CMPSCI 473: Introduction to Computer Graphics (Fall 2010)

Lecture 12: Rasterization, Fractals

Rui Wang
3D and Viewing Transformations

3D Geometric Primitives

- Model Transformation

- Camera Transformation

- Viewing (projection) Transformation

- Rasterization

...
3D and Viewing Transformations

3D Geometric Primitives

- Model Transformation
- Camera Transformation
- Viewing (projection) Transformation

ModelViewMatrix

ProjectionMatrix

- glTranslatef
- glRotatef
- glScalef
- ...
- gluLookAt
- gluPerspective

...
Rasterization

- Generate pixels covered by geometric primitives
- Draw image pixels
Overview

- **Rasterization**
  - Draw bitmap images
  - Generate pixels covered by points, lines, circles, triangles, polygons etc.
  - Flood Fill

- **Fractals**
  - Koch Curve
  - Sierpinski triangle
  - Definition of ‘dimension’
Drawing Points

3D Geometric Primitives

Modelview Transformation

Viewing (projection) Transformation

Rasterization

$\begin{pmatrix} x, y, z, 1 \end{pmatrix}$

3D

$\begin{pmatrix} p_x, p_y, p_z \end{pmatrix}$

2D

$\begin{pmatrix} p_x, p_y, p_z \end{pmatrix}$

$\text{Pixels}(\lfloor p_x \rfloor, \lfloor p_y \rfloor) = \text{Color}$
Drawing Lines

\[(x_1, y_1, z_1, 1)\]

\[(p_{x1}, p_{y1})\]

\[(x_2, y_2, z_2, 1)\]

\[(p_{x2}, p_{y2})\]

\[\text{?}\]
Method 1: Use Line Equations

\[
\begin{bmatrix}
  x \\
  y
\end{bmatrix} = \begin{bmatrix}
  (1-t) \cdot x_1 + t \cdot x_2 \\
  (1-t) \cdot y_1 + t \cdot y_2
\end{bmatrix}
\]
Method 1: Use Line Equations

\[
\begin{bmatrix}
  x \\
  y \\
\end{bmatrix} = \begin{bmatrix}
(1-t) \cdot x_1 + t \cdot x_2 \\
(1-t) \cdot y_1 + t \cdot y_2 \\
\end{bmatrix}
\]

Under the assumption \(-1<\text{slope}<1\)

- Line(int x1, int y1, int x2, int y2)
  
  \{ 
  for x = x1 to x2 do 
    t = (x - x1) / (x2 - x1);
    y = y1 + t \cdot (y2 - y1);
    Draw(x, round(y));
  \}

DDA (Digital Differential Analyser)

```c
void Line(int x1, int y1, int x2, int y2)
{
    int dx = x2 - x1, dy = y2 - y1;
    int n = max(abs(dx), abs(dy));
    float dt = n, dxdt = dx / dt, dydt = dy / dt;
    float x = x1, y = y1;

    while (n--)
    {
        DrawPoint( round(x), round(y) );
        x += dxdt;
        y += dydt;
    }
}
```
How to improve it?

- Get rid of all nasty floating point operations
- The idea: find the next pixel from the current

Under the assumption $0 < \text{slope} < 1$

- We’re only ever going to either go right one pixel, or up and right one pixel.
- So which of the green pixels is next?
The Key

- We’re only ever going to either go right one pixel, or up and right one pixel.
- Call these two choices “E” and “NE”
- Let’s think of pixels as “lattice points” on a grid. Given a point \((x,y)\), we only need to choose between \(y\) and \(y+1\)
The Key

\[ y \]

\[ y+1 \]
The Midpoint Test

- The diagram illustrates a line graph with points labeled with expressions like y+1, y+2, y+3, x, x+1, x+2, etc. The line passes through these points, indicating a linear relationship.

- The graph helps in understanding the midpoint test, which is a technique used to determine the midpoint of a line segment on a coordinate plane.
The Midpoint Test

- Look at the vertical grid line that our line intersects
- See if the midpoint \((y + 1/2)\) falls above or below the line?
- If the midpoint is below the line, go NE; otherwise go E
How to check?

