EECS 487: Interactive Computer Graphics

Lecture 2:

- Polygonal Mesh
- The Graphics Pipeline: A Grand Tour

Model Representation

Geometric rendering engine (such as OpenGL's) deals only with primitives consisting of points, lines, and polygons

How do you represent:

- curves, and
- curved surfaces

using only lines and polygons?

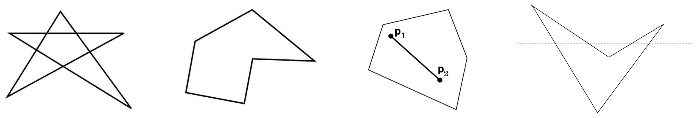
Why limit ourselves to only points, lines, and polygons?

Polygons

What are polygons, edges, and vertices?

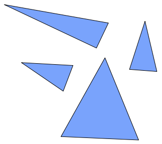
GPU deals only with simple and convex polygons

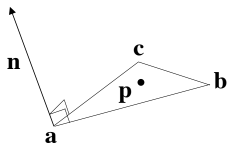
not simple: simple: convex: concave:



Triangles

Triangle is the preferred polygon in CG, why?





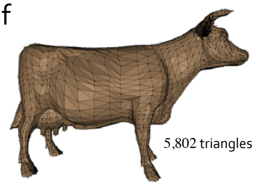
Polygonal Mesh

> “Computer graphics models are like movie sets in that usually only the parts that will be seen are actually built.”
> — *Cook, Carpenter, Catmull*

What is a polygonal mesh?

Ubiquitous in CG because:

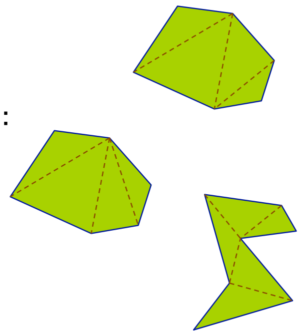
- no restriction on the shape and complexity of object to be modeled
  - volumes bounded by planar surfaces
  - approximate curved surfaces
  - trade off accuracy and speed
    - either closer piecewise linear approximation
    - or less space and lower processing/rendering time
    - accuracy is application dependent: CAD vs. games



- plenty of algorithms and hardware to render visually appealing shaded versions of polygonal objects
  - computers are very good at executing repetitive, simple tasks, fast

Yu

Triangular Mesh



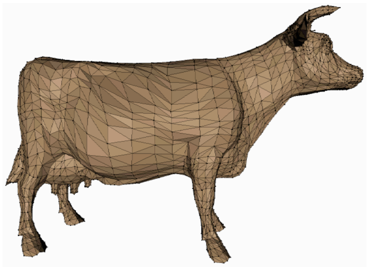
Problems with quadrilateral mesh:

- points may not be planar
- must approximate normal

⇒ just convert it to triangular mesh!
known as triangulation/tessellation

Triangular mesh:

- can convert any planar polygon into exact equivalent in triangles



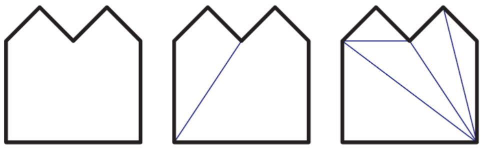
Tessellation

To tessellate: to completely cover a surface, without gaps, using one or more 2D shapes

Reasons to tessellate:

- renderer may handle only convex polygons ⇒ convex partitioning
- many graphics APIs and hardware are optimized for triangles, but polygons may not arrive as triangles ⇒ tessellate (try to avoid long, thin triangles)
- surface may need to be subdivided/meshed to catch shadows and reflected light

Direct3D11 and OpenGL 4.0 have tessellation shader

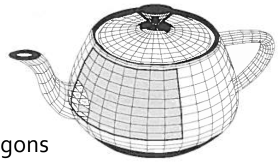


RTR

The Problems with Polygons

Not a very compact representation

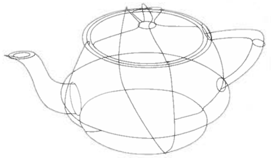
- needs a lot of flat elements to represent smooth or highly detailed surfaces
- accuracy: exactness of representation can only be approximated by increasing the number of polygons
  - if image is enlarged, planar surfaces again become obvious



Intersection test? Inside/outside test?

