shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

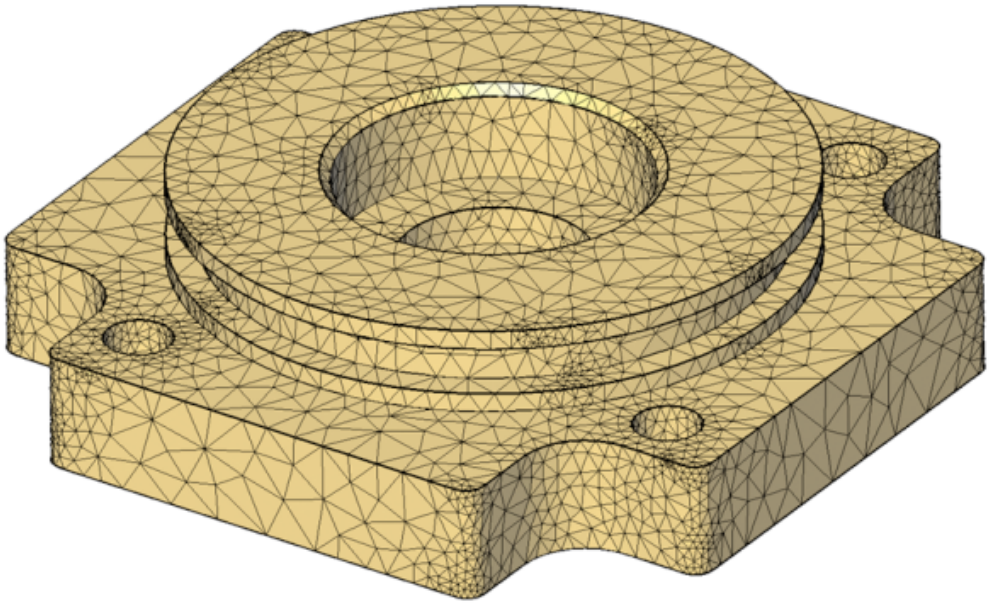
Pipeline for per-pixel (Phong) interpolation

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - transform position (eye to screen space)
  - pass through color
- Rasterizer
  - interpolated parameters: z' (screen z); r, g, b color; x, y, z normal
- Fragment stage (output: color, z')
  - compute shading using interpolated color and normal
  - write to color planes only if interpolated z' < current z'

