



Chapter 6 Hidden-Line Elimination

- Line Segments and Triangles
- Tests for Visibility
- Input Format of 3D Objects
 - Holes and Invisible Line Segments
 - Individual Faces and Line Segments
- Automatic Generation of Input Files
- HP-GL Output Format
- Implementation

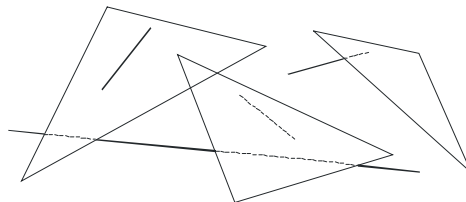
©2006 Wiley & Sons

1



Line Segments and Triangles

- A line drawing = set of line segments + set of faces.
- We need to remove invisible line segments.
- Faces can be triangulated into triangles.
- Hidden-Line Elimination becomes testing “**whether each line segment is blocked by one or more triangles**”

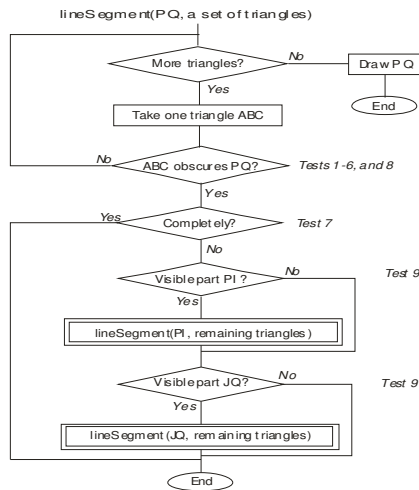


- A line on an edge is considered visible

©2006 Wiley & Sons

2

Tests for Visibility

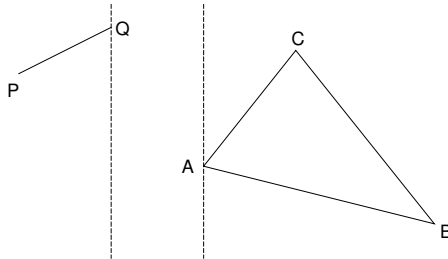


©2006 Wiley & Sons

3

Tests for Visibility (cont'd)

- **Test 1:** Minimax – both P & Q on the left or right of ABC



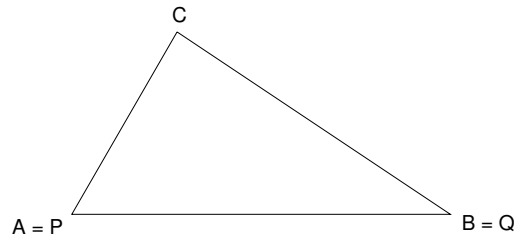
©2006 Wiley & Sons

4



Tests for Visibility (cont'd)

- **Test 2:** PQ is one of the triangle edges of ABC



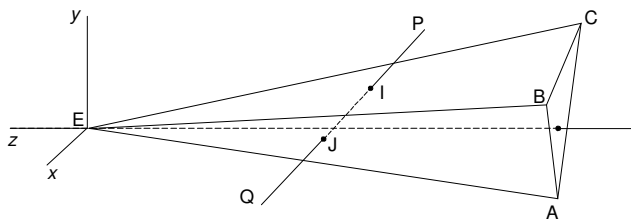
©2006 Wiley & Sons

5



Tests for Visibility (cont'd)

- **Test 3:** Minimax – both P & Q nearer than ABC



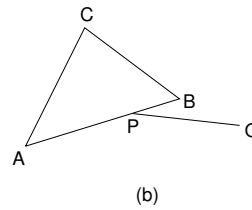
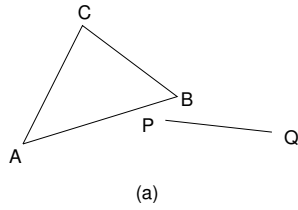
©2006 Wiley & Sons

6



Tests for Visibility (cont'd)

