18.9 Fractals

A **fractal** is a geometric figure that can be generated from a pattern repeated recursively (Fig. 18.12). The figure is modified by recursively applying the pattern to each segment of the original figure. Although such figures had been studied before the 20th century, it was the mathematician Benoit Mandelbrot who in the 1970s introduced the term “fractal,” along with the specifics of how a fractal is created and the practical applications of fractals. Mandelbrot’s fractal geometry provides mathematical models for many complex forms found in nature, such as mountains, clouds and coastlines. Fractals have many uses in mathematics and science. They can be used to better understand systems or patterns that appear in nature (e.g., ecosystems), in the human body (e.g., in the folds of the brain), or in the universe (e.g., galaxy clusters). Not all fractals resemble objects in nature. Drawing fractals has become a popular art form. Fractals have a **self-similar property**—when subdivided into parts, each resembles a reduced-size copy of the whole. Many fractals yield an exact copy of the original when a portion of the fractal is magnified—such a fractal is said to be **strictly self-similar**.

18.9.1 Koch Curve Fractal

As an example, let’s look at the strictly self-similar **Koch Curve** fractal (Fig. 18.12). It’s formed by removing the middle third of each line in the drawing and replacing it with two lines that form a point, such that if the middle third of the original line remained, an equilateral triangle would be formed. Formulas for creating fractals often involve removing all or part of the previous fractal image. This pattern has already been determined for this fractal—we focus here on how to use those formulas in a recursive solution.

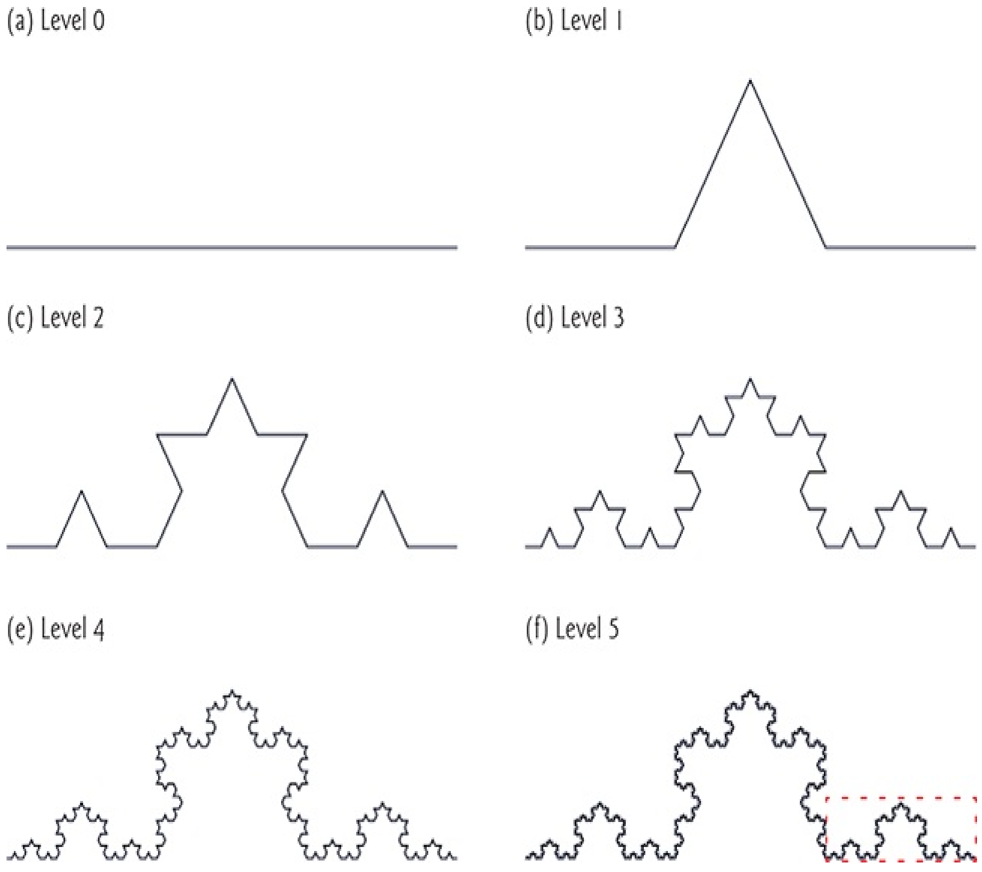


Fig. 18.12

Koch Curve fractal.

We start with a straight line (Fig. 18.12(a)) and apply the pattern, creating a triangle from the middle third (Fig. 18.12(b)). We then apply the pattern again to each straight line, resulting in Fig. 18.12(c). Each time the pattern is applied, we say that the fractal is at a new **level**, or **depth** (sometimes the term **order** is also used). Fractals can be displayed at many levels—for example, a fractal at level 3 has had three iterations of the pattern applied (Fig. 18.12(d)). After only a few iterations, this fractal begins to look like a portion of a snowflake (Fig. 18.12(e and f)). Since this is a strictly self-similar fractal, each portion of it contains an exact copy of the fractal. In Fig. 18.12(f), for example, we've highlighted a portion of the fractal with a dashed box. If the image in this box were increased in size, it would look exactly like the entire fractal of part (f). Exercise 18.24 asks you to use the drawing techniques you'll learn in this section to implement the Koch Curve.

