Lab Exercise 13: Recursion
CS 2334

November 17, 2016

Introduction

In this lab, you will use recursive logic to create a fractal triangle, commonly known as a Sierpinski triangle. Recursion has many practical uses outside of graphics, but fractals provide a great way to visualize the concept.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Read existing code and documentation in order to complete an implementation
2. Define the base and recursive cases of a recursive formulation
3. Correctly call a method recursively
4. Create a timer for animation purposes

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Preparation

1. Import the existing lab 13 implementation into your eclipse workspace.

   (a) Download the lab 13 implementation:
   `http://www.cs.ou.edu/~fagg/classes/cs2334/labs/lab13/lab13.zip`

   (b) In Eclipse, select *File/Import*

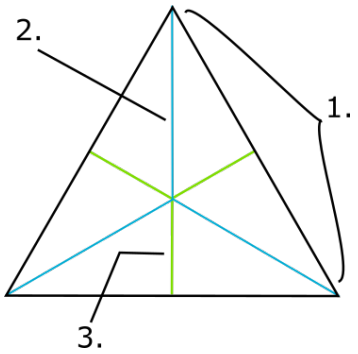
   (c) Select *General/Existing projects into workspace*. Click *Next*

   (d) Select *Select archive file*. Browse to the lab13.zip file. Click *Finish*

Sierpinski Triangle

The Sierpinski triangle is a fractal and attractive fixed set with the overall shape of an equilateral triangle, subdivided recursively into smaller equilateral triangles (from `https://en.wikipedia.org/wiki/Sierpinski_triangle`).

For this lab, there are some properties of equilateral triangles that you will need to know:

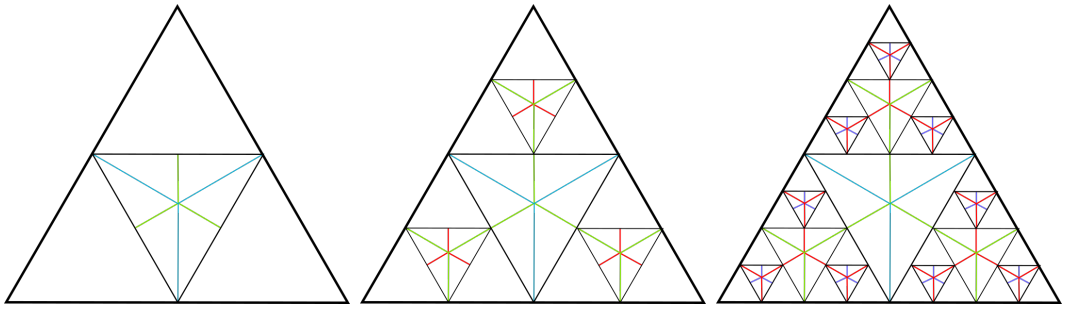


1. A side of the triangle, s.
2. The radius of the triangle, r.
3. The apothem of the triangle, a.

The following relations may be of some use to you:

1. Given the radius, a side is equal to: $s = \sqrt{3} \times r$
2. Equivalently, the radius is equal to: $r = \frac{s}{\sqrt{3}}$
3. The apothem is equal to: $a = \frac{\sqrt{3}}{6} \times s$

The first three iterations of the Sierpinski triangle are shown below (With radii and apothems shown—these won't be drawn in your implementation):