- **Test 4:** PQ on a different side of an edge from the 3rd vertex



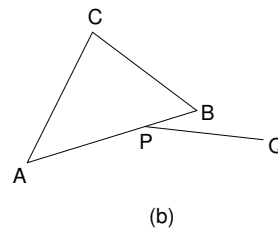
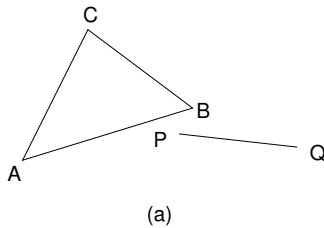
©2006 Wiley & Sons

7



Tests for Visibility (cont'd)

- **Test 4:** PQ on a different side of an edge from the 3rd vertex



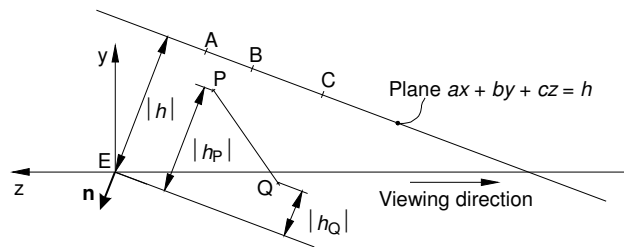
©2006 Wiley & Sons

8



Tests for Visibility (cont'd)

- **Test 6:** PQ on a different side of an edge from the 3rd vertex



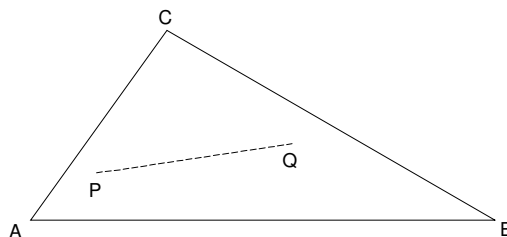
©2006 Wiley & Sons

9



Tests for Visibility (cont'd)

- **Test 7:** PQ on a different side of an edge from the 3rd vertex



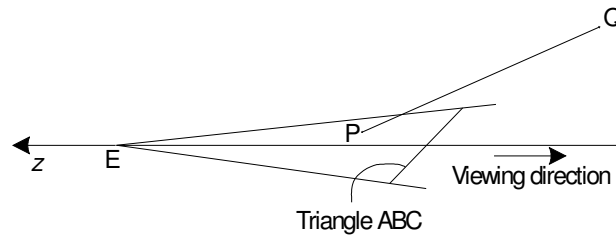
©2006 Wiley & Sons

10



Tests for Visibility (cont'd)

- **Test 8:** PQ on a different side of an edge from the 3rd vertex



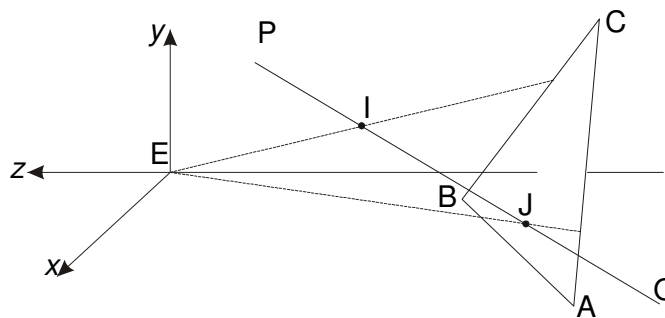
©2006 Wiley & Sons

11



Tests for Visibility (cont'd)

- **Test 9:** PQ on a different side of an edge from the 3rd vertex



©2006 Wiley & Sons

12



Input Format of 3D Objects

- Two parts:
 - A vertex per line: $\text{vertex \# } x_w y_w z_w$
(i.e. vertex number followed by vertex's world coordinates)
 - After "Faces:", a polygon (face) per line represented by a sequence of vertex numbers and terminated by a '.' (ccw when viewed from outside). The 2nd vertex must be a convex vertex.

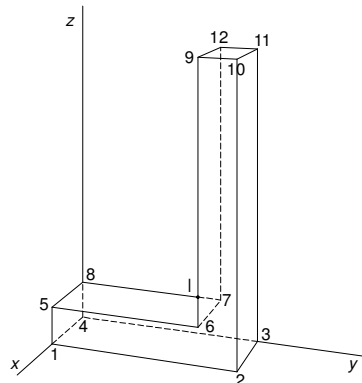
©2006 Wiley & Sons

13



Input Format of 3D Objects (cont'd)

- 1 20 0 0
- 2 20 50 0
- 3 0 50 0
- 4 0 0 0
- 5 20 0 10
-
- Faces:**
- 1 2 10 9 6 5.
- 3 4 8 7 12 11.
- 2 3 11 10.
- 7 6 9 12.
- 4 1 5 8.
- 9 10 11 12.
- 5 6 7 8.
- 1 4 3 2.