A similar fractal, the **Koch Snowflake**, is similar to the Koch Curve but begins with a triangle rather than a line. The same pattern is applied to each side of the triangle, resulting in an image that looks like an enclosed snowflake. Exercise 18.25 asks you to research the Koch Snowflake, then create an app that draws it.

18.9.2 (Optional) Case Study: Lo Feather Fractal

We now demonstrate using recursion to draw fractals by writing a program to create a strictly self-similar fractal. We call this the “Lo feather fractal,” named for Sin Han Lo, a Deitel & Associates colleague who created it. The fractal will eventually resemble one-half of a feather (see the outputs in Fig. 18.20). The base case, or fractal level of 0, begins as a line between two points, A and B (Fig. 18.13). To create the next higher level, we find the midpoint (C) of the line. To calculate the location of point C, use the following formula:

```
xC = (xA + xB) / 2;
yC = (yA + yB) / 2;
```

[*Note:* The x and y to the left of each letter refer to the *x*-coordinate and *y*-coordinate of that point, respectively. For example, xA refers to the *x*-coordinate of point A, while yC refers to the *y*-coordinate of point C. In our diagrams we denote the point by its letter, followed by two numbers representing the *x*- and *y*-coordinates.]

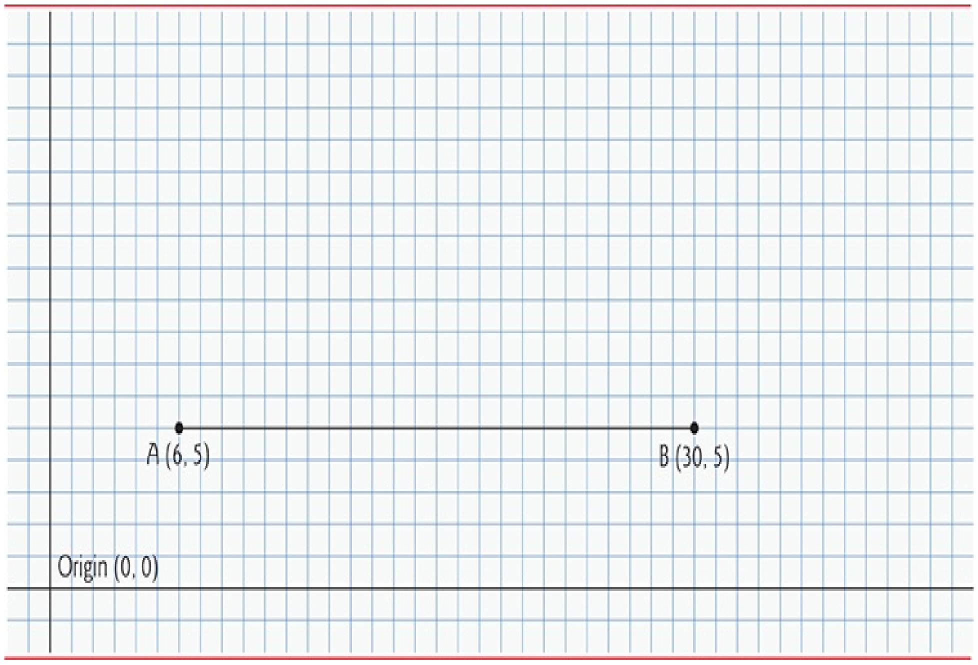


Fig. 18.13

“Lo feather fractal” at level 0.

To create this fractal, we also must find a point D that lies left of segment AC and creates an isosceles right triangle ADC. To calculate point D’s location, use the formulas:

```
xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
```

We now move from level 0 to level 1 as follows: First, add points C and D (as in Fig. 18.14). Then, remove the original line and add segments DA, DC and DB. The remaining lines will curve at an angle, causing our fractal to look like a feather.

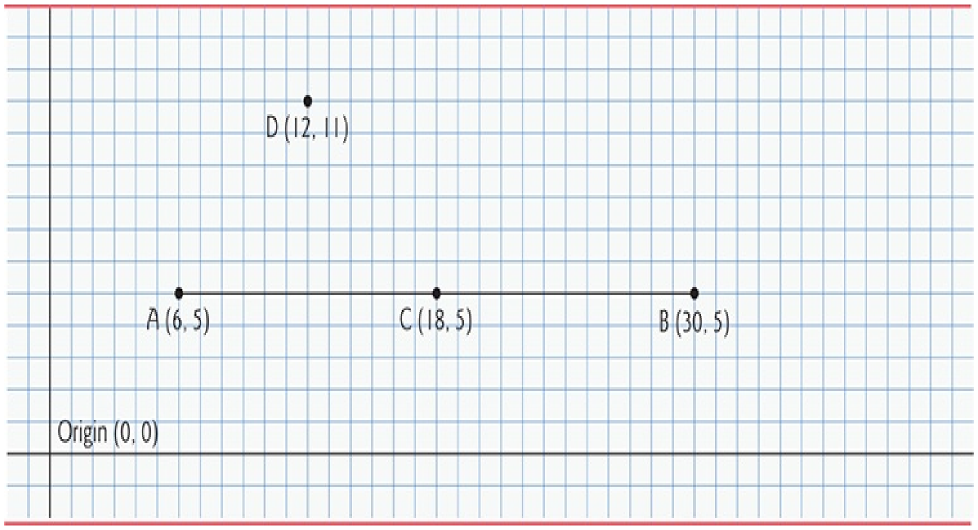


Fig. 18.14

Determining points C and D for level 1 of the “Lo feather fractal.”