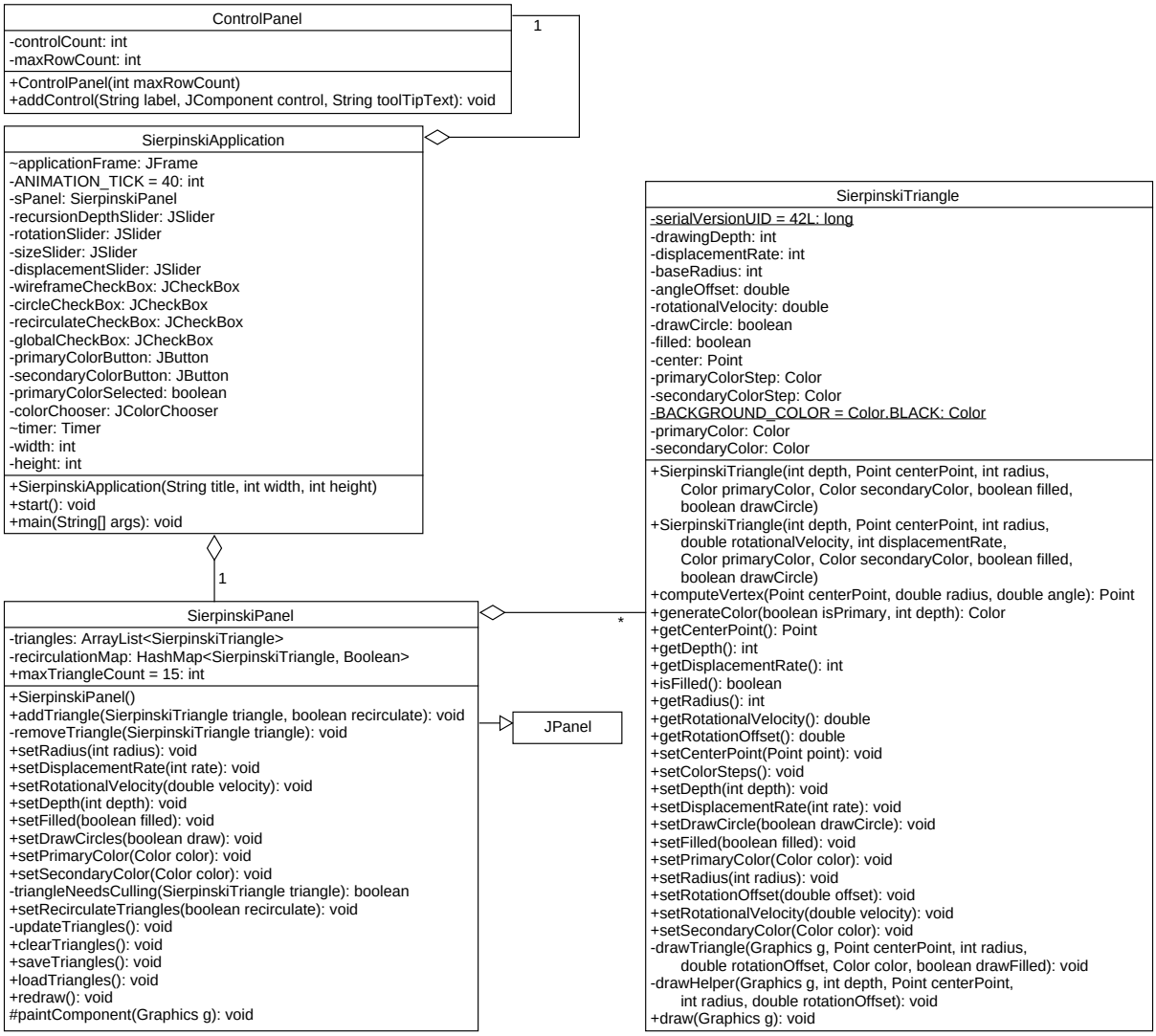
Excluding the black line segments, segments of the same color are the same length. Taking note of this, one can see that the radius of each of the three smaller triangles surrounding a larger triangle is equal to the larger triangle's apothem.

Furthermore, the distance between their center points and the larger triangle's center point is twice the apothem of the larger triangle.

The Sierpinski triangle can be constructed by first starting with an equilateral triangle in the standard orientation (sitting flatly on a side, one vertex pointing straight up). After drawing this, proceed to recursively draw smaller triangles rotated 180 degrees relative to the orientation of the base triangle. This can be done as follows:

1. The *draw()* method of the *SierpinskiTriangle* class is called.
2. Draw the base triangle (sitting flatly on a side, one vertex pointing straight up).
3. Call the *drawHelper()* method with the correct arguments to draw a triangle in the correct orientation (one vertex pointing straight down), in the correct position, with radius determined using the mathematical relations listed above.
4. In *drawHelper()*, compute the apothem of the triangle.
5. Compute the center points of the three surrounding triangles (Remember that all of these triangles are equilateral, so the triangle center points will be evenly spaced 120 degrees around the center triangle).
6. Go to step 3 for the three surrounding triangles
7. Draw the current triangle.

