Acknowledgement: The course slides are adapted from the slides prepared by Steve Marschner of Cornell University.

Pipeline and Rasterization

Week 6
The graphics pipeline

• The standard approach to object-order graphics

• Many versions exist
  – software, e.g. Pixar’s REYES architecture
    • many options for quality and flexibility
  – hardware, e.g. graphics cards in PCs
    • amazing performance: millions of triangles per frame

• We’ll focus on an abstract version of hardware pipeline

• “Pipeline” because of the many stages
  – very parallelizable
  – leads to remarkable performance of graphics cards (many times the flops of the CPU at \( \sim 1/5 \) the clock speed)
Pipeline

3D transformations; shading

conversion of primitives to pixels

blending, compositing, shading

user sees this

you are here

APPLICATION

COMMAND STREAM

VERTEX PROCESSING

TRANSFORMED GEOMETRY

RASTERIZATION

FRAGMENTS

FRAGMENT PROCESSING

FRAMEBUFFER IMAGE

DISPLAY
Primitives

• Points
• Line segments
  – and chains of connected line segments
• Triangles
• And that’s all!
  – Curves? Approximate them with chains of line segments
  – Polygons? Break them up into triangles
  – Curved regions? Approximate them with triangles
• Trend has been toward minimal primitives
  – simple, uniform, repetitive: good for parallelism
Rasterization

• First job: enumerate the pixels covered by a primitive
  – simple, aliased definition: pixels whose centers fall inside
• Second job: interpolate values across the primitive
  – e.g. colors computed at vertices
  – e.g. normals at vertices
  – will see applications later on
Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside
Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside
Point sampling

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels
Point sampling

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels
Point sampling in action
Bresenham lines (midpoint alg.)

- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45° lines are now thinner
Bresenham lines (midpoint alg.)

- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45° lines are now thinner
Bresenham lines (midpoint alg.)

- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45° lines are now thinner
Midpoint algorithm in action
Algorithms for drawing lines

- line equation:
  \[ y = b + m x \]
- Simple algorithm:
  evaluate line equation per column
- W.l.o.g. \( x_0 < x_1 \); \( 0 \leq m \leq 1 \)

```plaintext
for x = ceil(x0) to floor(x1)
  y = b + m*x
  output(x, round(y))
```

\[ y = 1.91 + 0.37 \times x \]
Optimizing line drawing

- Multiplying and rounding is slow
- At each pixel the only options are E and NE
- \( d = m(x + 1) + b - y \)
- \( d > 0.5 \) decides between E and NE
Optimizing line drawing

- \[ d = m(x + 1) + b - y \]
- Only need to update \( d \) for integer steps in \( x \) and \( y \)
- Do that with addition
- Known as “DDA” (digital differential analyzer)
Midpoint line algorithm

\[ x = \text{ceil}(x_0) \]
\[ y = \text{round}(mx + b) \]
\[ d = m(x + 1) + b - y \]
while \( x < \text{floor}(x_1) \)
  if \( d > 0.5 \)
    \[ y += 1 \]
    \[ d -= 1 \]
  \[ x += 1 \]
\[ d += m \]
output(\( x, y \))
Linear interpolation

• We often attach attributes to vertices
  – e.g. computed diffuse color of a hair being drawn using lines
  – want color to vary smoothly along a chain of line segments

• Recall basic definition
  – 1D: \( f(x) = (1 - \alpha) y_0 + \alpha y_1 \)
  – where \( \alpha = (x - x_0) / (x_1 - x_0) \)

• In the 2D case of a line segment, alpha is just the fraction of the distance from \((x_0, y_0)\) to \((x_1, y_1)\)
Linear interpolation

- Pixels are not exactly on the line
- Define 2D function by projection on line
  - this is linear in 2D
  - therefore can use DDA to interpolate

\[ \alpha = \frac{v \cdot (q - p_0)}{L} \]
\[ L = v \cdot (p_1 - p_0) \]
Linear interpolation

• Pixels are not exactly on the line
• Define 2D function by projection on line
  – this is linear in 2D
  – therefore can use DDA to interpolate

\[
\alpha = \mathbf{v} \cdot (\mathbf{q} - \mathbf{p}_0) / L \\
L = \mathbf{v} \cdot (\mathbf{p}_1 - \mathbf{p}_0)
\]
Linear interpolation