For the next level of the fractal, this algorithm is repeated on each of the three lines in level 1. For each line, the formulas above are applied, where the former point `D` is now considered to be point `A`, while the other end of each line is considered to be point `B`. Figure 18.15 contains the line from level 0 (now a dashed line) and the three added lines from level 1. We’ve changed point `D` to be point `A`, and the original points `A`, `C` and `B` to `B1`, `B2` and `B3`, respectively. The preceding formulas have been used to find the new points `C` and `D` on each line. These points are also numbered 1–3 to keep track of which point is associated with each line. The points `C1` and `D1`, for example, represent points `C` and `D` associated with the line formed from points `A` to `B1`.

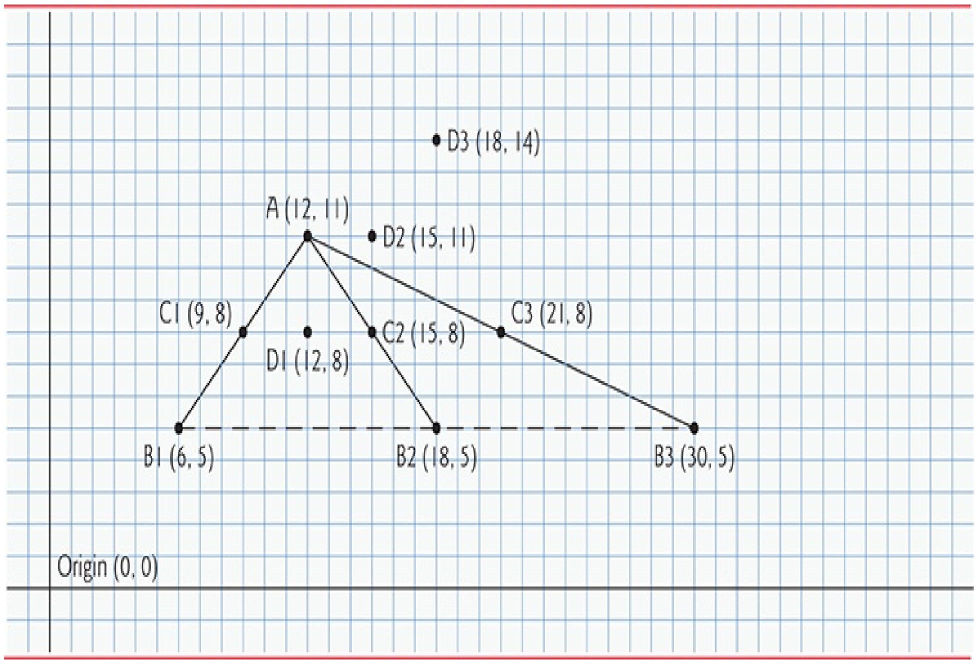


Fig. 18.15

“Lo feather fractal” at level 1, with C and D points determined for level 2. [*Note:* The fractal at level 0 is included as a dashed line as a reminder of where the line was located in relation to the current fractal.]

Description

To achieve level 2, the three lines in Fig. 18.15 are removed and replaced with new lines from the `C` and `D` points just added. Figure 18.16 shows the new lines (the lines from level 2 are shown as dashed lines for your convenience).

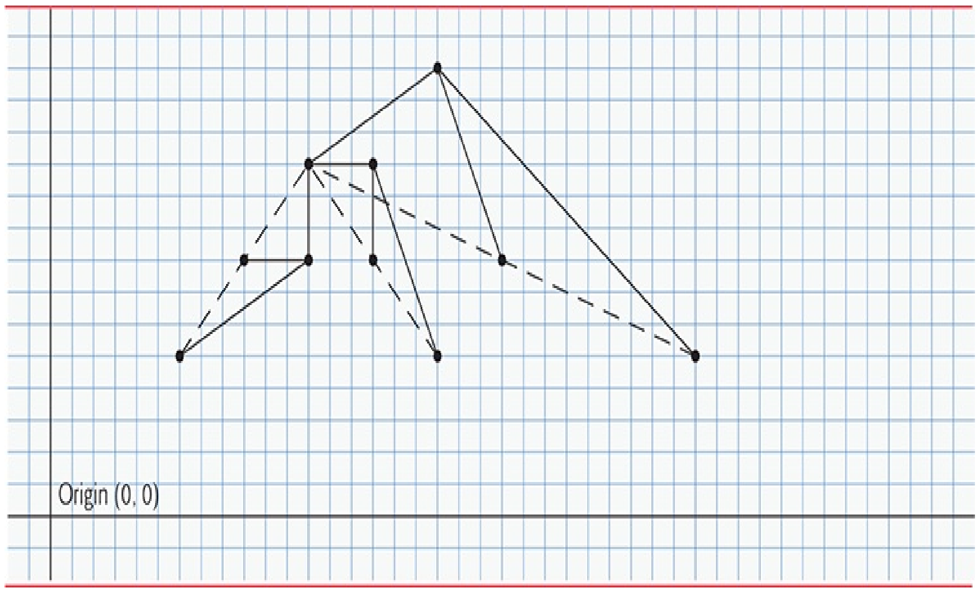


Fig. 18.16

“Lo feather fractal” at level 2, with dashed lines from level 1 provided.