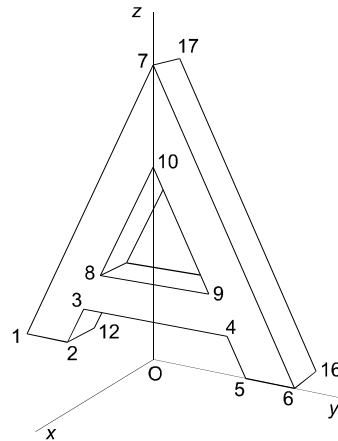
©2006 Wiley & Sons

14



Holes and Invisible Line Segments

- Top and bottom faces of letter 'A' are not proper polygons since they have a hole



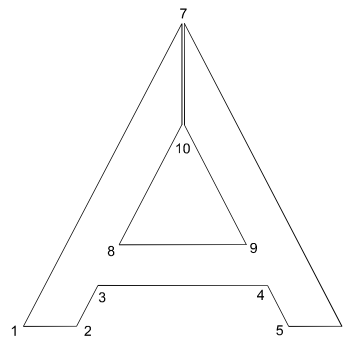
©2006 Wiley & Sons

15



Holes and Invisible Line Segments

- After creating a gap, we have a proper polygon:
1 2 3 4 5 6 7 10 9 8 10 7
- To avoid drawing the line (7 10), we rewrite the above as:
1 2 3 4 5 6 7 -10 9 8 10 -7
- Rule: a line ended with a negative vertex is not drawn (only used in "Faces:" part).



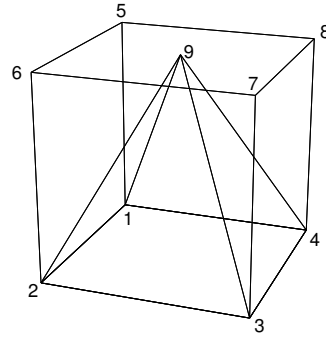
©2006 Wiley & Sons

16



Individual Faces and Lines

- We sometimes want to treat some lines specially, not to obscure lines behind, e.g. line(6 7).
- We simply put these lines in “Faces:” part.



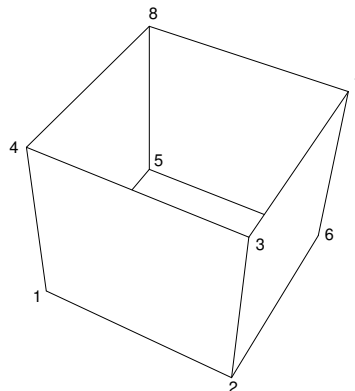
©2006 Wiley & Sons

17



Individual Faces and Lines

- To make individual faces visible from both sides, we specify them twice (ccw and cw) in “Faces:” part.



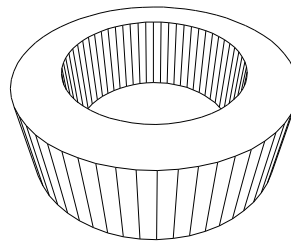
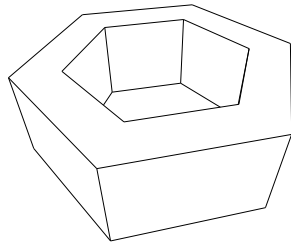
©2006 Wiley & Sons

18



Auto Generation of Input Files

- To automatically generate a cylinder as follows:



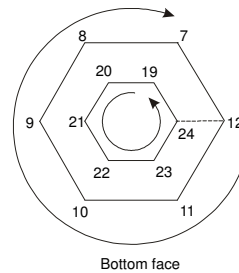
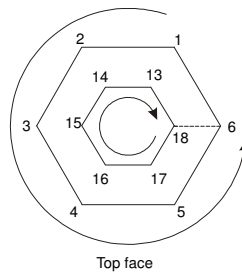
©2006 Wiley & Sons