- Pixels are not exactly on the line
- Define 2D function by projection on line
  - this is linear in 2D
  - therefore can use DDA to interpolate
Alternate interpretation

• We are updating $d$ and $\alpha$ as we step from pixel to pixel
  – $d$ tells us how far from the line we are
  – $\alpha$ tells us how far along the line we are

• So $d$ and $\alpha$ are coordinates in a coordinate system oriented to the line
Alternate interpretation

- View loop as visiting all pixels the line passes through
  - Interpolate $d$ and $\alpha$ for each pixel
  - Only output frag. if pixel is in band
- This makes linear interpolation the primary operation
Pixel-walk line rasterization

\[
x = \text{ceil}(x_0) \\
y = \text{round}(mx + b) \\
d = mx + b - y \\
\text{while } x < \text{floor}(x_1) \\
\quad \text{if } d > 0.5 \\
\quad \quad y += 1; d -= 1; \\
\quad \text{else} \\
\quad \quad x += 1; d += m; \\
\quad \text{if } -0.5 < d \leq 0.5 \\
\quad \quad \text{output}(x, y)
\]
Rasterizing triangles

- The most common case in most applications
  - with good antialiasing can be the only case
  - some systems render a line as two skinny triangles
- Triangle represented by three vertices
- Simple way to think of algorithm follows the pixel-walk interpretation of line rasterization
  - walk from pixel to pixel over (at least) the polygon’s area
  - evaluate linear functions as you go
  - use those functions to decide which pixels are inside
Rasterizing triangles

• Input:
  – three 2D points (the triangle’s vertices in pixel space)
    • \((x_0, y_0); (x_1, y_1); (x_2, y_2)\)
  – parameter values at each vertex
    • \(q_{00}, \ldots, q_{0n}; q_{10}, \ldots, q_{1n}; q_{20}, \ldots, q_{2n}\)

• Output: a list of fragments, each with
  – the integer pixel coordinates \((x, y)\)
  – interpolated parameter values \(q_0, \ldots, q_n\)
**Rasterizing triangles**

- **Summary**
  1. Evaluation of linear functions on pixel grid
  2. Functions defined by parameter values at vertices
  3. Using extra parameters to determine fragment set
Incremental linear evaluation

• A linear (affine, really) function on the plane is:

\[ q(x, y) = c_x x + c_y y + c_k \]

• Linear functions are efficient to evaluate on a grid:

\[ q(x + 1, y) = c_x (x + 1) + c_y y + c_k = q(x, y) + c_x \]
\[ q(x, y + 1) = c_x x + c_y (y + 1) + c_k = q(x, y) + c_y \]
Incremental linear evaluation

linEval(xl, xh, yl, yh, cx, cy, ck) {

    // setup
    qRow = cx*xl + cy*yl + ck;

    // traversal
    for y = yl to yh {
        qPix = qRow;
        for x = xl to xh {
            output(x, y, qPix);
            qPix += cx;
        }
        qRow += cy;
    }
}

\[ c_x = .005; c_y = .005; c_k = 0 \]
(image size 100x100)
Rasterizing triangles

- **Summary**
  1. Evaluation of linear functions on pixel grid
  2. Functions defined by parameter values at vertices
  3. Using extra parameters to determine fragment set
Defining parameter functions

• To interpolate parameters across a triangle we need to find the $c_x, c_y,$ and $c_k$ that define the (unique) linear function that matches the given values at all 3 vertices

  – this is 3 constraints on 3 unknown coefficients:
    
    \[
    \begin{align*}
    c_x x_0 + c_y y_0 + c_k &= q_0 \\
    c_x x_1 + c_y y_1 + c_k &= q_1 \\
    c_x x_2 + c_y y_2 + c_k &= q_2
    \end{align*}
    \]

    (each states that the function agrees with the given value at one vertex)

  – leading to a 3x3 matrix equation for the coefficients:

    \[
    \begin{bmatrix}
    x_0 & y_0 & 1 \\
    x_1 & y_1 & 1 \\
    x_2 & y_2 & 1 \\
    \end{bmatrix}
    \begin{bmatrix}
    c_x \\
    c_y \\
    c_k
    \end{bmatrix}
    =
    \begin{bmatrix}
    q_0 \\
    q_1 \\
    q_2
    \end{bmatrix}
    \]