Description

Figure 18.17 shows level 2 without the dashed lines from level 1. Once this process has been repeated several times, the fractal will begin to look like one-half of a feather, as shown in the output of Fig. 18.20.

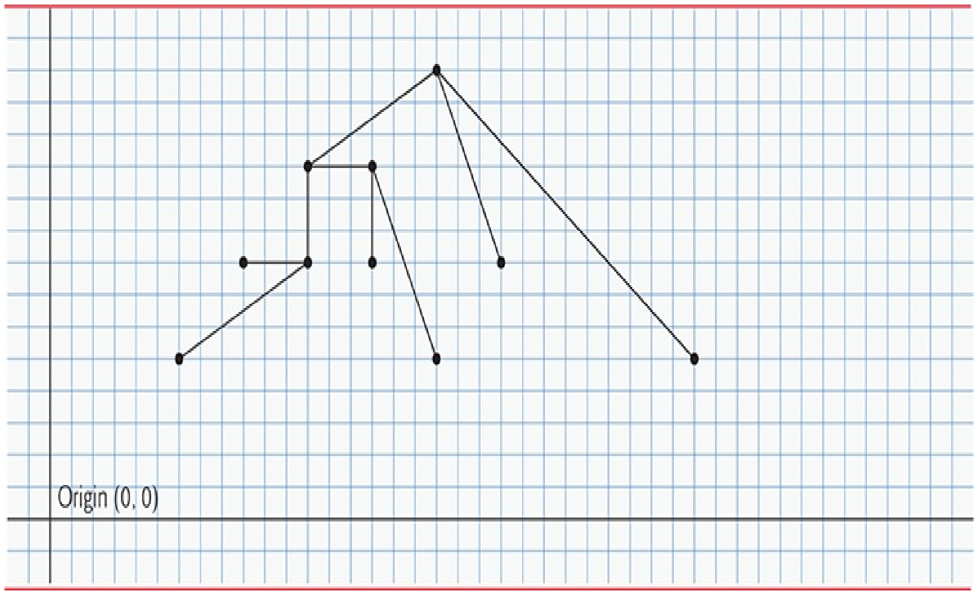


Fig. 18.17

“Lo feather fractal” at level 2.

18.9.3 (Optional) Fractal App GUI

In this section and the next, we build a JavaFX **Fractal** app that displays the Lo Fractal—we do not show the example’s `Application` subclass, because it performs the same tasks we demonstrated in Chapters 12–13 to load and display the app’s FXML.

Figure 18.18 shows the GUI with the **fx:id** values for the controls we access programmatically. Like the **Painter** app in Section 13.3, this app uses a `BorderPane` layout with a with a white background color:

- In the top area, we placed a `ToolBar` (located in the Scene Builder **Library**’s **Containers** section). A `ToolBar` layout arranges its controls horizontally (by default) or vertically. Typically, you place `ToolBar`s at your GUI’s edges, such as in a `BorderPane`’s top, left, bottom or right areas.
- In the center, we placed a 400-by-480 pixel `Canvas` (from the Scene Builder **Library**’s **Miscellaneous** section). `Canvas` is a `Node` subclass in which you can draw graphics using a `GraphicsContext` (both in the package `javafx.scene.canvas`). We show you how to draw a line in a specific color in this example and discuss classes `Canvas` and `GraphicsContext` in detail in Section 22.10.

To size the `BorderPane` layout to its contents, we reset its **Pref Width** and **Pref Height** property values (as you learned in Chapter 13) to `USE_COMPUTED_SIZE`.