19



Auto Generation of Input Files

- Outer radius: R
- Inner radius: r
- Number of sides: n



- $n = 6$
- Top face: $z=1$
- Bottom face: $z=0$

©2006 Wiley & Sons

20



Auto Generation of Input Files

- It becomes a circle when n is large enough.
- It becomes solid when $r = 0$.
- In “Faces:” part:
 - 1 2 3 4 5 6 -18 17 16 15 14 13 18 -6 (top face)
 - 12 11 10 9 8 7 -19 20 21 22 23 24 19 -7 (bottom face)
- Top face can be generalized to:
 - 12 ... $n-3n$ $3n-1$ $3n-2$... $2n+1$ $3n-n$
- Bottom face can be generalized to:
 - $2n$ $2n-1$... $n+1$ $-(3n+1)$ $3n+2$ $3n+3$... $4n$ $3n+1$ $-(n+1)$

©2006 Wiley & Sons

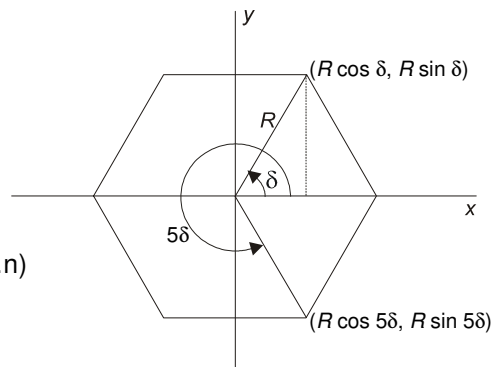
21



Auto Generation of Input Files

- To calculate world coordinates of vertices

$$\delta = \frac{2\pi}{n}$$



- So, outer circle ($i=1\dots n$)
 - $x_i = R \cos(i\delta)$
 - $y_i = R \sin(i\delta)$
- Same for inner circle

©2006 Wiley & Sons

22



HP-GL Output Format

- Hewlett-Packard Graphics Language (HP-GL) for output representation
 - *IN*: Initialize
 - *SP*: Set pen
 - *PU*: Pen up
 - *PD*: Pen down
 - *PA*: Plot absolute

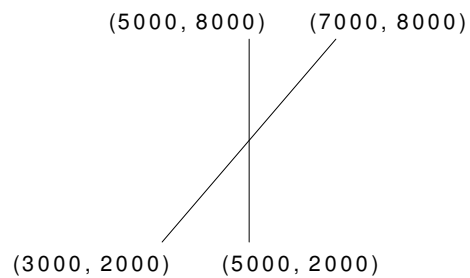
©2006 Wiley & Sons

23



HP-GL Output Format (cont'd)

- Letter 'X':
 - **IN; SP1;**
 - **PU; PA5000, 2000; PD; PA5000, 8000;**
 - **PU; PA3000, 2000; PD; PA7000, 8000;**



©2006 Wiley & Sons

24



Chapter 7 Hidden-Face Elimination

- Back-Face Culling
- Coloring Individual Faces
- Painter's Algorithm
- Z-Buffer Algorithm

©2006 Wiley & Sons

25



Back-Face Culling

- Given a viewpoint invisible faces (back faces) should not be drawn
- *Backface culling* algorithm:
Draw cube:
 - Find center of world coordinate system
 - $d = \rho * \text{ImageSize}/\text{ObjectSize}$
 - Viewing and perspective transformations
 - For each vertex of 6 faces:
 - Find screen coordinates of the vertex and store it
 - Set a different color
 - Use area2 to check if a face is visible, if so, fill it
- This algorithm does not work for removing partially visible faces.

