EECS 487: Interactive Computer Graphics

Lecture 4: GPU Overview
Raster Graphics
Line Rasterization

Pipeline Architecture

• Provides task parallelism, meaning?
• Benefits?
  • assume a 7-stage pipeline, each stage taking time $r$ to complete, how long does it take to complete 7 jobs without pipelining? with pipelining?
• the output of a pipeline is determined by its bottleneck stage
• each stage can use parallel processing (data parallelism)

Unified Pipeline Architecture

Multi-Core GPU with Unified Pipeline Architecture

[Hanrahan09]
Signal Digitization and Sampling

Analog audio signal:

How to digitize the signal?

**Sampling**: reading the signal at certain rate to collect samples

Signal can be reconstructed from the samples

Raster Images

**Pixel** (picture element) ==
- a discrete, point sample of a scene
- the **digitized code values** captured by an addressable photoelement (sensor hardware) [ISO]
  - including intensity, RGB color, depth, etc.

**Image** == sampling of a scene rasterized as a 2D array of pixels

**Raster** == a 2D array of pixels
- indexed \((i, j)\)
- bottom-left pixel is \((0, 0)\)
- top-right is \((N_x - 1, N_y - 1)\)

Raster Graphics Display

Generates and stores raster image in frame buffer
Reads frame buffer contents and turns on pixels
- requires constant update (60-80 Hz)
- most display devices nowadays are **raster display** (as opposed to **vector display**)

Example Raster Display

**LCD**:
- pixels turned on one row at a time via the row electrodes
- individual pixels in the row turned on/off by appropriate voltages on the column electrodes
- active matrix pixels keep their state between updates (resulting in brighter display)
### Raster Graphics Systems

- System Data Bus
  - Copy bits from memory to framebuffer
  - `BitBlt(xo, yo, xn, yn, imagedata);`

![System Diagram]

- CG
- GPU
- Video graphics controller
- Graphics board
- Display

### Pixel Values

- **Three channels:** Red, Green, Blue
  - These three colors are enough to create a rich palette
  - Each channel takes a range of values
    - 24-bit color means
      - 3 bytes → one byte per channel
      - 0...255 for a color
  - 1 byte per channel is enough for display
    - But may not be enough if you want to perform computations
      - Instead, use floating point values 0.0 to 1.0 for computations

**RGBA:** $A (\alpha \in (0,1))$ is to control opacity
- Useful for compositing: $c = \alpha c_{front} + (1-\alpha) c_{back}$

- **Be careful with your float to int conversion**
- **Know when to use floorf(), ceilf(), and rintf()**

### Double Buffering

- **Problem:** Image tearing if framebuffer is only partially updated when the next scan starts

- **Solution:** Double buffering:
  - Render to back buffer...
  - ...while the front buffer is displayed (multiple times if necessary)
  - Swap buffers during vertical blank/retrace interval when back buffer is ready
Completing the Drawing

Issued GL commands may be stuck in buffers along the pipeline, e.g., waiting for more commands to be issued before sending them in batch.

You need to flush all these buffers if you have no more commands to issue to effect execution start:

- `void glFlush(void);`
  - flushes the buffers and start execution of commands
- `void glFinish(void);`
  - waits for commands to finish executing before returning
- `void glutSwapBuffers(void);`
  - swaps back and front buffers if double buffering is in effect (as specified with `glutInitDisplayMode()`), implicitly calls `glFlush();` no effect if single-buffered

Double Buffered GLUT Display Mode

Skeletal `display()` callback for GL window to render a line:

```c
void disp(void)
{
    /* Set color, linewidth etc */
    glBegin(GL_LINES);
    glVertex2f(x0,y0);
    glVertex2f(x1,y1);
    glEnd();
    glutSwapBuffers(); /* swap buffers */
    /* was glFlush() or glFinish();
    glutSwapBuffers() automatically calls glFlush() */
}
```

Rendering a Line

Lines are a basic primitive that must be done well.

In OpenGL: just specify connection type and vertices:

```c
glBegin(GL_LINES);
    glVertex2f(x0,y0);
    glVertex2f(x1,y1);
glEnd();
```

How is it implemented?