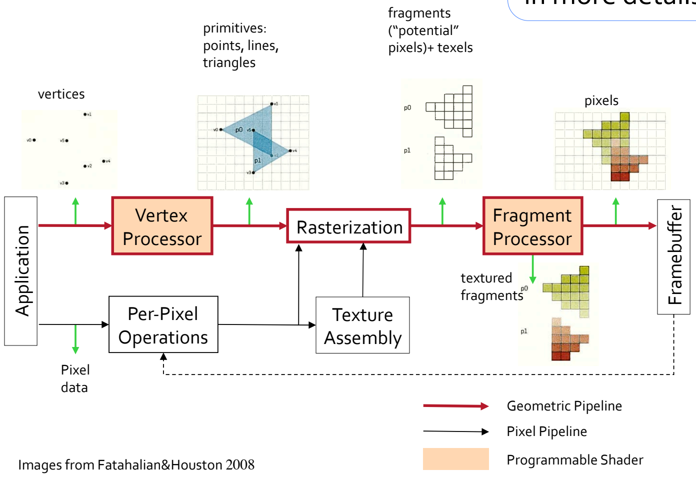
Hard to edit

- creating polygonal objects is straightforward … but laborious and tedious
- how do you edit a polygonal-mesh surface?
  - don’t want to move individual vertices …
- difficult to deform object: a region of low curvature, represented with low polygon count, cannot be deformed into a high curvature region
- it is more a machine representation than a convenient user representation



The Graphics Pipeline: A Grand Tour

Let's look at the Geometric Pipeline in more details



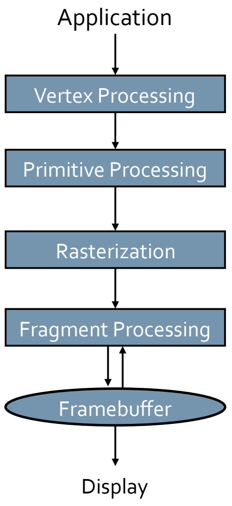
Images from Fatahalian&Houston 2008

The Geometric Pipeline

Application:

Developer has full control of objects and processes in the application space

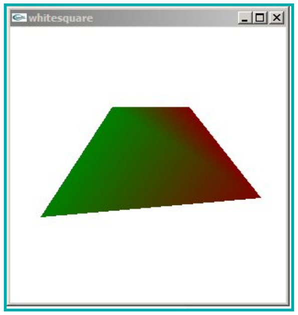
- e.g., runs and controls simulation, collision detection, animation, handles user input
- always executes in software: implementation can be easily changed
- main task:
  - sets graphics parameters
  - feeds geometry and textures into the pipeline
- has to live with what other stages do if not doing shader programming



Akeley/Hanrahan

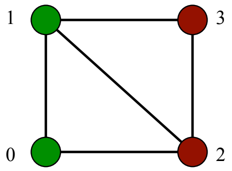
Sample Application

Want:



Send to OpenGL:

```
glBegin(GL_TRIANGLE_STRIP);
  glColor3f(0.0, 0.5, 0.0);  // green
  glVertex3f(0.0, 0.0, 0.0); // vertex 0
  glVertex3f(0.0, 1.0, 0.0); // vertex 1
  glColor3f(0.5, 0.0, 0.0);  // red
  glVertex3f(1.0, 0.0, 0.0); // vertex 2
  glVertex3f(1.0, 1.0, 0.0); // vertex 3
glEnd();
```



Akeley/Hanrahan

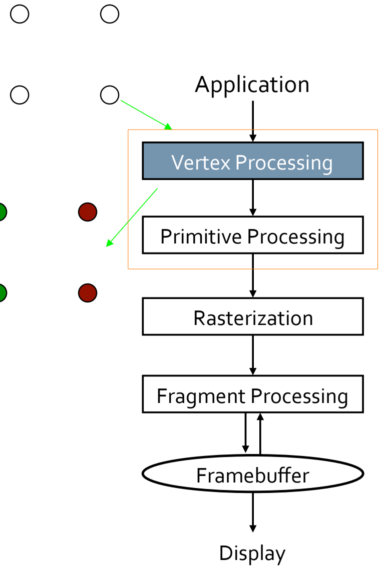
Vertex Processing

Vertex assembly:

- type conversion, e.g., to float
- initialize values, e.g., $z = 0, w = 1$
- initialize state: color, etc.