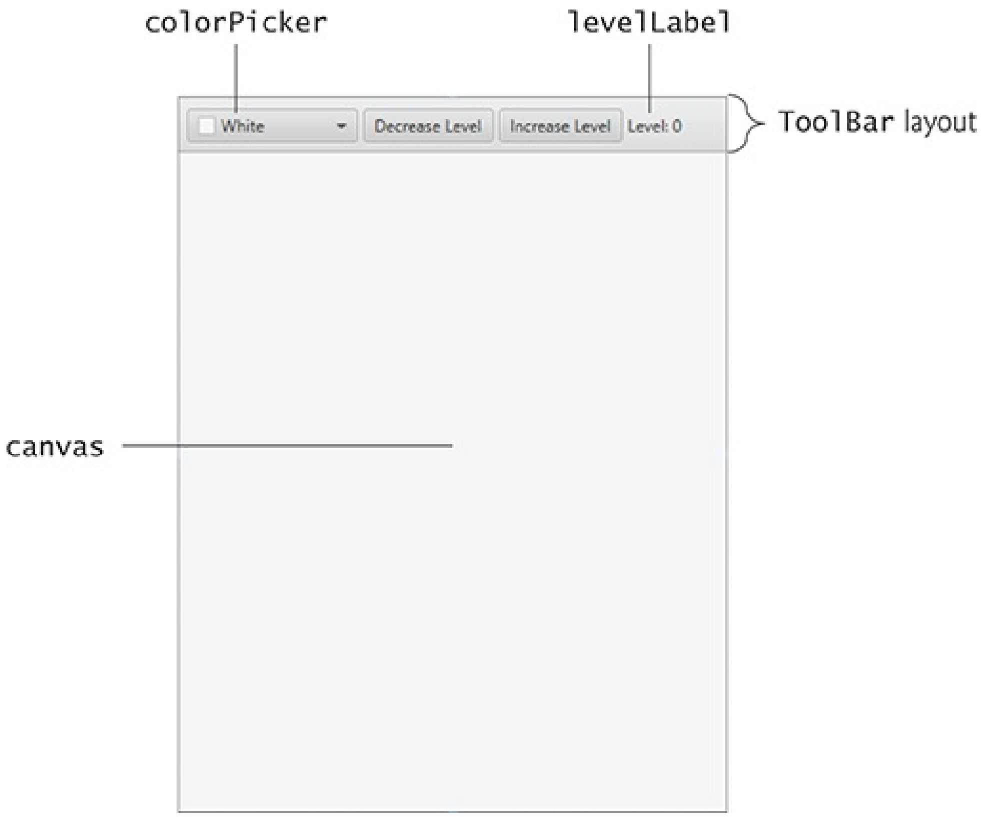


Fig. 18.18

Fractal GUI labeled with **fx:id**s for the programmatically manipulated controls.

Description

`ToolBar` and Its Additional Controls

By default, the `ToolBar` you drag onto your layout has one `Button`. You can drag additional controls onto the `ToolBar` and, if necessary, remove the default `Button`. We added a `ColorPicker` control (with **fx:id** `colorPicker`), another `Button` and a `Label` (with **fx:id** `levelLabel`). For the two `Button`s and the `Label`, we set their text as shown in Fig. 18.18.

`ColorPicker`

A `ColorPicker` provides a predefined color-selecting GUI that, by default, enables the user to choose colors by color swatches (small squares of sample colors). Initially, the selected color is **White**. We'll programmatically set this to **Blue** in the app's controller. Figure 18.19 shows the `ColorPicker`'s default GUI and the **Custom Colors** GUI that appears when you click the `ColorPicker`'s **Custom**

Color… link. The **Custom Colors** GUI enables you to select any custom color.

Event Handlers

When the user selects a color in the default GUI or customizes a color and presses **Save** in the **Custom Colors** GUI, an `ActionEvent` occurs. In Scene Builder, for the `ColorPicker`'s **On Action** event handler, we specified `colorSelected`. In addition, for the **Decrease Level** and **Increase Level** `Button`'s **On Action** events handlers, we specified `decreaseLevelButtonPressed` and `increaseLevelButtonPressed`, respectively.

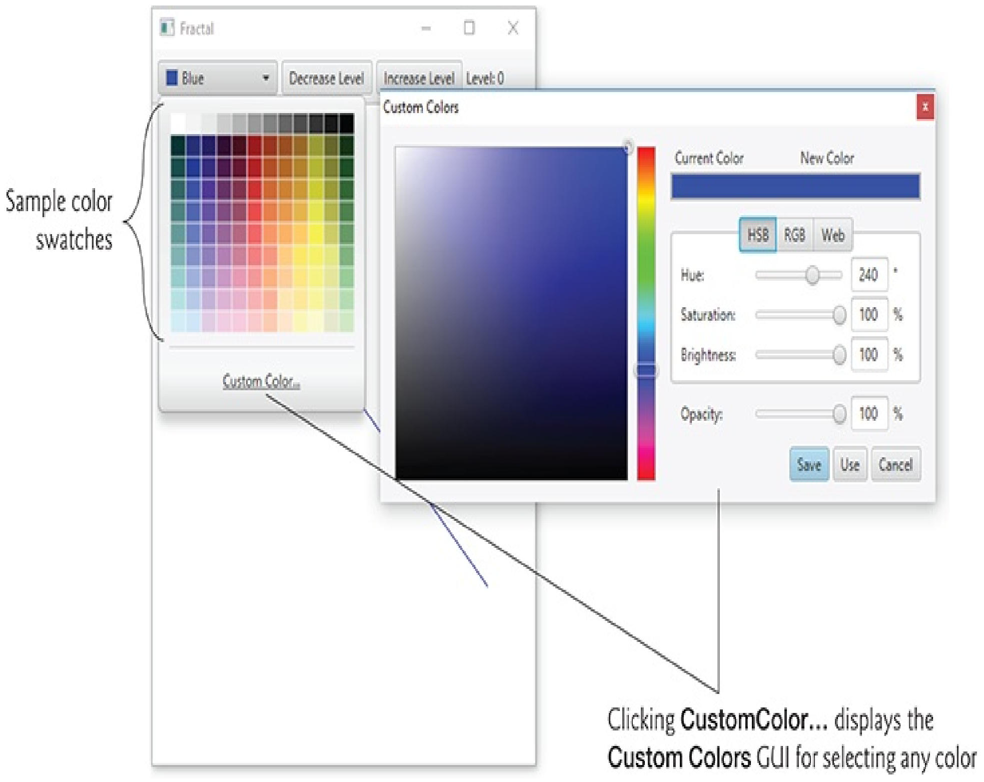


Fig. 18.19

`ColorPicker`’s predefined GUI.