Requirements:
- continuous appearance, close to actual continuous line
- uniform thickness and brightness
- fast (line drawing happens a lot!)
Describing a Line and Line Segment

Three ways to describe a line:
- **slope-intercept (or explicit) form**: \( y = mx + b \)
- **implicit** form: \( f(x, y) = mx - y - b = 0 \)
- **parametric** form, using point and vector:
  \[ p(t) = p_0 + t(p_1 - p_0) \]

Given a line segment between \((x_0, y_0)\) and \((x_1, y_1)\), it can be described in slope-intercept form as \( y = mx + b \), where
\[
m = \frac{y_1 - y_0}{x_1 - x_0} \quad \text{and} \quad b = y_0 - mx_0 = y_0 - \left(\frac{y_1 - y_0}{x_1 - x_0}\right)x_0
\]
Or in implicit form: \( f(x, y) = y - \left(\frac{y_1 - y_0}{x_1 - x_0}\right)x - (y_0 - mx_0) = 0 \)

*“implicit” means the equation doesn’t generate points on the line, rather it confirms whether a point is on the line.

Drawing a Line Segment in Raster Graphics

To draw a line from \((1, 2)\) to \((6, 4)\):
- for now assume integer coordinates

Want: thinnest line possible, with no gap, i.e., the pixels must be touching each other, even if only at the corner

Options:
- use the slope-intercept equation
- point sampling: turn on every pixel whose center the line “touches”
- Bresenham midpoint algorithm

Use Slope-Intercept

\[
y = mx + b
\]

// step by \( dx = 1 \)
\[
m = \frac{y_1 - y_0}{x_1 - x_0}
\]
dy = \( \frac{(y_1 - y_0)(x_1 - x_0)}{x_1 - x_0} \)
\[
y = y_0
\]
for \((x = x_0; x < x_1; x++)\) {
  \( \text{set}(x, \text{round}(y)) \);
  \( y += dy \)
}

Use Bresenham

Line Segment and Relative Position

A point at \((x_2, y_2)\) is below the line segment if
\[
\frac{y_2 - y_0}{x_2 - x_0} < \frac{y_1 - y_0}{x_1 - x_0}
\]
or, evaluating the line’s implicit equation at \((x_2, y_2)\) gives:
\[
f(x_2, y_2) = y_2 - \left(\frac{y_1 - y_0}{x_1 - x_0}\right)x_2 - (y_0 - mx_0) < 0
\]

In general, \( f(x, y) > 0 \) describes area above the line, and \( f(x, y) < 0 \) describes area below the line.

Halfspaces

Ideally, it can be discretized rounding, error accumulation, or, with \( |m| > 1 \)
**Point Sampling**

Turn on every pixel whose center falls within the line

Problem: not thinnest line possible

**Midpoint Algorithm**

Thinnest line possible, with no gap, i.e., the pixels must be touching each other, even if only at the corner

**Implicit 2D Lines**

In computing slope \( m \), to avoid dealing with fractional slope (or worse, zero denominator), restate \( f(x, y) \) as:

\[
f(x, y) = (y - mx - b)(x_1 - x_0) = 0
\]

Let: \( C = x_0y_1 - x_1y_0 = \begin{vmatrix} x_0 & x_1 \\ y_0 & y_1 \end{vmatrix} \)

Then \( f(x, y) = Ax + By + C \) or, for \( A = (y_0 - y_1), B = (x_1 - x_0): f(x, y) = Ax + By + C \)
Incremental Midpoint Algorithm

Lines can then be computed incrementally (assume $0 \leq m \leq 1$):

\[
\begin{align*}
    f(x,y) &= \Delta_y y - \Delta_x x + C \\
    f(x+1,y) &= \Delta_y y - \Delta_x (x+1) + C \\
    &= \Delta_y y - \Delta_x x - \Delta_x + C \\
    &= f(x,y) - \Delta_x \\
    f(x+1,y+1) &= f(x,y) + (\Delta_x - \Delta_y)
\end{align*}
\]

and drawn incrementally:

\[
\begin{align*}
y &= y_0; \quad dx = x_1-x_0; \quad dy = y_1-y_0; \\
fmid &= f(x_0+1, y_0+0.5); \\
for \ (x = x_0; x <= x_1; x++) { \\
    set_pixel(x, y); \\
    if (fmid < 0) { // line passes above midpoint \\
        y++; fmid += dx-dy; \\
    } else { fmid -= dy; } \\
}\}
\]

Example: (3,7) to (11,10)

\[
y = y_0; \quad dx = x_1-x_0; \quad dy = y_1-y_0; \\
fmid = f(x_0+1, y_0+0.5); \\
for \ (x = x_0; x <= x_1; x++) { \\
    set_pixel(x, y); \\
    if (fmid < 0) { // line passes above midpoint \\
        y++; fmid += dx-dy; \\
    } else { fmid -= dy; } \\
}\}
\]

Integer Only Line?