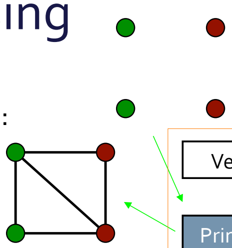
Per-vertex operations:

- model and view transforms
- per-vertex lighting and shading
- compute and transform per-vertex texture coordinate
- lots of floating point operations
  - a scene with a single light requires about 100 floating point ops

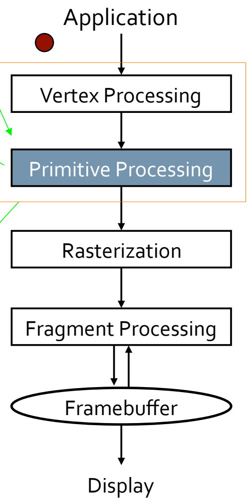


Akeley/Hanrahan

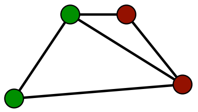
Primitive Processing



Primitive assembly:
- group vertices into primitives:
  - 1 vertex → point
  - 2 vertices → line
  - 3 vertices → triangle
- polygon/quad tessellation
- duplicate vertices in strips or fans



Primitive operations:
- perspective projection
- clipping
- screen mapping
- culling, back-face culling, 2-sided lighting

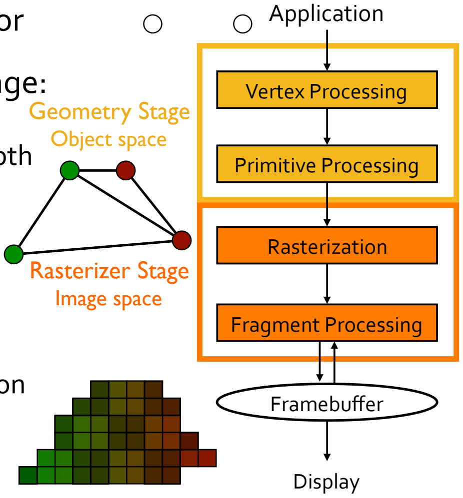


Akeley/Hanrahan

Rasterizer Stage

Goal: assign per pixel color

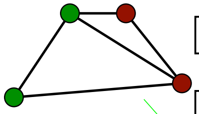
Input from Geometry Stage:
- 2D vertices (in screen coordinates)+depth
- vertex color and texture coordinates

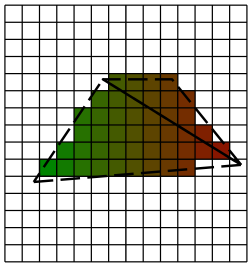


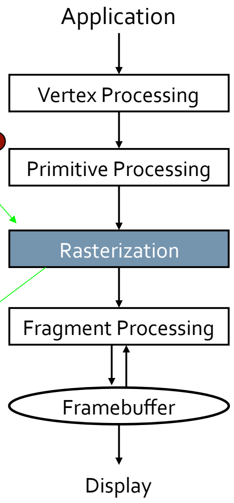
Rasterizer Stage Operations:
- Rasterization/scan conversion
- Texture mapping
- Fragment shading
- Fragment merge

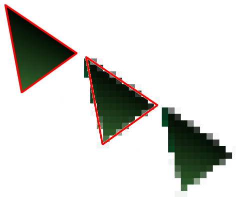
Rasterization/Scan Conversion

Convert triangle into fragments
- discretization
- enumerate covered pixels
- interpolate all values inside the triangle
  - colors
  - texture coordinates
  - depth
- anti-aliasing