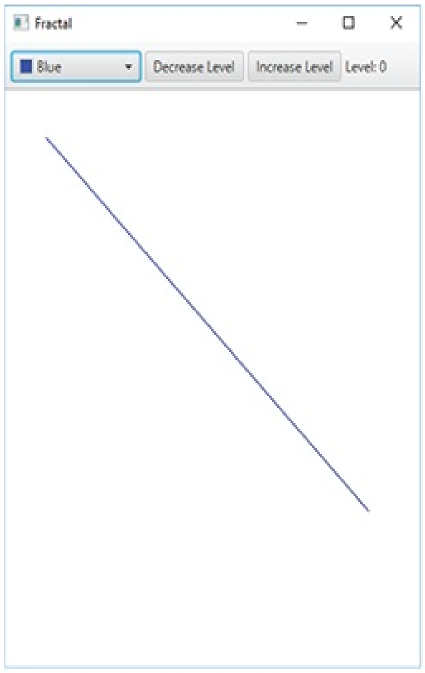
Description

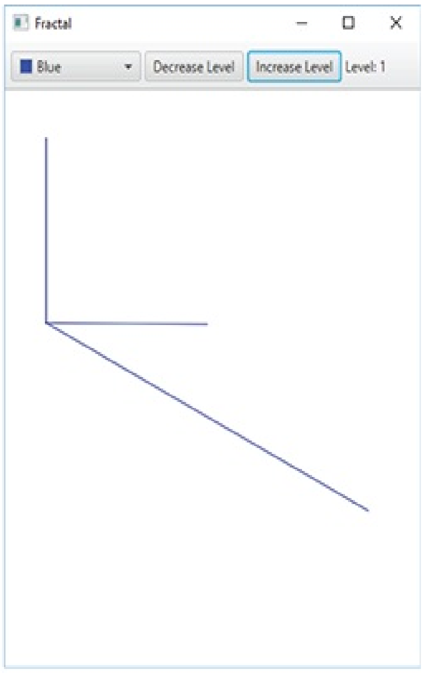
18.9.4 (Optional) `FractalController` Class

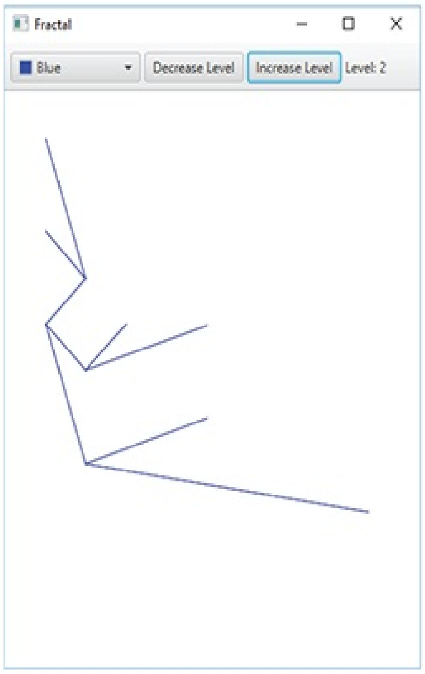
Figure 18.20 presents class `FractalController`, which defines the app’s event handlers and the methods for drawing the Lo Fractal recursively. The outputs show the development of the fractal from levels 0–5, then for levels 8 and 11. If we focus on one of the arms of this fractal, it will be identical to the whole image. This property defines the fractal to be strictly self-similar.

