Computer Graphics (CS 543)
Lecture 3a: Fractals

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
What are Fractals?

- Mathematical expressions to generate pretty pictures
- Evaluate math functions to create drawings
  - approach infinity -> converge to image
- Utilizes recursion on computers
- Popularized by Benoit Mandelbrot (Yale university)
- Dimensional:
  - Line is 1-dimensional
  - Plane is 2-dimensional
Fractals: Self-similarity

- Fractals are defined in terms of self-similarity
- See similar sub-images within image as we zoom in
- Example: surface roughness or profile same as we zoom in
Applications of Fractals

- Grass
- Coastline
- Fire
- Clouds

Other applications:
- Mountains
- Branches of a tree
- Surface of a sponge
- Cracks in the pavement
- Designing antennae (www.fractenna.com)
Example: Mandelbrot Set
Example: Fractal Terrain

Courtesy: Mountain 3D
Fractal Terrain software
Application: Fractal Art
Recall: Sierpinski Gasket Program

- Popular fractal
Koch Curves

- Discovered in 1904 by Helge von Koch
- Start with straight line of length 1
- Recursively:
  - Divide line into 3 equal parts
  - Replace middle section with triangular bump, sides of length $1/3$
  - New length $= 4/3$
Koch Snowflakes

Can form Koch snowflake by joining three Koch curves
Koch Snowflakes

Pseudocode, to draw $K_n$:

If (n equals 0) draw straight line
Else{
  Draw $K_{n-1}$
  Turn left 60°
  Draw $K_{n-1}$
  Turn right 120°
  Draw $K_{n-1}$
  Turn left 60°
  Draw $K_{n-1}$
}
L-Systems: Lindenmayer Systems

- Express complex curves as simple set of **string-production** rules
- Example rules:
  - ‘F’: go forward a distance 1 in current direction
  - ‘+’: turn right through angle $A$ degrees
  - ‘-’: turn left through angle $A$ degrees
- Using these rules, can express Koch curve as: “F-F++F-F”
- Angle $A = 60$ degrees
L-Systems: Koch Curves

- Rule for Koch curves is $F \rightarrow F-F++F-F$
- Means each iteration replaces every ‘F’ occurrence with “F-F++F-F”
- So, if initial string (called the **atom**) is ‘F’, then
  - $S_1 = “F-F++F-F”$
  - $S_2 = “F-F++F-F- F-F++F++ F-F++F-F- F-F++F-F”$
  - $S_3 = .....$
- Gets very large quickly
Hilbert Curve

- Discovered by German Scientist, David Hilbert in late 1900s
- Space filling curve
- Drawn by connecting centers of 4 sub-squares, make up larger square.
- **Iteration 0:** 3 segments connect 4 centers in upside-down U
Hilbert Curve: Iteration 1

- Each of 4 squares divided into 4 more squares
- U shape shrunk to half its original size, copied into 4 sectors
- In top left, simply copied, top right: it's flipped vertically
- In the bottom left, rotated 90 degrees clockwise,
- Bottom right, rotated 90 degrees counter-clockwise.
- 4 pieces connected with 3 segments, each of which is same size as the shrunken pieces of the U shape (in red)
Hilbert Curve: Iteration 2

- Each of the 16 squares from iteration 1 divided into 4 squares
- Shape from iteration 1 shrunk and copied.
- 3 connecting segments (shown in red) are added to complete the curve.
- Implementation? Recursion is your friend!!
Gingerbread Man

- Each new point $q$ is formed from previous point $p$ using the equation

$$
q.x = M(1 + 2L) - p.y + |p.x - LM|;
q.y = p.x.
$$

- For 640 x 480 display area, use constants $M = 40$  $L = 3$

- A good starting point $p$ is (115, 121)
The Fern

Start at initial point (0,0). Draw dot at (0,0).

Use either f1, f2, f3 or f4 with probabilities .01, .07, .07, .85 to generate next point.

The Fern

Each new point \((\text{new.x}, \text{new.y})\) is formed from the prior point \((\text{old.x}, \text{old.y})\) using the rule:

\[
\text{new.x} := a[\text{index}] * \text{old.x} + c[\text{index}] * \text{old.y} + tx[\text{index}];
\]

\[
\text{new.y} := b[\text{index}] * \text{old.x} + d[\text{index}] * \text{old.y} + ty[\text{index}];
\]

\(a[1] := 0.0; \) \(b[1] := 0.0; \) \(c[1] := 0.0; \) \(d[1] := 0.16; \)

\(tx[1] := 0.0; \) \(ty[1] := 0.0; \) \(\text{i.e \ values \ for \ function \ f1}\)

\(a[2] := 0.2; \) \(b[2] := 0.23; \) \(c[2] := -0.26; \) \(d[2] := 0.22; \)

\(tx[2] := 0.0; \) \(ty[2] := 1.6; \) \(\text{values \ for \ function \ f2}\)

\(a[3] := -0.15; \) \(b[3] := 0.26; \) \(c[3] := 0.28; \) \(d[3] := 0.24; \)

\(tx[3] := 0.0; \) \(ty[3] := 0.44; \) \(\text{values \ for \ function \ f3}\)

\(a[4] := 0.85; \) \(b[4] := -0.04; \) \(c[4] := 0.04; \) \(d[4] := 0.85; \)

\(tx[4] := 0.0; \) \(ty[4] := 1.6; \) \(\text{values \ for \ function \ f4}\)
Mandelbrot Set