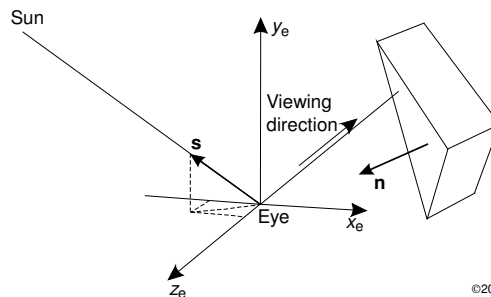
©2006 Wiley & Sons

26



Coloring Individual Faces

- To obtain a color code for a face:
 - Predetermine direction of Sun, then its inner product with $\mathbf{n} = (a, b, c)$ can determine what color to use for face $ax + by + cz = h$



©2006 Wiley & Sons

27



Coloring Individual Faces (cont'd)

Assume a range of inner products

$$\text{inprodRange} = \text{inprodMax} - \text{inprodMin}$$

- `int colorCode(double a, double b, double c)`
- `{ double inprod = a * sunX + b * sunY + c * sunZ;`
- `return (int)Math.round(((inprod - inprodMin)/inprodRange)`
- `* 255);`
- `}`

So we can get color code by calling

```
int cCode = obj.colorCode(a, b, c);  
g.setColor(new color(cCode, cCode, 0)); // shades of yellow
```

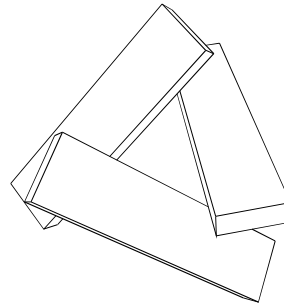
©2006 Wiley & Sons

28



Painter's Algorithm

- Main idea:
 - Display polygons in the order of their distances toward viewpoint
- Features:
 - Fast
 - Working in most cases
 - Not working in special cases



©2006 Wiley & Sons

29



Painter's Algorithm (cont'd)

Steps:

- Compute eye and screen coordinates for whole 3D object (*Obj3D.eyeAndScreen*)
- Computer a , b , c , and h of $ax + by + cz = h$ for every polygon face (*Obj3D.planeCoeff*)
- Triangulate every polygon face (*Polygon3D.triangulate*)
- Decide color of each triangle (*colorCode*)
- Sort all triangles according to their Ze coordinates
- Display all triangles in their predetermined colors

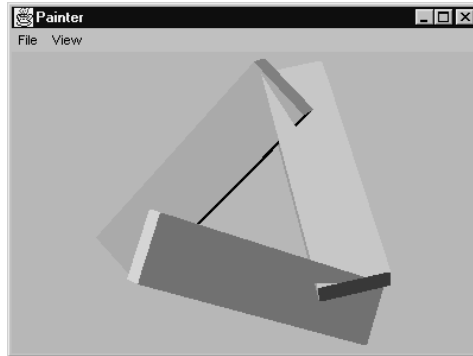
©2006 Wiley & Sons

30



Painter's Algorithm (cont'd)

- Problem: fails to order polygons in certain cases



©2006 Wiley & Sons

31



Z-Buffer Algorithm

- Z-coordinates denote distance from viewpoint
- Z-buffer as a large 2D array of canvas size, storing z-coordinates
- Two buffers used:
 - Frame buffer storing color values, initialized as background color
 - Z-buffer storing z value of each pixel, initialized to 0

©2006 Wiley & Sons

32



Z-Buffer Algorithm (cont'd)

- `int pz; // face's z at p(x, y)`
- `for (y=0; y<ymax; y++)`
- `for (x=0; x<xmax; x++) {`
- `putPixel(x, y, backgroundColor);`
- `zbuff(x, y) = 0; }`
- `for each polygon face`
- `for each pixel in polygon's projection {`
- `pz = polygon's z-value at (x, y);`
- `if (pz >= zbuff(x, y)) { // new point nearer`
- `putPixel(x, y, polygon's color at (x, y));`
- `zbuff(x, y) = pz;`
- `}`
- `}`
- `}`

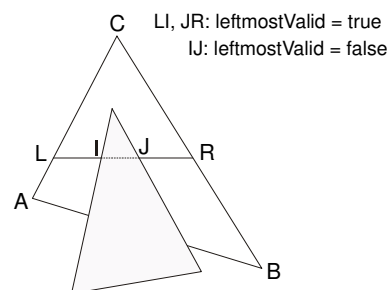
©2006 Wiley & Sons

33



Optimized Z-Buffer Algorithm

- For each scan line
 - `xL, xJ, xK ← 1030`
 - `xL1, xJ1, xK1 ← 10-30`
 - Work on BC, CA, and AB
- Get intersection points L and R
 - e.g. for each scan line between C and B
 - Find intersection R'(xR', y)
 - `xL, xL1, xR'`
 - `xL ← min(xL, xJ, xK)`
 - `xR ← max(xL1, xJ1, xK1)`
- Draw line (L, R)