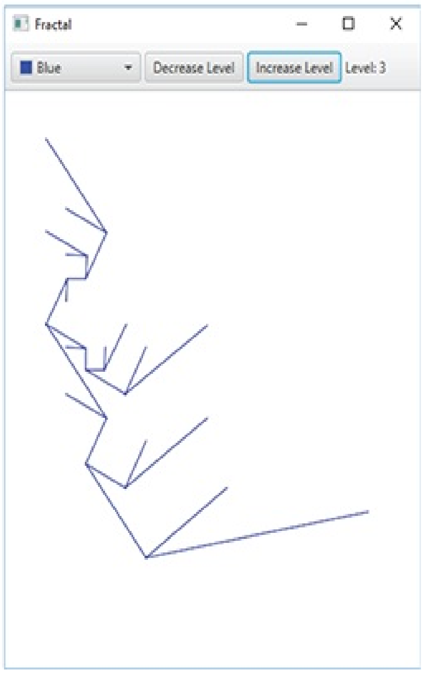
```
// Fig. 18.20: FractalController.java
// Drawing the "Lo feather fractal" using recurs
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.ColorPicker;
import javafx.scene.control.Label;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;

public class FractalController {
    // constants
    private static final int MIN_LEVEL = 0;
    private static final int MAX_LEVEL = 15;

    // instance variables that refer to GUI compo
    @FXML private Canvas canvas;
    @FXML private ColorPicker colorPicker;
    @FXML private Label levelLabel;

    // other instance variables
    private Color currentColor = Color.BLUE;
    private int level = MIN_LEVEL; // initial fra
    private GraphicsContext gc; // used to draw o

    // initialize the controller
    public void initialize() {
       levelLabel.setText("Level: " + level);
       colorPicker.setValue(currentColor); // sta
       gc = canvas.getGraphicsContext2D(); // get
       drawFractal();
    }

    // sets currentColor when user chooses a new
    @FXML
    void colorSelected(ActionEvent event) {
       currentColor = colorPicker.getValue(); //
       drawFractal();
    }

    // decrease level and redraw fractal
    @FXML
    void decreaseLevelButtonPressed(ActionEvent e
       if (level > MIN_LEVEL) {
          --level;
          levelLabel.setText("Level: " + level);
          drawFractal();
       }
    }

    // increase level and redraw fractal
```

```java
    @FXML
    void increaseLevelButtonPressed(ActionEvent e
        if (level < MAX_LEVEL) {
            ++level;
            levelLabel.setText("Level: " + level);
            drawFractal();
        }
    }

    // clear Canvas, set drawing color and draw t
    private void drawFractal() {
        gc.clearRect(0, 0, canvas.getWidth(), canv
        gc.setStroke(currentColor);
        drawFractal(level, 40, 40, 350, 350);
    }

    // draw fractal recursively
    public void drawFractal(int level, int xA, in
        // base case: draw a line connecting two g
        if (level == 0) {
            gc.strokeLine(xA, yA, xB, yB);
        }
        else { // recursion step: determine new po
            // calculate midpoint between (xA, yA)
            int xC = (xA + xB) / 2;
            int yC = (yA + yB) / 2;

            // calculate the fourth point (xD, yD)
            // isosceles right triangle between (xA
            // where the right angle is at (xD, yD)
            int xD = xA + (xC - xA) / 2 - (yC - yA)
            int yD = yA + (yC - yA) / 2 + (xC - xA)

            // recursively draw the Fractal
            drawFractal(level - 1, xD, yD, xA, yA);
            drawFractal(level - 1, xD, yD, xC, yC);
            drawFractal(level - 1, xD, yD, xB, yB);
        }
    }
}
```