Need a way to check whether a point lies above or below a line.
How to check?

- Need a way to check whether a point lies above or below a line.
- Use line's implicit equation
  - We know a line can be defined as
    \[ f(x, y) = Ax + By + C = 0 \]
  - If there is a point \((xx, yy)\) that makes \(f(xx, yy) > 0\), we say it's above (on the left side) of the line.
  - Similarly, a point \((xx, yy)\) that makes \(f(xx, yy) < 0\) is below (on the right side) of the line.
The Midpoint Test

□ Step 0. Our line equation:

\[ f(x, y) = (y_1 - y_2)x + (x_2 - x_1)y + (x_1y_2 - x_2y_1) = 0 \]

□ Step 1. Initialize

■ We know that

\[ f(x_1, y_1) = 0 \]

■ Let’s look at its mid point on the right side

\[ d = f(x_1 + 1, y_1 + 0.5) = A + \frac{B}{2} \]
The Midpoint Test

- **Step 2. Update**
  - If $d < 0$, go NE, $x++$, $y++$
  - If $d \geq 0$, go E, $x++$, $y$ remains the same

  We then need to update $d$ for the next midpoint. Now observe that (above)
The Midpoint Test

- **Step 2. Update**
  - If $d < 0$, go NE, $x++$, $y++$

  Next mid-point: $f(x_1 + 2, y_1 + 1.5)$

  - If $d \geq 0$, go E, $x++$, $y$ remains the same

  Next mid-point: $f(x_1 + 2, y_1 + 0.5)$

  - We then need to update $d$ for the next midpoint. Now observe that (above)
The Midpoint Test

- **Step 2. Update**
  - If $d < 0$, go NE, $x++$, $y++$
    \[ f(x_1 + 2, y_1 + 1.5) - f(x_1 + 1, y_1 + 0.5) = A + B \]
  - If $d \geq 0$, go E, $x++$, $y$ remains
    \[ f(x_1 + 2, y_1 + 0.5) - f(x_1 + 1, y_1 + 0.5) = A \]
  - We then need to update $d$ for the next midpoint. Now observer that (above)
The Midpoint Test

☐ Step 2. Update
  ■ If \( d < 0 \), go NE, \( x++ \), \( y++ \)

\[
d+ = (A + B)
\]

■ If \( d \geq 0 \), go E, \( x++ \), \( y \) remains

\[
d+ = A
\]

☐ In fact, **only the sign of \( d \) matters!**
Midpoint Line Drawing – Summary

☐ Step 1. Initialize, \( x=x_1, y=y_1 \)

\[
d = 2A + B
\]

☐ Step 2. Update

■ If \( d < 0 \), go NE, \( x++, y++ \)

\[
d+ = 2(A + B)
\]

■ If \( d \geq 0 \), go E, \( x++, y \) remains

\[
d+ = 2A
\]
Midpoint Line Drawing

Line(int x1, int y1, int x2, int y2) {
    int B = x2 - x1, A = y1 - y2;
    int d = 2 * A + B;
    int incrE = 2 * A, incrNE = 2 * (A + B);
    int x = x1, y = y1;
    DrawPoint( x, y );
    while (x < x2) {
        x++;
        if (d <= 0) { y++; e += incrNE; }
        else { e += incrE; }
        DrawPoint( x, y );
    }
}
Drawing Circles

- Only considers an octant (the others are symmetric)
- Midpoint still works: either go right, or go right and down
The Circle’s Implicit Function:

\[ f(x, y) = x^2 + y^2 - r^2 = 0 \]

- \( f(x,y) > 0 \) means a point \((x,y)\) is outside of the circle;
- \( f(x,y) < 0 \) means the point is inside the circle;
- Of course, \( f(x,y) = 0 \) means the point is on the circle.
Midpoint Circle Drawing

- The Circle’s Implicit Function:
  \[ f(x, y) = x^2 + y^2 - r^2 = 0 \]

- Once we know the value of the implicit function at one midpoint, we can get the value at the next midpoint by the same differencing technique:
  - If going E: \[ f(x + 2, y - 0.5) - f(x + 1, y - 0.5) = 2x + 3 \]
  - If going SE: \[ f(x + 2, y - 1.5) - f(x + 1, y - 0.5) = 2(x - y) + 5 \]
Increments are no longer constants
However, we can incrementally update the increments (2^{nd} order difference)!