- Based on iteration theory
- Function of interest:
  \[ f(z) = (s)^2 + c \]
- Sequence of values (or orbit):
  \[ d_1 = (s)^2 + c \]
  \[ d_2 = (((s)^2 + c)^2 + c)^2 + c \]
  \[ d_3 = (((((s)^2 + c)^2 + c)^2 + c)^2 + c)^2 + c \]
  \[ d_4 = ((((((s)^2 + c)^2 + c)^2 + c)^2 + c)^2 + c)^2 + c \]
Mandelbrot Set

- Orbit depends on $s$ and $c$
- Basic question,: 
  - For given $s$ and $c$,
    - does function stay finite? (within Mandelbrot set)
    - explode to infinity? (outside Mandelbrot set)
- Definition: if $|d| < 1$, orbit is finite else infinite
- Examples orbits:
  - $s = 0, c = -1$, orbit = 0,-1,0,-1,0,-1,0,-1,..... finite
  - $s = 0, c = 1$, orbit = 0,1,2,5,26,677...... explodes
Mandelbrot Set

- Mandelbrot set: use complex numbers for $c$ and $s$
- Always set $s = 0$
- Choose $c$ as a complex number
- For example:
  - $s = 0, c = 0.2 + 0.5i$
- Hence, orbit:
  - $0, c, c^2+ c, (c^2+ c)^2 + c, \ldots$...
- Definition: Mandelbrot set includes all finite orbit $c$
Mandelbrot Set

- Some complex number math:

\[ i^2 = -1 \]

- Example:

\[ 2i^3 = -6 \]

- Modulus of a complex number, \( z = ai + b \):

\[ |z| = \sqrt{a^2 + b^2} \]

- Squaring a complex number:

\[ (x + yi)^2 = (x^2 - y^2) + (2xy)i \]
Mandelbrot Set

Examples: Calculate first 3 terms

- with $s=2$, $c=-1$, terms are
  
  \[2^2 - 1 = 3\]
  \[3^2 - 1 = 8\]
  \[8^2 - 1 = 63\]

- with $s = 0$, $c = -2+i$
  
  \[(x + yi)^2 = (x^2 - y^2) + (2xy)i\]
  
  \[0 + (-2 + i) = -2 + i\]
  \[(-2 + i)^2 + (-2 + i) = 1 - 3i\]
  \[(1 - 3i)^2 + (-2 + i) = -10 - 5i\]
Mandelbrot Set

- **Fixed points**: Some complex numbers converge to certain values after $x$ iterations.

- **Example**:
  - $s = 0$, $c = -0.2 + 0.5i$ converges to $-0.249227 + 0.333677i$ after 80 iterations
  - **Experiment**: square $-0.249227 + 0.333677i$ and add $-0.2 + 0.5i$
  - Mandelbrot set depends on the fact the convergence of certain complex numbers
Mandelbrot Set Routine

- Math theory says calculate terms to infinity
- On computer, cannot iterate forever: our program will hang!
- Instead iterate 100 times
- **Math theorem:**
  - if no term has exceeded 2 after 100 iterations, never will!
- Routine returns:
  - 100, if modulus doesn’t exceed 2 after 100 iterations
  - Number of times iterated before modulus exceeds 2, or

\[
\begin{align*}
\text{Mandelbrot function} & \quad \text{Number} < 100 \\
& \quad (\text{first term} > 2) \\
& \quad \text{Number} = 100 \quad (\text{did not explode})
\end{align*}
\]
Mandelbrot dwell( ) function

\[(x + yi)^2 = (x^2 - y^2) + (2xy)i\]
\[(x + yi)^2 + (c_x + c_yi) = [(x^2 - y^2) + c_x] + (2xy + c_y)i\]

```c
int dwell(double cx, double cy)
{
    // return true dwell or Num, whichever is smaller
    #define Num 100 // increase this for better pics

    double tmp, dx = cx, dy = cy, fsq = cx*cx + cy*cy;
    for(int count = 0; count <= Num && fsq <= 4; count++)
    {
        tmp = dx;       // save old real part
        dx = dx*dx - dy*dy + cx; // new real part
        dy = 2.0 * tmp * dy + cy; // new imag. Part
        fsq = dx*dx + dy*dy;
    }
    return count; // number of iterations used
}
```
Mandelbrot Set

- Map real part to x-axis
- Map imaginary part to y-axis
- Decide range of complex numbers to investigate. E.g:
  - X in range [-2.25: 0.75], Y in range [-1.5: 1.5]

E.g. -1.5 + i

Range of complex Numbers ( \( c \) )
X in range [-2.25: 0.75], Y in range [-1.5: 1.5]

Call ortho2D to set range of values to explore
Mandelbrot Set

- Set world window (ortho2D) (range of complex numbers to investigate)
  - $X$ in range $[-2.25: 0.75]$, $Y$ in range $[-1.5: 1.5]$
- Set viewport (glviewport). E.g:
  - Viewport = $[V.L, V.R, W, H] = [60, 80, 380, 240]$
So, for each pixel:
- For each point \((c)\) in world window call your \(dwell()\) function
- Assign color <Red,Green,Blue> based on \(dwell()\) return value

Choice of color determines how pretty

Color assignment:
- Basic: In set (i.e. \(dwell() = 100\)), color = black, else color = white
- Discrete: Ranges of return values map to same color
  - E.g 0 – 20 iterations = color 1
  - 20 – 40 iterations = color 2, etc.
- Continuous: Use a function

\[ s, c \quad \begin{cases} \text{Mandelbrot function} \\ \text{Number} < 100 \quad (\text{first term} > 2) \\ \text{Number} = 100 \quad (\text{did not explode}) \end{cases} \]
Free Fractal Generating Software

- Fractint
- FracZoom
- 3DFrac
References