    (singular iff triangle is degenerate)
Defining parameter functions

• More efficient version: shift origin to \((x_0, y_0)\)

\[
q(x, y) = c_x(x - x_0) + c_y(y - y_0) + q_0
\]

\[
q(x_1, y_1) = c_x(x_1 - x_0) + c_y(y_1 - y_0) + q_0 = q_1
\]

\[
q(x_2, y_2) = c_x(x_2 - x_0) + c_y(y_2 - y_0) + q_0 = q_2
\]

– now this is a 2x2 linear system (since \(q_0\) falls out):

\[
\begin{bmatrix}
(x_1 - x_0) & (y_1 - y_0) \\
(x_2 - x_0) & (y_2 - y_0)
\end{bmatrix}
\begin{bmatrix}
 c_x \\
 c_y
\end{bmatrix}
= \begin{bmatrix}
 q_1 - q_0 \\
 q_2 - q_0
\end{bmatrix}
\]

– solve using Cramer’s rule (see Shirley):

\[
c_x = (\Delta q_1 \Delta y_2 - \Delta q_2 \Delta y_1) / (\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)
\]

\[
c_y = (\Delta q_2 \Delta x_1 - \Delta q_1 \Delta x_2) / (\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)
\]
Defining parameter functions

linInterp(xl, xh, yl, yh, x0, y0, q0, x1, y1, q1, x2, y2, q2) {

    // setup
    det = (x1-x0)*(y2-y0) - (x2-x0)*(y1-y0);
    cx = ((q1-q0)*(y2-y0) - (q2-q0)*(y1-y0)) / det;
    cy = ((q2-q0)*(x1-x0) - (q1-q0)*(x2-x0)) / det;
    qRow = cx*(xl-x0) + cy*(yl-y0) + q0;

    // traversal (same as before)
    for y = yl to yh {
        qPix = qRow;
        for x = xl to xh {
            output(x, y, qPix);
            qPix += cx;
        }
        qRow += cy;
    }
}
Interpolating several parameters

linInterp(xl, xh, yl, yh, n, x0, y0, q0[], x1, y1, q1[], x2, y2, q2[]) {

    // setup
    for k = 0 to n-1
        // compute cx[k], cy[k], qRow[k]
        // from q0[k], q1[k], q2[k]

    // traversal
    for y = yl to yh {
        for k = 1 to n, qPix[k] = qRow[k];
        for x = xl to xh {
            output(x, y, qPix);
            for k = 1 to n, qPix[k] += cx[k];
        }
        for k = 1 to n, qRow[k] += cy[k];
    }
}
Rasterizing triangles

- **Summary**
  1. evaluation of linear functions on pixel grid
  2. functions defined by parameter values at vertices
  3. using extra parameters to determine fragment set
Clipping to the triangle

- Interpolate three *barycentric coordinates* across the plane
  - each barycentric coord is 1 at one vert. and 0 at the other two
- Output fragments only when all three are > 0.
Barycentric coordinates

- A coordinate system for triangles
  - algebraic viewpoint:
    \[ p = \alpha a + \beta b + \gamma c \]
    \[ \alpha + \beta + \gamma = 1 \]
  - geometric viewpoint (areas):
- Triangle interior test:
  \[ \alpha > 0; \beta > 0; \gamma > 0 \]
Barycentric coordinates

- A coordinate system for triangles
  - geometric viewpoint: distances
  - linear viewpoint: basis of edges

\[ \alpha = 1 - \beta - \gamma \]
\[ p = a + \beta(b - a) + \gamma(c - a) \]
Barycentric coordinates

• Linear viewpoint: basis for the plane

– in this view, the triangle interior test is just

\[ \beta > 0; \quad \gamma > 0; \quad \beta + \gamma < 1 \]
Walking edge equations

- We need to update values of the three edge equations with single-pixel steps in $x$ and $y$
- Edge equation already in form of dot product
- Components of vector are the increments
Pixel-walk (Pineda) rasterization

- Conservatively visit a superset of the pixels you want
- Interpolate linear functions
- Use those functions to determine when to emit a fragment
Rasterizing triangles

- Exercise caution with rounding and arbitrary decisions
  - need to visit these pixels once
  - but it’s important not to visit them twice!
Clipping

• Rasterizer tends to assume triangles are on screen
  – particularly problematic to have triangles crossing the plane $z = 0$

• After projection, before perspective divide
  – clip against the planes $x, y, z = 1, -1$ (6 planes)
  – primitive operation: clip triangle against axis-aligned plane
Clipping a triangle against a plane