UML


ControlPanel
-controlCount: int
-maxRowCount: int
+ControlPanel(int maxRowCount)
+addControl(String label, JComponent control, String toolTipText): void
1
SierpinskiApplication
~applicationFrame: JFrame
-ANIMATION_TICK = 40: int
-sPanel: SierpinskiPanel
-recursionDepthSlider: JSlider
-rotationSlider: JSlider
-sizeSlider: JSlider
-displacementSlider: JSlider
-wireframeCheckBox: JCheckBox
-circleCheckBox: JCheckBox
-recirculateCheckBox: JCheckBox
-globalCheckBox: JCheckBox
-primaryColorButton: JButton
-secondaryColorButton: JButton
-primaryColorSelected: boolean
-colorChooser: JColorChooser
~timer: Timer
-width: int
-height: int
+SierpinskiApplication(String title, int width, int height)
+start(): void
+main(String[] args): void
1
SierpinskiPanel
-triangles: ArrayList<SierpinskiTriangle>
-recirculationMap: HashMap<SierpinskiTriangle, Boolean>
+maxTriangleCount = 15: int
+SierpinskiPanel()
+addTriangle(SierpinskiTriangle triangle, boolean recirculate): void
-removeTriangle(SierpinskiTriangle triangle): void
+setRadius(int radius): void
+setDisplacementRate(int rate): void
+setRotationalVelocity(double velocity): void
+setDepth(int depth): void
+setFilled(boolean filled): void
+setDrawCircles(boolean draw): void
+setPrimaryColor(Color color): void
+setSecondaryColor(Color color): void
-triangleNeedsCulling(SierpinskiTriangle triangle): boolean
+setRecirculateTriangles(boolean recirculate): void
-updateTriangles(): void
+clearTriangles(): void
+saveTriangles(): void
+loadTriangles(): void
+redraw(): void
#paintComponent(Graphics g): void
JPanel
*
SierpinskiTriangle
-serialVersionUID = 42L: long
-drawingDepth: int
-displacementRate: int
-baseRadius: int
-angleOffset: double
-rotationalVelocity: double
-drawCircle: boolean
-filled: boolean
-center: Point
-primaryColorStep: Color
-secondaryColorStep: Color
-BACKGROUND_COLOR = Color.BLACK: Color
-primaryColor: Color
-secondaryColor: Color
+SierpinskiTriangle(int depth, Point centerPoint, int radius,
Color primaryColor, Color secondaryColor, boolean filled,
boolean drawCircle)
+SierpinskiTriangle(int depth, Point centerPoint, int radius,
double rotationalVelocity, int displacementRate,
Color primaryColor, Color secondaryColor, boolean filled,
boolean drawCircle)
+computeVertex(Point centerPoint, double radius, double angle): Point
+generateColor(boolean isPrimary, int depth): Color
+getCenterPoint(): Point
+getDepth(): int
+getDisplacementRate(): int
+isFilled(): boolean
+getRadius(): int
+getRotationalVelocity(): double
+getRotationOffset(): double
+setCenterPoint(Point point): void
+setColorSteps(): void
+setDepth(int depth): void
+setDisplacementRate(int rate): void
+setDrawCircle(boolean drawCircle): void
+setFilled(boolean filled): void
+setPrimaryColor(Color color): void
+setRadius(int radius): void
+setRotationOffset(double offset): void
+setRotationalVelocity(double velocity): void
+setSecondaryColor(Color color): void
-drawTriangle(Graphics g, Point centerPoint, int radius,
double rotationOffset, Color color, boolean drawFilled): void
-drawHelper(Graphics g, int depth, Point centerPoint,
int radius, double rotationOffset): void
+draw(Graphics g): void

Lab 13: Specific Instructions

All of the necessary classes are provided in lab13.zip.

1. Look carefully through the existing code and implement any TODOs.
2. Do not add major functionality to the classes beyond what has been specified.
3. Don't forget to document as you go!

SierpinskiApplication class

- *SierpinskiApplication()* — At the bottom of the constructor a new javax.swing. Timer object needs to be created, using *SierpinskiApplication.ANIMATION_TICK* as the delay value. An ActionListener that calls code from a *SierpinskiPanel* instance should cause a step in the animation to occur (triangles move and are redrawn to the screen).

SierpinskiTriangle Class

- *draw(Graphics g)* — This method is responsible for actually drawing the Sierpinski triangle to the screen, using the provided Graphics object. Using the Color *SierpinsikiTriangle.BACKGROUND_COLOR*, draw the base triangle. Then, call drawHelper() with the appropriate arguments to draw the first, inner triangle, on top of the base triangle.
- *drawHelper(Graphics g, int depth, Point centerPoint, int radius, double rotationOffset)* — This is a recursive method that draws a triangle given the provided parameters, and then calls itself to draw surrounding triangles.