- If going E:
  
  \[ x++; \]
  
  \[ d+ = incE; \]
  
  \[ incE+ = 2; \]
  
  \[ incSE+ = 2; \]

- If going SE:
  
  \[ x++; \]
  \[ y--; \]
  
  \[ d+ = incSE; \]
  
  \[ incE+ = 2; \]
  
  \[ incSE+ = 4; \]
```c
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    double d = 5.0 / 4.0 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0) /* Select E */
            d += 2.0 * x + 3.0;
        else /* Select SE */
            d += 2.0 * (x - y) + 5.0;
        y--;
    }
    x++;
    CirclePoints (x, y, value);
} /* while */
} /* MidpointCircle */
```
Drawing Triangles

\[(x_1, y_1, z_1, 1)\]

\[(p_{x1}, p_{y1})\]

\[(x_2, y_2, z_2, 1)\]

\[(p_{x2}, p_{y2})\]

\[(x_3, y_3, z_3, 1)\]
void ScanTriangle(Triangle T, Color rgba)
{
    for each pixel P within bounding box {
        if (Inside(P, T))
            SetPixel(x, y, rgba);
    }
}
Drawing Triangles – Method I

- Color pixels inside a triangle

```c
void ScanTriangle(Triangle T, Color rgba){
    for each pixel P within bounding box {
        if (Inside(P, T))
            SetPixel(x, y, rgba);
    }
}
```

- But how do you check inside?

![Diagram of a triangle with points P1, P2, and P3]
Drawing Triangles – Method I

- Check point inside triangle:
  - Triangles are formed by three lines
  - A point is inside a triangle if it’s on the left side of all boundary lines (assuming triangle vertices are defined in CW order)
Edge testing method only works for convex polygons.
Drawing Polygons

- Inside polygon rule

- Concave
- Self-Intersecting
- With Holes
Inside Polygon Rule

- Odd-parity rule
  - Any ray from P to infinity crosses odd number of edges

- Concave
- Self-Intersecting
- With Holes
Drawing Triangles – Sweep Line

- Take advantage of line drawing and spatial coherence
  - Keep track of pixels on the vertical scan-lines
  - Fill inside pixels using vertical scan-lines
Hardware Scan Conversion

- Convert everything into triangles
  - Scan convert the triangles
Rasterization for Fun

☐ Demo
Assignment 3

☐ Due Thursday next week.
☐ Hierarchical modeling.
Fractals

- Fractal: any curve or shape exhibiting self-similarity at any scale
- Fractals appear in nature all the time
- Part of the general science of chaos
The Koch Curve

☐ Start with a line
☐ Replace the line with four lines, and repeat:

What’s the length of the curve?
The Koch Curve

☐ Start with a line
☐ Replace the line with four lines, and repeat:

What’s the length of the curve?
A: Infinity.
The Koch Snowflake

What’s the area of the shape?
The Koch Snowflake

What’s the area of the shape?

\[ \frac{8}{5} A \]

So it’s an boundary of infinite length enclosing a finite area.
Sierpinski triangle

What’s the area of the shape?
What’s the area of the shape?
A: Zero!
Let’s look at familiar “self-similar” shapes:

<table>
<thead>
<tr>
<th>Shape</th>
<th>#Parts</th>
<th>Scale</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Line" /></td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><img src="image" alt="Square" /></td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><img src="image" alt="Cube" /></td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><img src="image" alt="Fractal" /></td>
<td>4</td>
<td>3</td>
<td>log₃ 4 ≈ 1.26</td>
</tr>
</tbody>
</table>
Fractal Dimensions

- Let’s look at familiar “self-similar” shapes.
- Definition of Fractal Dimension:

\[ \text{Dimension} = \log_{\text{Scale Ratio}} \text{Part} \]
More Siepinski Fractals
More Siepinski Fractals