Can the line be drawn using only integers (no floats)?

to reduce round-off error and increase performance

\[2f(x,y) = 0\] is a valid description of the line \(f(x,y)\)

hence, the algorithm can be rephrased as:

\[
\begin{align*}
fmid2 &= 2*f(x_0+1, y_0+0.5); // thus no more 0.5 \\
dx2 &= 2*(x_1-x_0); \quad dy2 = 2*(y_1-y_0); \\
for \ (x = x_0; x <= x_1) { \\
    set_pixel(x, y); \\
    if (fmid2 < 0) { y++; fmid2 += dx2-dy2; } \\
    else fmid2 -= dy2; \\
}\}
\]

Be careful with your float to int conversion

Know when to use floorf(), ceilf(), and rintf()

Rasterization implemented in GPU as a fixed function

What if !(0 \leq m \leq 1)?

Case I: $0 \leq m \leq 1$, done
Case II (steep slope): swap the $x$ and $y$ coordinates
Case III (going towards smaller $x$): 
swap the two points
Case IV: swap the points and then swap the $x, y$ coordinates of each point

What to do in case of negative slope?

Midpoint algorithm can be used to draw circle and other conic sections (ellipses, parabolas, hyperbolas)

• exploit symmetry – only need to compute 1 octant
Image Coordinates Conventions

OpenGL (and Labs 1 & 2 & PA1)
- pixel center is at half-integers
- \((0,0)\) at bottom left corner of screen (Cartesian coordinates)

Direct3D 9
- pixel center is at integers
- \((0,0)\) at screen top-left corner (raster scan direction)

Direct3D 10/11
- pixel center is at half-integers
- but \((0,0)\) at screen top-left corner

Parametric $2D$ Lines

Given two points \(p_0 = (x_0, y_0)\) and \(p_1 = (x_1, y_1)\), a line can be described as

\[
\begin{bmatrix}
x(t) \\
y(t)
\end{bmatrix} = \begin{bmatrix}
x_0 + t(x_1 - x_0) \\
y_0 + t(y_1 - y_0)
\end{bmatrix}
\]

in parametric form: \(p(t) = p_0 + t(p_1 - p_0)\)

or: \(p(t) = (1 - t)p_0 + t p_1\)

or as a point \(o\) and a vector \(d\): \(p(t) = o + td\)

Linear, Affine, Convex Combinations

Given points \(p_1, p_2, p_3, \ldots, p_n\) and coefficients \(\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n\):
- linear combination:
  \[\alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3 + \ldots + \alpha_n p_n\]
- affine combination:
  linear combination and \(\alpha_1 + \alpha_2 + \alpha_3 + \ldots + \alpha_n = 1\)
- convex combination:
  affine combination and each \(\alpha_i \geq 0\)

What is described by this set of points in 3D?
\[\alpha p + (1 - \alpha)q,\] where \(0 \leq \alpha \leq 1\)

Parametric Line Drawing

\(p(t) = (1 - t)p_0 + t p_1\): is a convex combination (interpolation) of two points:
- any convex combination of two points lies on the straight line segment between the points

\[\begin{align*}
(1 - t)p_0 + t p_1
\end{align*}\]

- a remarkably general concept, quite useful for blending
- interpolation of: positions, colors, vectors
Coloring Thick Lines

How to interpolate the color of each pixel on a thick (multi-pixel width) line?

1. Compute how far the pixel is from both endpoints
2. How to compute?
   • project pixel center onto the line to find parameter $t$ of the parametric line equation
3. Correctly blended color can then be computed based on $t$

Dot Product Review

The dot product of two vectors $\mathbf{u}$ and $\mathbf{v}$ is a scalar value $\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\|\cos \theta \Rightarrow \cos \theta = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|}$

Another way to compute $\mathbf{u} \cdot \mathbf{v}$:

$$\mathbf{u} \cdot \mathbf{v} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \mathbf{u}^T \mathbf{v} = u_0v_0 + u_1v_1 + u_2v_2$$

If $\theta = 90^\circ$ ($\mathbf{u} \perp \mathbf{v}$) then $\mathbf{u} \cdot \mathbf{v} = 0$

If $\mathbf{u}$ is normalized, i.e., $\|\mathbf{u}\| = 1$, $\mathbf{u} \cdot \mathbf{u} = 1$

$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$ (commutative)

$(\mathbf{u} + \mathbf{v}) \cdot \mathbf{w} = \mathbf{u} \cdot \mathbf{w} + \mathbf{v} \cdot \mathbf{w}$ (distributive)