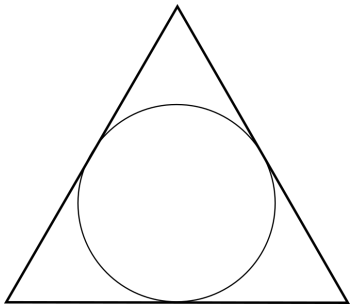
  This method takes in the following parameters:

  1. g — The Graphics object to use for drawing.
  2. depth — The current recursive depth. The initial call to this method should have depth equal to the triangle's depth field.
  3. centerPoint — The center point of the triangle to be drawn.
  4. radius — The radius of the triangle to be drawn.
  5. rotationOffset — The angle offset at which to draw this triangle.

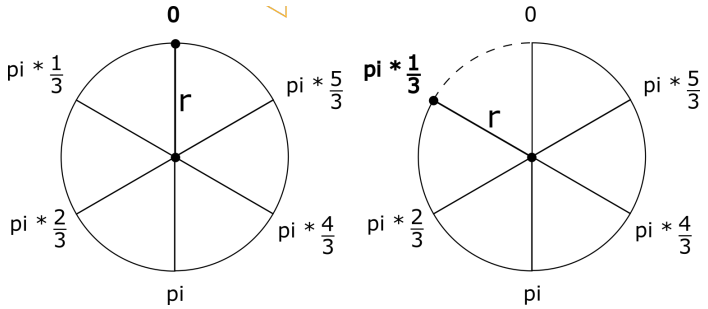
For each call to this method, use the color returned by *generateColor()* to draw the triangle (Triangles are drawn with the primary color).

If the *drawCircle* boolean has been set to true, you will need to draw an inscribed circle on top of this current triangle; use the color returned by the *generateColor()* method to do so (The circles are drawn with the secondary color).

An inscribed circle looks like the following:



The *SierpinskiTriangle* class has a helper method called *computeVertex()* that takes in a center point, a radius, and an angle. The method starts by generating a point *radius* units directly above the specified center point. The point is then rotated *angle* radians about the center point.



SierpinskiPanel Class

- *updateTriangles()* — This method is responsible for computing the new position and orientation of each triangle. You will need to complete the method implementation such that a triangle's center point and rotation offset are properly updated at each animation step.

Final Steps

1. Generate Javadoc using Eclipse.
   - Select *Project/Generate Javadoc...*
   - Make sure that your project (and all classes within it) is selected
   - Select *Private* visibility
   - Use the default destination folder
   - Click *Finish*.
2. Open the *lab13/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that that all of your classes are listed and that all of your documented methods have the necessary documentation.
3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission Instructions

- All required components (source code and compiled documentation) are due at 11:59:00pm on Friday, November 18th.
- Method 1: Submit through Eclipse
   1. From the *Window* menu, select *Preferences/Configured Assignment*.
   2. Select your project.
   3. From the Project menu, select *Submit Assignment*.
   4. Under *Select the assignment to submit*, select *Lab 13: Recursion*.
   5. Click *Change Username or Password...*. Enter your Web-Cat username and password. Click *OK*. You should only need to do this step once per session.
   6. Click *Finish*.
   7. Your browser should automatically open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

- Method 2: Submit directly to the Web-Cat server

  1. From the File menu, select *Export*.
  2. Select *Java/JAR File*. Click *Next*.
  3. Select and expand your project folder.
  4. Select your *src* and *doc* folders.
  5. Select *Export Java source files and resources*.
  6. Select an export destination location (e.g., your *Documents* folder/directory). This file should end in *.jar*
  7. Select *Add directory entries*.
  8. Click *Finish*.
  9. In your web browser, login to the Web-Cat server.
  10. Click the *Submit* button.
  11. Browse to your jar file.
  12. Click the *Upload Submission* button.
  13. The next page will give you a list of all files that you are uploading. If you selected the correct jar file, then click the *Confirm* button.
  14. Your browser will then open a Web-Cat page that shows your submission being graded. After a short wait, the page will show a report of your submission. See the main class web page for a link that describes the Web-Cat output.

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 45 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is proportional to the fraction of tests that your code passes (so 22.5 points means that your code passed half of the tests)

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 35 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code (up to 15 points)

If you do not submit compiled Javadoc for your lab, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions (where points may be deducted).

Bonus: up to 5 points

You will earn one bonus point for every two hours that your assignment is submitted early.

Penalties: up to 100 points

You will lose ten points for every minute that your assignment is submitted late.