3D graphics pipeline

Vertices → Vertex processor → Clipper and primitive assembler → Rasterizer → Fragment processor → Pixels
Triangle rasterization
Which pixels should be used to approximate a triangle?
Triangle rasterization issues
Which pixels should be used to approximate a triangle?

Who should fill in shared edge?

but who should fill in pixels for a shared edge?
Which pixels should be used to approximate a triangle?

give to triangle that contains pixel center
– but we have some ties
why can’t neither/both triangles draw the pixel?
  neither: gaps
  both: indeterminacy (due to indeterminate drawing order), incorrect, e.g., if both triangles are partially transparent
we want a unique assignment
Which pixels should be used to approximate a triangle?

Use Midpoint Algorithm for edges and fill in?

That could be one possibility but we use a different approach based on barycentric coordinates.
Which pixels should be used to approximate a triangle?

Use an approach based on barycentric coordinates.

For each pixel, we compute its barycentric coordinates. If the coordinates are all \( \geq 0 \), then the pixel is covered by the triangle.
We can interpolate attributes using barycentric coordinates

\[ \mathbf{c} = \alpha \mathbf{c}_0 + \beta \mathbf{c}_1 + \gamma \mathbf{c}_2 \]

Gouraud shading
(Gouraud, 1971)

Using barycentric coordinates also has the advantage that we can easily interpolate colors or other attributes from triangle vertices
Triangle rasterization algorithm

for all $x$ do
  for all $y$ do
    compute $(\alpha, \beta, \gamma)$ for $(x,y)$
    if ($\alpha \in [0, 1]$ and $\beta \in [0, 1]$ and $\gamma \in [0, 1]$) then
      $c = \alpha c_0 + \beta c_1 + \gamma c_2$
      drawpixel($x,y$) with color $c$
for all x do
  for all y do
    compute($\alpha$, $\beta$, $\gamma$) for (x,y)
    if ($\alpha \in [0, 1]$ and $\beta \in [0, 1]$ and $\gamma \in [0, 1]$) then
      $c = \alpha c_0 + \beta c_1 + \gamma c_2$
      drawpixel(x,y) with color c

the rest of the algorithm is to make the steps in red more efficient
Triangle rasterization algorithm

use a bounding rectangle

for $x$ in $[x_{\text{min}}, x_{\text{max}}]$
  for $y$ in $[y_{\text{min}}, y_{\text{max}}]$
    compute $(\alpha, \beta, \gamma)$ for $(x,y)$
    if $(\alpha \in [0, 1] \text{ and } \beta \in [0, 1] \text{ and } \gamma \in [0, 1])$ then
      $c = \alpha c_0 + \beta c_1 + \gamma c_2$
      drawpixel(x,y) with color c
Triangle rasterization algorithm

for x in \([x_{\text{min}}, x_{\text{max}}]\)
  for y in \([y_{\text{min}}, y_{\text{max}}]\)
    \(\alpha = \frac{f_{bc}(x, y)}{f_{bc}(x_a, y_a)}\)
    \(\beta = \frac{f_{ca}(x, y)}{f_{ca}(x_b, y_b)}\)
    \(\gamma = \frac{f_{ab}(x, y)}{f_{ab}(x_c, y_c)}\)
    if (\(\alpha \in [0, 1]\) and \(\beta \in [0, 1]\) and \(\gamma \in [0, 1]\)) then
      \(c = \alpha c_0 + \beta c_1 + \gamma c_2\)
      drawpixel(x,y) with color c

<whiteboard> : computing alpha, beta, and gamma
Triangle rasterization algorithm

Optimizations?

for x in [x_min, x_max]
  for y in [y_min, y_max]
    $\alpha = \frac{f_{bc}(x, y)}{f_{bc}(x_a, y_a)}$
    $\beta = \frac{f_{ca}(x, y)}{f_{ca}(x_b, y_b)}$
    $\gamma = \frac{f_{ab}(x, y)}{f_{ab}(x_c, y_c)}$
    if ($\alpha \in [0, 1]$ and $\beta \in [0, 1]$ and $\gamma \in [0, 1]$) then
      $c = \alpha c_0 + \beta c_1 + \gamma c_2$
      drawpixel(x,y) with color c