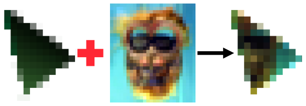






Akeley/Hanrahan

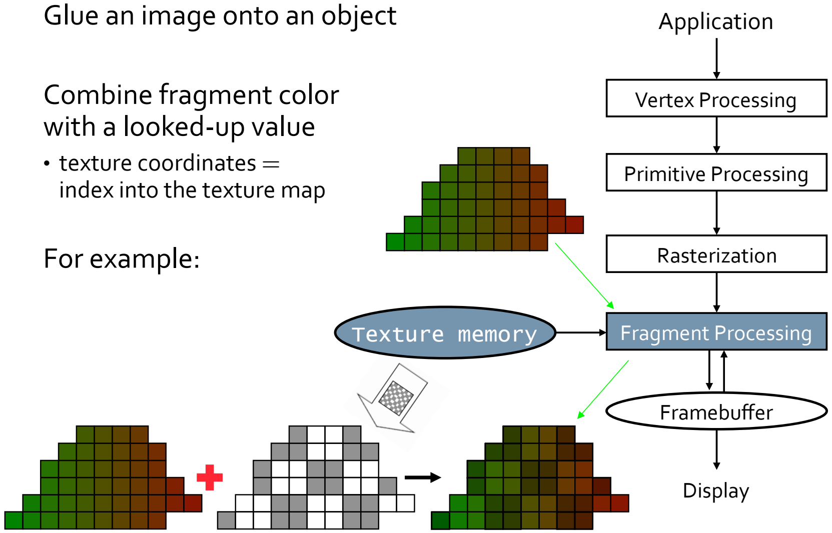
Texture Mapping



Glue an image onto an object

Combine fragment color with a looked-up value
- texture coordinates = index into the texture map

For example:

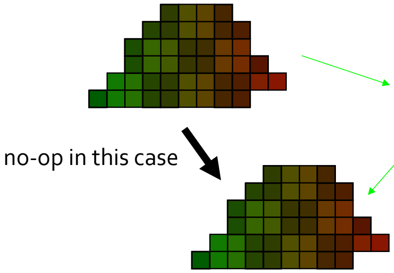


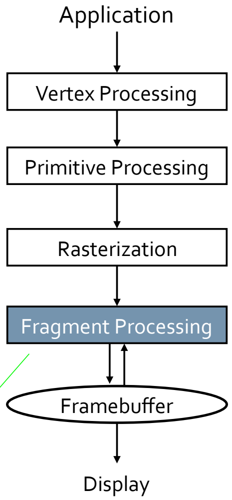
Akeley/Hanrahan

Fragment Shading

Fragment operations:

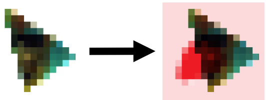
- texture combiners
- per-fragment shading
- fragment tests: owner, scissor, decal, alpha (transparency), fog





Akeley/Hanrahan

Fragment Merge

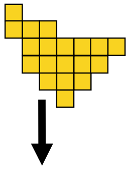


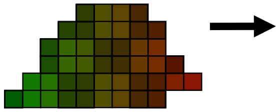
The framebuffer consists of

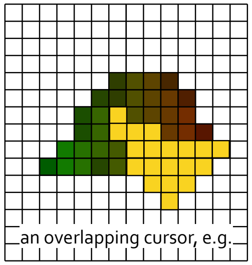
- color buffer (RGBa)
- depth- or z-buffer
- stencil buffer
- (accumulation buffer)

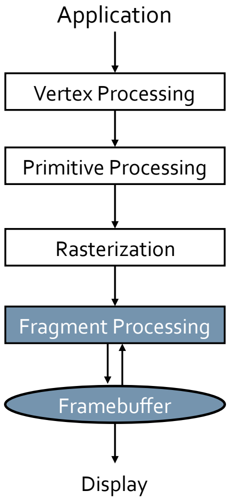
Framebuffer operations:

- blending or compositing
- depth testing (z-buffering)
- shadow maps









Akeley/Hanrahan

Geometric Pipeline

The most common but not only architecture

Vertex processing

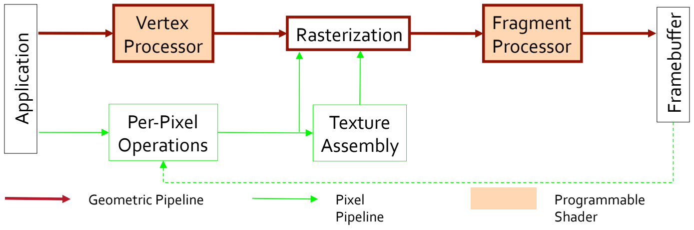
- transformations
- lighting and shading
- projection: 3D to 2D$^{(+\text{depth})}$
- clipping
- primitive assembly

Rasterization

- interpolate values between vertices
- scan conversion

Fragment processing

- texture mapping
- depth test
- alpha test



Alternative Architectures

- The Reyes Architecture:
  - "patches" as primitives, not polygons
  - patches tessellated (diced) into micropolygons
  - multiple fragments (micropolygons) per pixel
  - fragment clipping and visibility after processing
- Tile Architecture:
  - each pipeline handles only a sub-region/tile of a frame
- Frameless Rendering:
  - framebuffer updated at random locations to avoid tearing
- Direct3D 10/OpenGL 3.2 added geometry processor
- Direct3D 11/OpenGL 4.0 added tessellation processor
- Raytracing? Radiosity?