The Graphics Pipeline

Steve Marschner
CS 4620
Cornell University
Two approaches to rendering
Two approaches to rendering

```c
for each object in the scene {
    for each pixel in the image {
        if (object affects pixel) {
            do something
        }
    }
}
```

object order
or
rasterization
Two approaches to rendering

for each object in the scene {
    for each pixel in the image {
        if (object affects pixel) {
            do something
        }
    }
}

object order
or
rasterization

for each pixel in the image {
    for each object in the scene {
        if (object affects pixel) {
            do something
        }
    }
}

image order
or
ray tracing
Two approaches to rendering

for each object in the scene {
    for each pixel in the image {
        if (object affects pixel) {
            do something
        }
    }
}

object order
or
rasterization

for each pixel in the image {
    for each object in the scene {
        if (object affects pixel) {
            do something
        }
    }
}

image order
or
ray tracing

We did this first
Two approaches to rendering

for each object in the scene {
    for each pixel in the image {
        if (object affects pixel) {
            do something
        }
    }
}

object order
or
rasterization

for each pixel in the image {
    for each object in the scene {
        if (object affects pixel) {
            do something
        }
    }
}

image order
or
ray tracing

We did this first

We’ll do this now
Overview

- **Standard sequence of transformations**
  - efficiently move triangles to where they belong on the screen

- **Rasterization**
  - how triangles are converted to fragments

- **Hidden surface removal**
  - efficiently getting the right surface in front

- **Graphics pipeline**
  - the efficient parallel implementation of object-order graphics
The Graphics Pipeline

1. A standard sequence of transformations
object space

world space

eye space

modeling transformation

viewing transformation

projection transformation

viewport transformation

normalized device coordinates
Perspective viewing
The Graphics Pipeline

2. Rasterization
Rasterization

• **First job: enumerate the pixels covered by a primitive**
  – simple definition: pixels whose centers fall inside

• **Second job: interpolate values across the primitive**
  – e.g. colors computed at vertices
  – e.g. normals at vertices
  – e.g. texture coordinates
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

\[
\text{linEval}(x_m, x_M, y_m, y_M, c_x, c_y, c_k) \{
\]

// setup
qRow = \( c_x \cdot x_m + c_y \cdot y_m + c_k \);

// traversal
for y = y_m to y_M {
    qPix = qRow;
    for x = x_m to x_M {
        output(x, y, qPix);
        qPix += c_x;
    }
    qRow += c_y;
}

\[
c_x = .005; \quad c_y = .005; \quad c_k = 0
\]

(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 textbook):

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

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

\[
\text{linInterp}(x_m, x_M, y_m, y_M, x_0, y_0, q_0, \\
x_1, y_1, q_1, x_2, y_2, q_2) \{ \\
\]

// setup
\[
det = (x_1-x_0)*(y_2-y_0) - (x_2-x_0)*(y_1-y_0);
\]
\[
cx = ((q_1-q_0)*(y_2-y_0) - (q_2-q_0)*(y_1-y_0)) / det;
\]
\[
cy = ((q_2-q_0)*(x_1-x_0) - (q_1-q_0)*(x_2-x_0)) / det;
\]
\[
qRow = cx*(x_m-x_0) + cy*(y_m-y_0) + q_0;
\]