1. can make computation of bary. coords. incremental
   - $f(x,y) = Ax+By+C$
   - $f(x+1,y) = f(x,y) + A$
2. color computation can also be made incremental
3. $\alpha > 0$ and $\beta > 0$ and $\gamma > 0$ (if true $\Rightarrow$ they are also less than one)
Triangle rasterization algorithm

Optimizations?

for x in [x_min, x_max]
  for y in [y_min, y_max]
    α = \frac{f_{bc}(x, y)}{f_{bc}(x_a, y_a)}
    β = \frac{f_{ca}(x, y)}{f_{ca}(x_b, y_b)}
    γ = \frac{f_{ab}(x, y)}{f_{ab}(x_c, y_c)}
    if (α ≥ 0 and β ≥ 0 and γ ≥ 0) then
      c = αc_0 + βc_1 + γc_2
    drawpixel(x, y) with color c

1. can make computation of bary. coords. incremental
2. Color computation can also be made incremental
3. \alpha > 0 and \beta > 0 and \gamma > 0 (if true => they are also less than one)
for \( x \) in \([x_{\text{min}}, x_{\text{max}}]\)
for \( y \) in \([y_{\text{min}}, y_{\text{max}}]\)

\[
\alpha = \frac{f_{bc}(x, y)}{f_{bc}(x_a, y_a)}
\]

\[
\beta = \frac{f_{ac}(x, y)}{f_{ac}(x_b, y_b)}
\]

\[
\gamma = \frac{f_{ab}(x, y)}{f_{ab}(x_c, y_c)}
\]

if \( \alpha \geq 0 \) and \( \beta \geq 0 \) and \( \gamma \geq 0 \) then

if \( \alpha > 0 \) or \( f_{bc}(a)f_{bc}(r) > 0 \) and
(\( \beta > 0 \) or \( f_{ca}(b)f_{ca}(r) > 0 \) and
(\( \gamma > 0 \) or \( f_{ab}(c)f_{ab}(r) > 0 \))

\[
c = \alpha c_0 + \beta c_1 + \gamma c_2
\]

drawpixel(x, y) with color \( c \)

- compute \( f_{12}(r), f_{20}(r) \) and \( f_{01}(r) \) and make sure \( r \) doesn't hit a line
Clipping
3D graphics pipeline

Vertices → Vertex processor → Clipper and primitive assembler → Rasterizer → Fragment processor → Pixels
Perspective transformation incorrectly maps vertices behind the eye.
Perspective transformation incorrectly maps vertices behind the eye.

Need to clip before the perspective divide.

Clipping usually takes place in one of two places:
- in world coordinates against the six sides of the view volume, or
- after the 4D transformation but before perspective division (i.e., in homogenous coordinates)
Clip triangle against a plane
Simple pipeline examples

• Simple **2D** pipeline
  • application inputs pixel coordinates, pipeline only does the rasterization phase and overwrites framebuffer contents

• Simple **3D** pipeline
  • viewing transformation (camera, projection, and viewport), followed by rasterization

but how to deal with hidden surfaces?
Hidden Surface Removal
Painter’s algorithm

draw primitives in back-to-front order
Painter’s algorithm

draw primitives in back-to-front order

problem:
triangle intersection

who’s in front of whom?
Painter’s algorithm

draw primitives in back-to-front order

**problem:**
occlusion cycle

also, sorting primitives by depth is slow
Use a z-buffer for hidden surface removal

at each pixel, record distance to the closest object that has been drawn in a depth buffer
Use a **z-buffer** for hidden surface removal

at each pixel, record distance to the closest object that has been drawn in a *depth* buffer

- assume both spheres of the same size, red drawn last
Use a **z-buffer** for hidden surface removal

- **Vertices** → **Vertex processor** → **Clipper and primitive assembler** → **Rasterizer** → **Fragment processor** → **Pixels**

- **without z-buffer**
  - **with z-buffer**

  done in the **fragment blending** phase
  - each fragment must carry a depth
Use a z-buffer for hidden surface removal

http://www.beyond3d.com/content/articles/41/

graphics memory contains the previous color value and associated z value. computing a new color and z value. Compare z-values and if new z value is bigger than old z value, overwrite
Backface culling: another way to eliminate hidden geometry