Fractal
Blue
Decrease Level
Increase Level
Level: 0


Fractal
Blue
Decrease Level
Increase Level
Level: 1


Fractal
Blue
Decrease Level
Increase Level
Level: 2


Fractal
Blue
Decrease Level
Increase Level
Level: 3

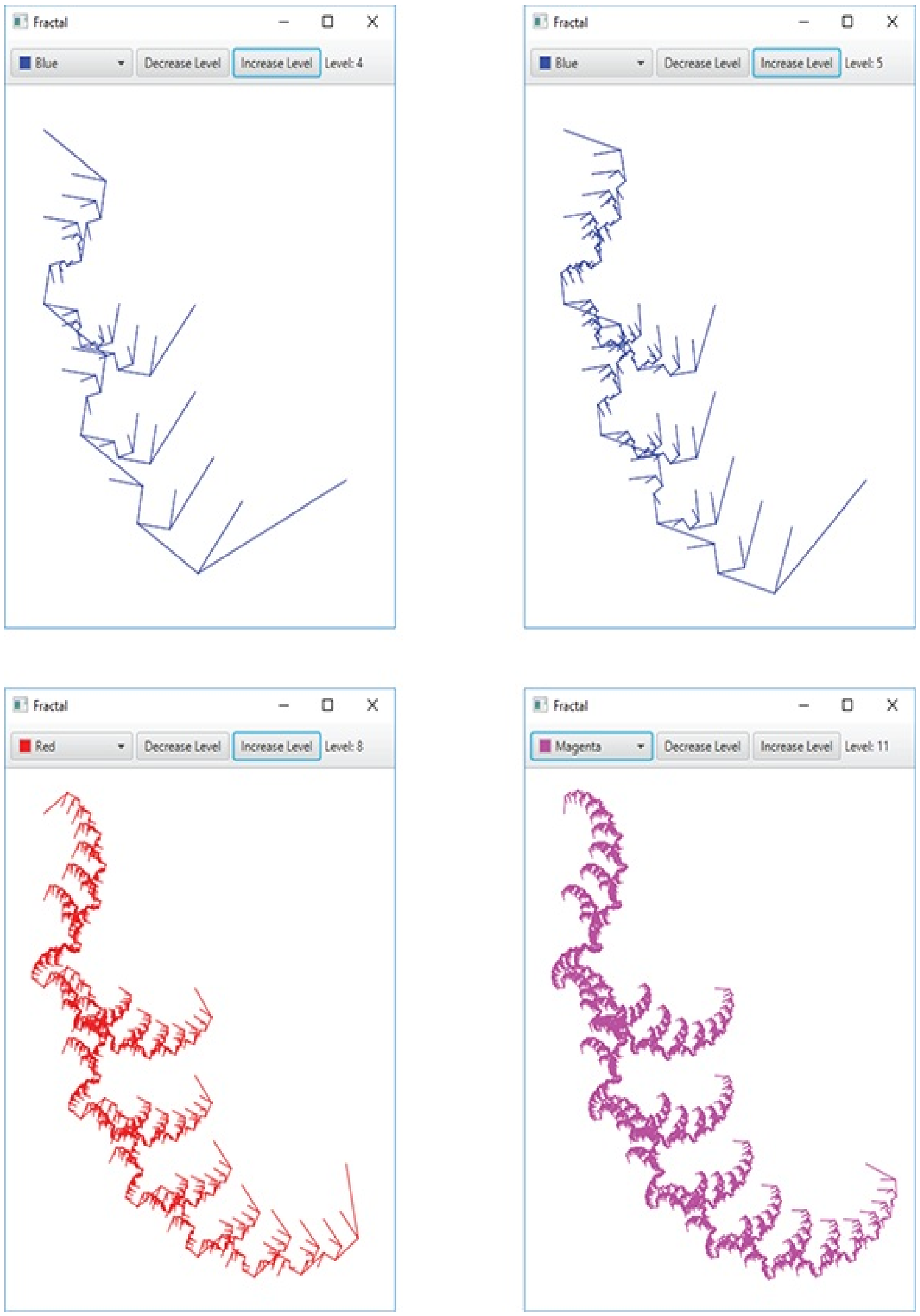


Fig. 18.20

Drawing the "Lo feather fractal" using recursion.

Description

`FractalController` Fields

Lines 14–25 declare the class’s fields:

- The constants `MIN_LEVEL` and `MAX_LEVEL` (lines 14–15) specify the range of Lo Fractal levels this app can draw. Once you get to level 15, the changes in the fractal (at this size) become difficult to see, so we set 15 as the maximum level. Around level 13 and higher, the fractal rendering becomes *slower* due to the details to be drawn.
- Lines 18–20 declare the `@FXML` instance variables that will reference the GUI’s controls that we programmatically manipulate. Recall that these are initialized by the `FXMLLoader`.
- Line 23 declares the `Color` variable `currentColor` and initialize it to the constant `Color.BLUE`. Class `Color` (package `javafx.scene.paint`) contains various constants for common colors and provides methods for creating custom colors.
- Line 24 declares the `int` variable `level`, which maintains the current fractal level.
- Line 25 declares the `GraphicsContext` variable `gc`, which will be used to draw on the app’s `Canvas`.

`initialize` Method

When the app’s controller is created and initialized, method initialize (lines 28–33):

- Sets the `levelLabel`’s initial text to indicate level 0.

- Sets the initial value of the `colorPicker` to `currentColor` (`Color.BLUE`).
- Gets the `canvas`’s `GraphicsContext` that will be used to draw lines in the currently selected color.
- Calls `drawFractal` (lines 63–67) to draw level 0 of the fractal.

`colorSelected` Event Handler

When the user choses a new color, the `colorPicker`’s `colorSelected` event handler (lines 36–40) uses `ColorPicker` method `getValue` to get the currently selected color, then calls `drawFractal` (lines 63–67) to redraw the fractal in the new color and current `level`.

`decreaseLevelButtonPressed` and `increaseLevelButtonPressed` Event Handlers

When the user presses the **Decrease Level** or **Increase Level** `Button`s, the corresponding event handler (lines 43–50 or 53–60) executes. These event handlers decrease or increase the level, set the `levelLabel`’s text accordingly and call `drawFractal` (lines 63–67) to redraw the fractal for the new `level` and `currentColor`. Attempting to decrease the `level` below `MIN_LEVEL` or above `MAX_LEVEL` does nothing in this app.

`drawFractal` Method with No Arguments

Each time method `drawFractal` is called, it uses `GraphicsContext` method `clearRect` to clear any prior drawing. This method clears a rectangular area by setting the contents of the specified area to the `Canvas`'s background color. The method's four arguments are the *x*- and *y*-coordinates of the rectangle's upper-left corner and the width and height of the rectangle. In this case, we clear the entire `Canvas`—methods `getWidth` and `getHeight` return the `Canvas`'s width and height, respectively.

Next, line 65 calls `GraphicsContext` method `setStroke` to set the drawing color to the `currentColor`. Then line 66 makes the first call to the overloaded *recursive* method `drawFractal` (lines 70–91) to draw the fractal.

`drawFractal` Method with Five Arguments

Lines 70–91 define the recursive method that creates the fractal. This method takes five parameters: the level of the Lo Fractal to draw and four integers that specify the *x*- and *y*-coordinates of two points. The base case for this method (line 72) occurs when `level` equals `0`, at which time line 73 uses `GraphicsContext` method `strokeLine` to draw a line between the two points the current call to `drawFractal` received as arguments.

In the recursive step (lines 75–90), lines 77–84 calculate

- (`xC`, `yC`)—the midpoint between (`xA`, `yA`) and (`xB`, `yB`), and
- (`xD`, `yD`)—the point that creates a right isosceles triangle with (`xA`, `yA`) and (`xC`, `yC`).

Then, lines 87–89 make three recursive calls on three different sets of points.

Since no lines will be drawn until the base case is reached, the distance between two points decreases on each recursive call. As the level of recursion increases, the fractal becomes smoother and more detailed. The shape of this fractal stabilizes as the level approaches 12—that is, the shape of the fractal remains approximately the same and the additional details become hard to perceive. Fractals will stabilize at different levels based on their shape and size.