// traversal (same as before)
for y = y_m to y_M {
    qPix = qRow;
    for x = x_m to x_M {
        output(x, y, qPix);
        qPix += cx;
    }
    qRow += cy;
}
\]
Interpolating several parameters

```cpp
linInterp(xm, xM, ym, yM, n, x0, y0, q0[],
         x1, y1, q1[], x2, y2, q2[]) {

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

    // traversal
    for y = ym to yM {
        for k = 0 to n, qPix[k] = qRow[k];
        for x = xm to xM {
            output(x, y, qPix);
            for k = 0 to n, qPix[k] += cx[k];
        }
        for k = 0 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**
  - recall each barycentric coord is 1 at one vert. and 0 at the other two

- **Output fragments only when all three are > 0.**
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
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 $x/w, y/w, z/w = 1, -1$ (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)
Rasterizing triangles: edge cases

- **Exercise caution with rotation**
  - need to visit these pixels
  - but it’s important not to
The Graphics Pipeline

3. Hidden surface removal
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
The Graphics Pipeline

4. The rasterization pipeline
The graphics pipeline

- **The standard approach to object-order graphics**
- **Many versions exist**
  - software, e.g. Pixar’s REYES architecture
    - many options for quality, flexibility, scalability
  - 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 workload
  - leads to remarkable performance of graphics cards (many times the flops of the CPU at ~1/2 the clock speed)
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 surfaces? Approximate them with triangles

• Trend over the decades: toward minimal primitives
  – simple, uniform, repetitive: good for parallelism
Programmable shading

You get to control what happens here
Pipeline for minimal operation

- **Vertex stage (input: position / vtx)**
  - transform position (object to screen space)
- **Rasterizer**
  - nothing (extra) to interpolate
- **Fragment stage (output: color)**
  - write a fixed color to color planes
  - (color is a “uniform” quantity that is infrequently updated)
Result of minimal pipeline
Pipeline for basic z-buffer

- **Vertex stage** (input: position / vtx)
  - transform position (object to screen space)
- **Rasterizer**
  - interpolated parameter: $z'$ (screen $z$)
- **Fragment stage** (output: color, $z'$)
  - write fixed color to color planes only if interpolated $z' <$ current $z'$
Result of z-buffer pipeline
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 and normal / vtx)*
  - transform position and normal (object to eye space)
  - compute shaded color per vertex (using fixed diffuse color)
  - 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 \( z' < \text{current } z' \)
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 reflection)
  - problems with any highlights smaller than a triangle
Result of Gouraud shading pipeline
Transforming normal vectors

• **Transforming surface normals**
  – differences of points (and therefore tangents) transform OK
  – normals do not --> use inverse transpose matrix

\[
\begin{align*}
\text{have: } t \cdot n &= t^T n = 0 \\
\text{want: } Mt \cdot Xn &= t^T M^T X n = 0 \\
\text{so set } X &= (M^T)^{-1} \\
\text{then: } Mt \cdot Xn &= t^T M^T (M^T)^{-1} n = t^T n = 0
\end{align*}
\]
Transforming normal vectors

- **Transforming surface normals**
  - differences of points (and therefore tangents) transform OK
  - normals do not --> use inverse transpose matrix

have: \( \mathbf{t} \cdot \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0 \)

want: \( M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T X \mathbf{n} = 0 \)

so set \( X = (M^T)^{-1} \)

then: \( M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T (M^T)^{-1} \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0 \)
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

\[ N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|} \]
Per-pixel (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
Pipeline for per-pixel shading

- **Vertex stage (input: position and normal / vtx)**
  - transform position and normal (object to eye space)
  - transform position (eye to screen space)

- **Rasterizer**
  - interpolated parameters: $z'$ (screen $z$); $x, y, z$ normal

- **Fragment stage (output: color, $z'$)**
  - compute shading using fixed color and interpolated normal
  - write to color planes only if interpolated $z' <$ current $z'$
Result of per-pixel shading pipeline
Modern hardware graphics pipelines are flexible

- programmer defines exactly what happens at each stage
- do this by writing *shader programs* in domain-specific languages called *shading languages*
- rasterization is fixed-function, as are some other operations (depth test, many data conversions, …)

**One example: OpenGL and GLSL (GL Shading Language)**

- several types of shaders process primitives and vertices; most basic is the *vertex program*
- after rasterization, fragments are processed by a *fragment program*
GLSL Shaders

uniform variables

primitives

vertex program

rasterizer

fragment program

framebuffer

vertex attributes

varying parameters

varying parameters

depth

color

application