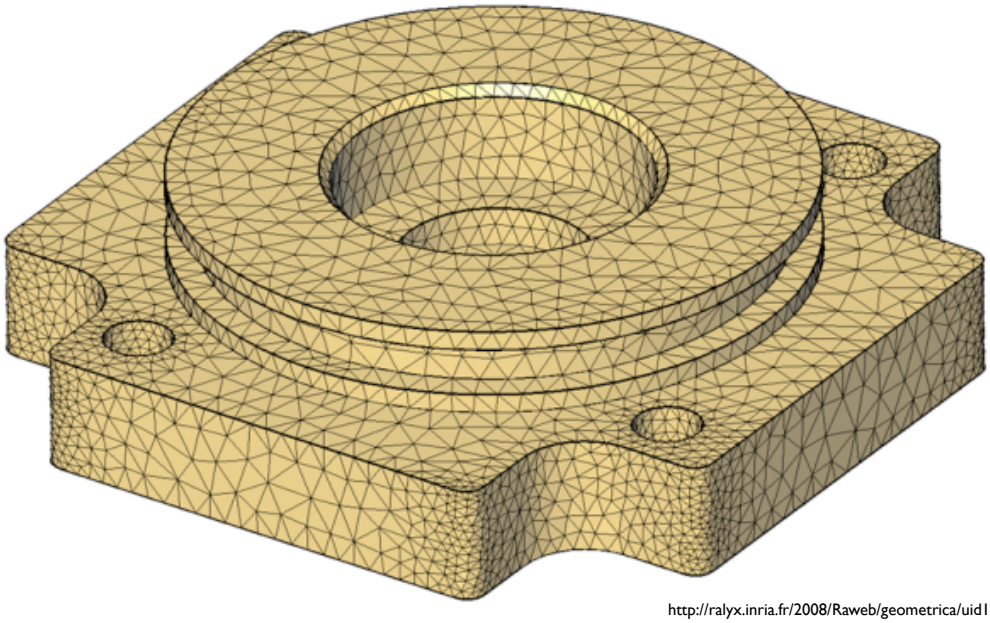
Result of per-pixel shading pipeline



Meshes



http://ralyx.inria.fr/2008/Raweb/geometrica/uid15.html



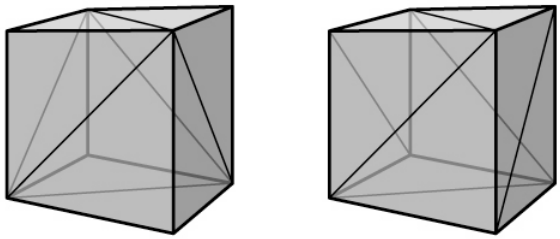
http://ralyx.inria.fr/2008/Raweb/geometrica/uid15.html

Aspects of meshes

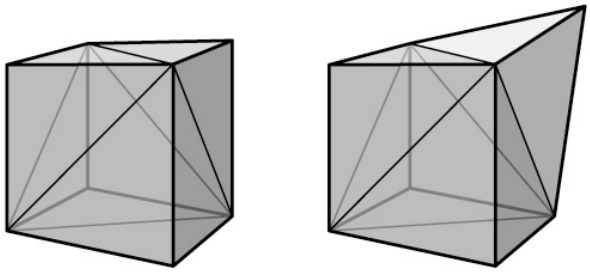
- in many cases we care about the mesh being able to bound a region of space nicely
- in other cases we want triangle meshes to fulfill assumptions of algorithms that will operate on them (and may fail on malformed input)
- two completely separate issues:
  - topology: how the triangles are connected (ignoring the positions entirely)
  - geometry: where the triangles are in 3D space

Topology/geometry examples

- same geometry, different mesh topology:

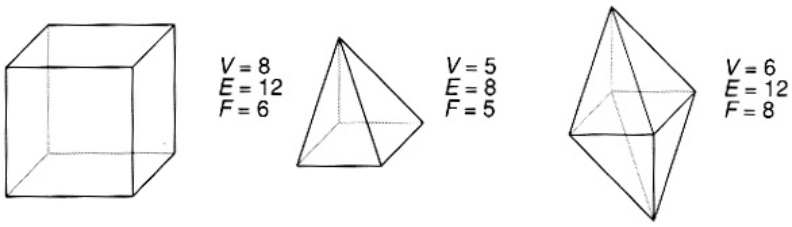


- same mesh topology, different geometry:



Notation

- n_T = #tris; n_V = #verts; n_E = #edges
- Euler: $n_V - n_E + n_T = 2$ for a simple closed surface
  - and in general sums to small integer



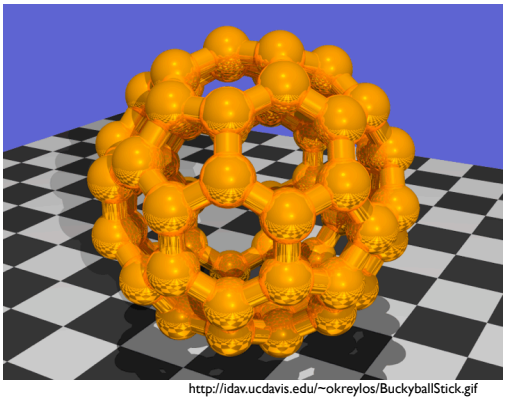
[Foley et al.]