©2006 Wiley & Sons

34



Optimized Z-Buffer Algorithm

```

boolean leftmostValid = false;
int xLeftmost = 0;
for (int ix=iXL; ix<=iXR; ix++)
{ if (zi < buff[ix][iy]) // < means nearer
  { if (!leftmostValid)
    { xLeftmost = ix;
      leftmostValid = true;
    }
    buff[ix][iy] = (float)zi;
  }
  else
  if (leftmostValid)
  { g.drawLine(xLeftmost, iy, ix-1, iy);
    leftmostValid = false;
  }
  zi += dzdx;
}
if (leftmostValid)
g.drawLine(xLeftmost, iy, iXR, iy);

```

©2006 Wiley & Sons

35



Z-Buffer Algorithm (cont'd)

- To find $z_i (= 1/z)$ for each triangle ABC:
 - Consider plane $ax + by + cz_i = k$, to know how much z_i increases if the point moves 1 pixel up or to the right, then $z_i = (k - ax - by)/c$

and

$$\frac{\partial z_i}{\partial x} = -\frac{a}{c} \quad \frac{\partial z_i}{\partial y} = -\frac{b}{c}$$

- Based on centroid $D(x_D, y_D)$ to compute z_i :

- $x_D = (x_A + x_B + x_C)/3$;
- $y_D = (y_A + y_B + y_C)/3$;
- $z_{Di} = (z_{Ai} + z_{Bi} + z_{Ci})/3$;

$$z_i = z_{Di} + (y - y_D) \frac{\partial z_i}{\partial y} + (x_L - x_D) \frac{\partial z_i}{\partial x}$$

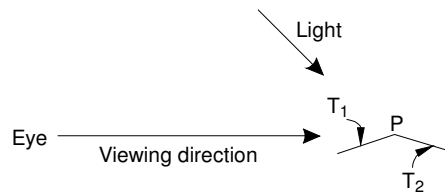
©2006 Wiley & Sons

36



Z-Buffer Algorithm (cont'd)

- Boundary problem:
 - P is on both triangles T_1 and T_2 , it will be colored twice as both T_1 's and T_2 's colors.



- To make P counted once for T_1 , use a factor of 1.01 in z_i :
$$z_i = 1.01 * z_{Di} + (y - y_D) * dzdy + (x_L - x_D) * dzdx$$

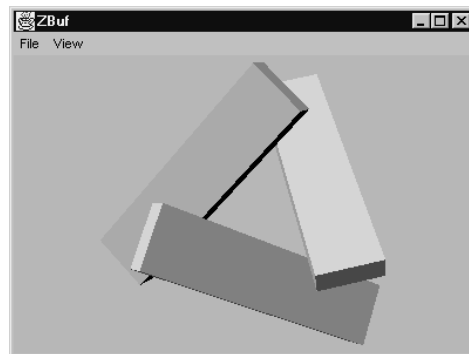
©2006 Wiley & Sons

37



Z-Buffer Algorithm (cont'd)

- Problem with Painters algorithm no longer exist



©2006 Wiley & Sons

38



Chapter 8 Fractals

- Introduction
- Koch Curves
- String Grammars
- Mandelbrot and Julia Sets



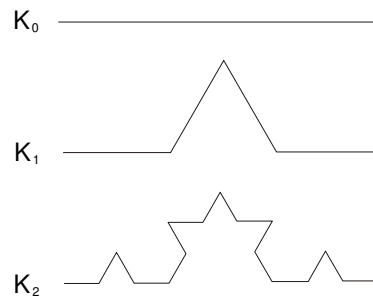
Introduction to Fractals

- Many aspects in nature repeat in patterns similar at different scales
- Self-similar structures can be modeled by *Fractal geometry*
- Fractals are useful in many applications, in science, technology, and computer generated art



Koch Curves

- Begin with a straight line and call it K_0 ;
- Divide each segment of K_n into 3 equal parts; and
- Replace the middle part by two sides of an equilateral triangle of the same length.