This is only okay for closed surfaces
Hidden Surface Removal in OpenGL

```c
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

glEnable(GL_DEPTH_TEST);

glEnable(GL_CULL_FACE);
```

For a perspective transformation, there is more precision in the depth buffer for z-values closer to the near plane.
Shading Polygonal Geometry
Smooth surfaces are often approximated by polygons

Shading approaches:

1. Flat
2. Smooth (Gouraud)
3. Phong

each polygon is flat and has a well-defined normal
Flat Shading

do the shading calculation once per *polygon*

valid for light at $\infty$
and viewer at $\infty$
and faceted surfaces

In general, $l$, $n$, and $v$ vary from point to point on a surface. If we assume a distant viewer, $v$ can be thought of as constant. If we assume a distant light source, $l$ can be thought of as constant. For a flat polygon, $n$ is constant.

If the light source or viewer is not at inf, we need heuristic for picking color – e.g., first vertex, or polygon center.
Mach Band Effect

Flat shading doesn’t usually look too good. The lateral inhibition effect makes flat shading seem even worse.
Smooth Shading

We assign the vertex normals based on the surrounding polygon normals

\[ n = \frac{n_1 + n_2 + n_3 + n_4}{||n_1 + n_2 + n_3 + n_4||} \]

Do the shading calculation once per vertex.

We assign the vertex normals based on the surrounding polygon normals.
Interpolating Normals

- Must renormalize
Interpolating Normals

- Must renormalize
Interpolating Normals

- Must renormalize
We can interpolate attributes using barycentric coordinates

\[ c = \alpha c_0 + \beta c_1 + \gamma c_2 \]

Gouraud shading
(Gouraud, 1971)

http://jtibble.dyndns.org/graphics/eecs487/eecs487.html

Using barycentric coordinates also has the advantage that we can easily interpolate colors or other attributes from triangle vertices
Phong Shading

Phong shading requires normals to be interpolated across each polygon -- this wasn’t part of the fixed function pipeline. This can now be done in the pipeline in the fragment shader.

do the shading calculation once per **fragment**
- Phong interpolation looks smoother -- can see edges on the Gouraud model
- but Phong is a lot more work
- both Phong and Gouraud require vertex normals
- both Phong and Gouraud leave silhouettes
Problems with Interpolated Shading

- Polygonal silhouette
- Perspective distortion
- Orientation dependence
- Unrepresentative surface normals

[Foley, van Dam, Feiner, Hughes]
Programmable Shading
Control pipeline through GL state variables

- The application supplies geometric primitives through a graphics API such as OpenGL or DirectX
- control of pipeline operation through state variables only
Programmable Pipeline

Supply shader programs to be executed on GPU as part of pipeline

- can supply shader programs to carry out vertex processing, geometry processing, and pixel processing
Graphics pipeline

- **Vertices**
  - Vertex Generation
  - Vertex stream
  - Vertex Processing
  - Vertex stream

- **Primitives**
  - Primitive Generation
  - Primitive stream
  - Primitive Processing
  - Primitive stream

- **Fragments**
  - Fragment Generation
  - Fragment stream
  - Fragment Processing
  - Fragment stream

- **Pixels**
  - Pixel Operations
  - Output image (pixels)

**Memory Buffers**

- Vertex Data Buffers
- Textures
Phong reflectance in vertex and pixel shaders using GLSL

Vertex Shader (Gouraud interpolation)

```glsl
void main(void)
{
    vec4 v = gl_ModelView_Matrix * gl_Vertex;
    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 l = normalize(gl_lightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = fl_LightSource[0].diffuse;
    vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(h,n)))
            + cl* pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(h,n)));

    gl_FrontColor = color;
    gl_Position = ftransform();
}
```

Pixel Shader (Phong interpolation)

```glsl
void main(void)
{
    vec3 l = normalize(gl_lightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = fl_LightSource[0].diffuse;
    vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(h,n)))
            + cl* pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(h,n)));

    gl_FragColor = color;
}
```

Phong reflectance as a vertex shader
- vertex shaders can be used to move/animate verts
- linear interpolation of vertex lighting

Phong reflectance as a fragment shader
- each fragment is calculated individually – don’t know about neighboring pixles
Programmable shader examples from NVIDIA and ATI