Examples of simple convex polyhedra

Name	Image	Vertices *V*	Edges *E*	Faces *F*	Euler characteristic: *V* − *E* + *F*
Tetrahedron		4	6	4	**2**
Hexahedron or cube		8	12	6	**2**
Octahedron		6	12	8	**2**
Dodecahedron		20	30	12	**2**
Icosahedron		12	30	20	**2**

http://en.wikipedia.org/wiki/Euler_characteristic

Examples of simple convex polyhedra



http://idav.ucdavis.edu/~okreylos/BuckyballStick.gif

Buckyball

V = 60

E = 90

F = 32 (12 pentagons + 20 hexagons)

V - E + F = 60 - 90 + 32 = 2

Examples (nonconvex polyhedra!)

Name	Image	Vertices *V*	Edges *E*	Faces *F*	Euler characteristic: *V* − *E* + *F*
Tetrahemihexahedron		6	12	7	**1**
Octahemioctahedron		12	24	12	**0**
Cubohemioctahedron		12	24	10	**−2**
Great icosahedron		12	30	20	**2**

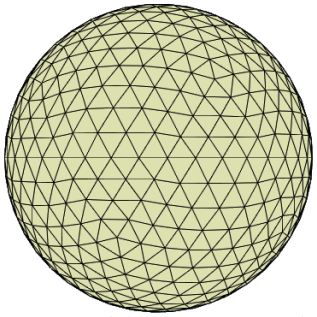
http://en.wikipedia.org/wiki/Euler_characteristic

Euler's Formula

- n_V = #verts; n_E = #edges; n_F = #faces
- Euler's Formula for a convex polyhedron:

$$n_V - n_E + n_F = 2$$

- Other meshes often sum to small integer
  - argument for implication that n_V:n_E:n_F is about 1:3:2



Representation of triangle meshes

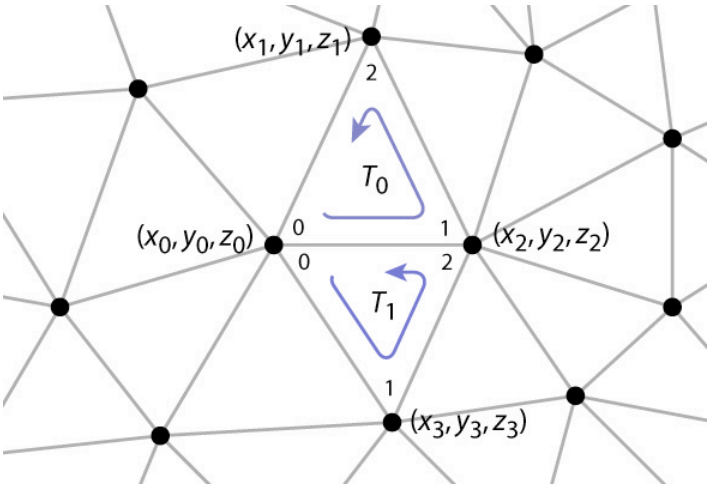
- Compactness
- Efficiency for rendering
  - enumerate all triangles as triples of 3D points
- Efficiency of queries
  - all vertices of a triangle
  - all triangles around a vertex
  - neighboring triangles of a triangle
  - (need depends on application)
    - finding triangle strips
    - computing subdivision surfaces
    - mesh editing

Representations for triangle meshes

- Separate triangles
- Indexed triangle set
  - shared vertices
- Triangle strips and triangle fans
  - compression schemes for transmission to hardware
- Triangle-neighbor data structure
  - supports adjacency queries
- Winged-edge data structure
  - supports general polygon meshes

Separate triangles

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



Separate triangles

- array of triples of points
  - float[n_T][3][3]: about 72 bytes per vertex
    - 2 triangles per vertex (on average)
    - 3 vertices per triangle
    - 3 coordinates per vertex
    - 4 bytes per coordinate (float)
- various problems
  - wastes space (each vertex stored 6 times)
  - cracks due to roundoff
  - difficulty of finding neighbors at all