©2006 Wiley & Sons

41



Koch Curves (cont'd)

Interesting characteristics:

- Each segment increased in length by a factor of $4/3$ (K_{n+1} is $4/3$ as long as K_n , and K_i has the total length of $(4/3)^i$).
- When n is getting large, the curve still appears to have the same shape and roughness.
- When n becomes infinite, the curve has an infinite length, while occupying a finite region in the plane.

©2006 Wiley & Sons

42



Koch Curves (cont'd)

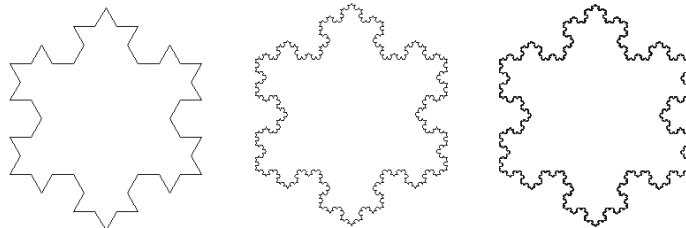
- Pseudo-code:
 - If $(n == 0)$ Draw a straight line;
 - Else
 - }
 - { Draw K_{n-1} ;
 - Turn left by 60° ;
 - Draw K_{n-1} ;
 - Turn right by 120° ;
 - Draw K_{n-1} ;
 - Turn left by 60° ;
 - Draw K_{n-1} ;
 - }

©2006 Wiley & Sons

43



Koch Snowflakes



©2006 Wiley & Sons

44



Turtle Graphics

- Originated in Logo programming language, *turtle graphics* is a means of computer drawing using the concept of a turtle crawling over the drawing space with a pen attached to its underside.
- The drawing is always relative to the current position and direction of the turtle.

©2006 Wiley & Sons

45



String Grammars

- Specification of a common pattern is called a *grammar*
- A string of characters defining a common pattern instructs the turtle to draw the pattern
- Most common character commands:
 - F - move forward distance D while drawing in current direction.
 - + - turn right through angle α .
 - - - turn left through angle α .

©2006 Wiley & Sons

46



String Production Rule

- For Koch curves, $F \rightarrow F-F++F-F$
- A string of characters defining a common pattern instructs the turtle to draw the pattern
- Most common character commands:
 - F - move forward distance D while drawing in current direction.
 - + - turn right through angle α .
 - - - turn left through angle α .

©2006 Wiley & Sons

47



Grammar Template

- (axiom, F-string, f-string, X-string, Y-string, angle)
- *axiom* from which turtle starts;
- *F-string* produces strings from F (turtle moves forward while drawing);
- *f-string* produces strings from f (turtle moves forward without drawing);
- *X-string* produces strings from X (turtle ignores);
- *Y-string* produces strings from Y (turtle ignores);
- *angle* at which turtle should turn.

©2006 Wiley & Sons

48



Grammar Template

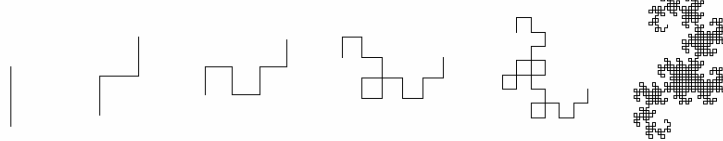
- Example curves:
 - Dragon curve:
($X, F, nil, X+YF+, -FX-Y, 90$)
 - Hilbert curve:
($X, F, nil, -YF+XFX+FY-, +XF-YFY-FX+, 90$)
 - Sierpinski arrowhead:
($YF, F, nil, YF+XF+Y, XF-YF-X, 60$)

©2006 Wiley & Sons

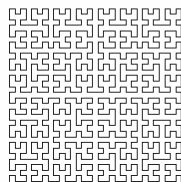
49



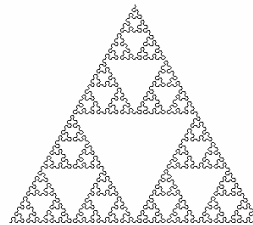
Example Curves



Dragon curves: 1st, 2nd, 3rd, 4th, 5th and 11th generations



Hilbert curve (5th generation)



Sierpinski arrowhead (7th generation)