- 4 cases, based on sidedness of vertices
  - all in (keep)
  - all out (discard)
  - one in, two out (one clipped triangle)
  - two in, one out (two clipped triangles)
Pipeline Operations
Pipeline

Application

Command Stream

Vertex Processing

Transformed Geometry

Rasterization

Fragments

Fragment Processing

Framebuffer Image

Display

3D transformations; shading

Conversion of primitives to pixels

Blending, compositing, shading

User sees this
Pipeline of transformations

- Standard sequence of transforms
Hidden surface elimination

• We have discussed how to map primitives to image space
  – projection and perspective are depth cues
  – occlusion is another very important cue
Back face culling

• For closed shapes you will never see the inside
  – therefore only draw surfaces that face the camera
  – implement by checking $\mathbf{n} \cdot \mathbf{v}$
Back face culling

• For closed shapes you will never see the inside
  – therefore only draw surfaces that face the camera
  – implement by checking $\mathbf{n} \cdot \mathbf{v}$
Back face culling

• For closed shapes you will never see the inside
  – therefore only draw surfaces that face the camera
  – implement by checking $\mathbf{n} \cdot \mathbf{v}$
Back face culling

• For closed shapes you will never see the inside
  – therefore only draw surfaces that face the camera
  – implement by checking $\mathbf{n} \cdot \mathbf{v}$
Painter’s algorithm

• Simplest way to do hidden surfaces
• Draw from back to front, use overwriting in framebuffer
Painter’s algorithm

• Simplest way to do hidden surfaces
• Draw from back to front, use overwriting in framebuffer
Painter’s algorithm

• Simplest way to do hidden surfaces
• Draw from back to front, use overwriting in framebuffer
Painter’s algorithm

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer
Painter’s algorithm

• Simplest way to do hidden surfaces
• Draw from back to front, use overwriting in framebuffer
Painter’s algorithm

• Simplest way to do hidden surfaces
• Draw from back to front, use overwriting in framebuffer
Painter’s algorithm

• Amounts to a topological sort of the graph of occlusions
  – that is, an edge from A to B means A sometimes occludes B
  – any sort is valid
    • ABCDEF
    • BADCFE
  – if there are cycles there is no sort
Painter’s algorithm

- Amounts to a topological sort of the graph of occlusions
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort
Painter’s algorithm

- Useful when a valid order is easy to come by
- Compatible with alpha blending
The z buffer

• In many (most) applications maintaining a z sort is too expensive
  – changes all the time as the view changes
  – many data structures exist, but complex

• Solution: draw in any order, keep track of closest
  – allocate extra channel per pixel to keep track of closest depth so far
  – when drawing, compare object’s depth to current closest depth and discard if greater
  – this works just like any other compositing operation
The z buffer

- another example of a memory-intensive brute force approach that works and has become the standard
Precision in z buffer

• The precision is distributed between the near and far clipping planes
  – this is why these planes have to exist
  – also why you can’t always just set them to very small and very large distances
• Generally use $z'$ (not world z) in z buffer
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space $\neq$ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space $\neq$ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Pipeline for minimal operation

- **Vertex stage** (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- **Rasterizer**
  - pass through color
- **Fragment stage** (output: color)
  - write to color planes
Result of minimal pipeline
Pipeline for basic z buffer

- **Vertex stage** (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color

- **Rasterizer**
  - interpolated parameter: $z'$ (screen $z$)
  - pass through color

- **Fragment stage** (output: color, $z'$)
  - write to color planes only if interpolated $z' <$ current $z'$
Result of z-buffer pipeline
Flat shading

• Shade using the real normal of the triangle
  – same result as ray tracing a bunch of triangles
• Leads to constant shading and faceted appearance
  – truest view of the mesh geometry
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 position $[x \ y \ z \ 0]$
  – many pipelines are faster if you use directional lights
Directional light

• Directional (infinitely distant) light source
  – light vector always points in the same direction
  – often specified by position \([x \ y \ z \ 0]\)
  – 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 shading**

- 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”
  - “Smooth shading”
Gouraud shading

• 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”
  – “Smooth shading”
Pipeline for Gouraud shading

- **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 shading
- 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

\[ N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|} \]
Non-diffuse Gouraud shading

• Can apply Gouraud shading 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
Phong shading

• 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 shading

• Bottom line: produces much better highlights
Pipeline for Phong shading

• 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 Phong shading pipeline