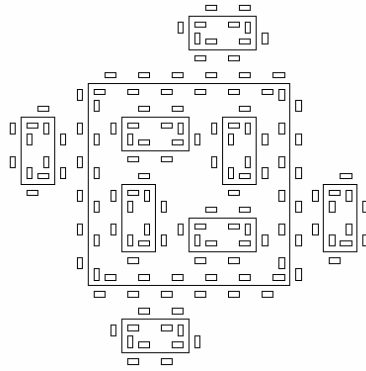
©2006 Wiley & Sons

50



Islands

- $(F+F+F+F, F+f-FF+F+FF+Ff+FF-f+FF-F-FF-Ff-FFF, fffff, nil, nil, 90)$



©2006 Wiley & Sons

51



Branching

- turtle's *state* - current position + direction
- two more character commands:
 - [- *store current state of turtle*
 -] - *restore turtle's previously stored state*

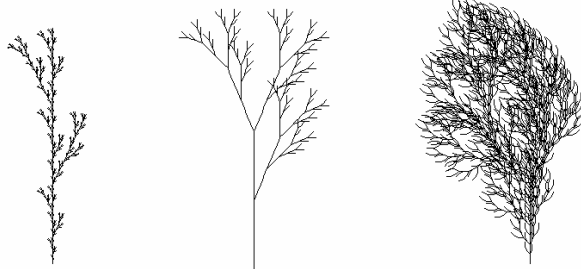
©2006 Wiley & Sons

52



Branching: Fractal Trees

- Left: $(F, F[+F]F[-F]F, nil, nil, nil, 25.7)$
- Middle: $(X, FF, nil, F[+X]F[-X]+X, nil, 20.0)$
- Right: $(F, FF-[-F+F+F]+[+F-F-F], nil, nil, nil, 22.5)$



©2006 Wiley & Sons

53



Mandelbrot and Julia Sets

- Investigates: what happens when one iterates a function endlessly
- Mandelbrot set uses a simple function
$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c \text{ (} c \text{ is a complex number)}$$
- A sequence of values, called *orbit*:
 - $z_0 = 0$
 - $z_1 = z_0^2 + c = c$
 - $z_2 = z_1^2 + c = c^2 + c$
 -

©2006 Wiley & Sons

54



Mandelbrot Set

- If orbit stays within a distance of 2 from the origin forever, c is said to be in Mandelbrot set
- If orbit diverges from the origin, c is not in the set.

©2006 Wiley & Sons

55



Draw Mandelbrot Images

- Complex numbers are 2D!
- $z = x + yi$ (x – real part, y – imaginary part) is displayed at position (x, y)

- Distance:

$$|z| = \sqrt{x^2 + y^2}$$

- We perform the test

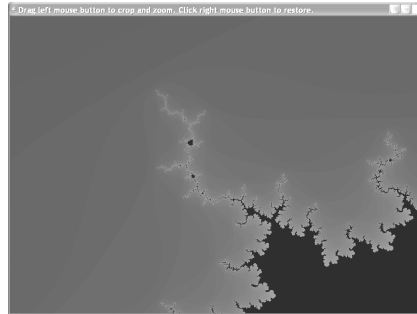
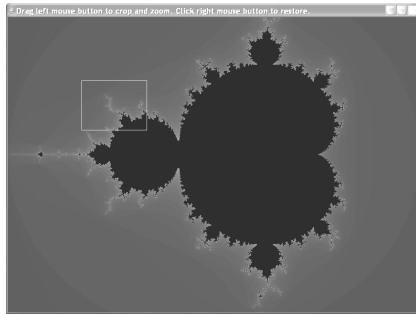
$$|z|^2 > 4$$

to avoid computing square root

©2006 Wiley & Sons

56

Mandelbrot Sets



©2006 Wiley & Sons

57

Julia Set

- Associated with every point in complex plane is a set similar to Mandelbrot set, called a *Julia set*
- Use the same iteration $z_{n+1} = z_n^2 + c$
- Mandelbrot set can be used to select c for Julia set, it forms an index into Julia set

©2006 Wiley & Sons

58

Julia Sets (cont'd)

- We choose points near boundaries of Mandelbrot set as starting values z_0
- E.g. $c = -0.76 + 0.084i$, the point near top of the circle on the left of Mandelbrot set



©2006 Wiley & Sons

59