Project 3: Animated L-System

CAP 3027 - Fall 2024

textbfLast Updated: 10/9/24

Updated 10/9/24: Typo corrections.

This assignment introduces using turtle graphics to draw L-Systems, which can be used to procedurally generate mathematical and artistic images. We also use `EventListeners` to provide interaction to users, and Threads to efficiently run our code.

Assignment Overview

In this exercise, you will create an animated viewer for L-systems. Users will be able to interact with the visualization by panning and zoomimg into the L-system. These are the five main features of this assignment:

1. generating the L-system string
2. drawing the L-system
3. animating the L-system with proper threading
4. zooming and panning in the viewer

1 L-System Generation

Our L-systems are generated based on four main components:

- An *alphabet*, or a series of characters with specific meanings.
- A *starting axiom*, or the initial state of the system.
- A *set of rules* that are iteratively applied to our current axiom. Consists of two parts, a character and a string which will replace that character when we iterate.
- N, the number of times to iterate.

There are also parameters that affect how the system is drawn:

- L, the length of a segment
- W, the width of a segment
- A, a turn angle

To generate our L-system, we iterate over the starting axiom and replace characters with rules if present, or copy them over if no rule exists. We do this N times.

1.1 Alphabet

Our L-system alphabet contains five symbols and a default case for symbols not in the alphabet. These symbols will transform a "turtle" (e.g., a `Graphics2D`) and draw on an image.

Symbol	**Meaning**
>	The turtle rotates A degrees clockwise.
<	The turtle rotates A degrees anticlockwise.
+	The current width W is increased by 1
-	The current width W is decreased by 1
[	The turtle saves its current location and rotation (i.e., *AffineTransform*) and W to its list.
]	The turtle looks at its list, returns to the latest orientation (i.e., *AffineTransform*) and W written down, and removes them from the list
*	The turtle draws a red circle of radius $L/4$ **centered at its current position.**
default (any other symbol)	The turtle moves forward L pixels and draws a green line with stroke width W between its new spot and old spot.

1.2 Starting Axiom

The starting set of symbols we will iterate on in cases where $N>0$.
*e.g., *A*>B**

1.3 Rule Set

The set of rules we will apply to the axiom. In our project, we will only use a maximum of two rules. At each iteration, we replace the character on the left with the string on the right.
e.g. $A \rightarrow A$*>B, $B \rightarrow BB$

1.4 Example Iterations

At each step, A is replaced with A*>B and B is replaced with BB (these are the rules). Any character that does not have a rule is simply copied over to the next iteration. Using the previous starting axiom and rules,

- N=1: *A*>B*>BB*
- N=2: *A*>B*>BB*>BBBB*
- N=3: *A*>B*>BB*>BBBB*>BBBBBBBB*
- N=4: *A*>B*>BB*>BBBB*>BBBBBBBB*>BBBBBBBBBBBBBBBB*

2 L-system Drawing with Turtle Graphics

After generating your final L-system string, we will draw it one step at a time. Each symbol in the alphabet will be converted into some "command" for the turtle (a `Graphics2D` object).

You *could* implement drawing the system by hand, keeping track manually of the turtle's coordinates and orientation. However, we can also leverage affine transformations to move our turtle and bypass manual calculations.

2.1 Graphics2D Overview

`Graphics2D` has a set of functions you should use which will automatically apply transformations to everything it draws. An alternative way of thinking about this is that we can modify the coordinate space on which we are drawing; we can tell the turtle to start at a different location and orientation. We are also able to get its current transform, or set the current transformation to some known affine transform. The relevant functions include:

- `translate(x,y)`: translates the "pen" by x and y
- `rotate(theta)`: rotates the "pen" theta radians
- `scale(sx,sy)`: scales the "pen" by sx and sy
- `getTransform()`: gets the current transform for the pen
- `setTransform(AffineTransform at)`: sets the current transform to `at`

We will also need only two draw commands to implement our alphabet.

- `drawLine(x1,y1,x2,y2)`: draws a line between coordinates (x1,y1) and (x2,y2)
- `fillOval(x,y,w,h)`: draws an oval of dimensions wxh with its corner at coordinate (x,y)

2.2 Training the Turtle

Each time you draw the system, you will want to get a Graphics2D (turtle) for the image you are drawing. You will then put the turtle in the correct starting spot, and then it will move and draw by reading the L-system string.

2.2.1 Turtle Placement

Our turtle should begin in the middle of the image, with its head facing "up". It should think that the positive X direction is to its right. Recall that by default, the turtle is placed at the top-left corner of the image and faces down. You will need to apply a series of transformations to the turtle to move it to the correct location, and orient it's sense of direction.

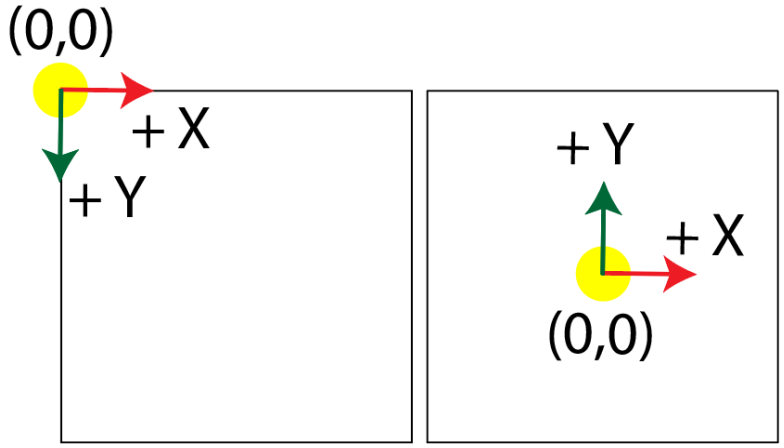

Figure 1: **Left**: The turtle's default placement and perspective. **Right:** How we want the turtle oriented when it begins to draw the L-system.

2.2.2 Turtle Tricks

Each symbol in the alphabet should be able to be translated to:

1. the turtle being transformed
2. the turtle drawing something
3. the turtle saving something to memory
4. some combination of these

Transformation and drawing commands should use the previously discussed functions. Turtle memory may be implemented using any of Java's prebuilt data structure. Possible solutions include arrays, `Stack`, or `ArrayList`.

3 Animating the L-System

The above sections should allow you to draw static L-systems. However, we want to animate their growth over time. To do this, you will use a Timer to repeatedly draw more and more of the system. You should also use proper threading to maintain a responsive application.

3.1 Method

When a new system is generated, the Timer should start by drawing just the first step of the system. Then, on the next trigger, it should start from the beginning again and draw the first two steps. Then, the first three steps. This continues until you reach the last character. After that, you should redraw the whole system only when the user pans or zooms (see Section X).

Your timer should default to a delay of 50ms, and this delay should change if the Speed slider is adjusted. You should implement its `ChangeListener` to do this.

You **should not** regenerate the L-system each time the Timer runs. Depending on the system configuration, this could result is poor performance! You should make the entire string representing the system before drawing, and then adjust *how much* of that string you are drawing each time the Timer triggers.

There is also an option to turn or off drawing the turtle. If the "Draw Turtle" checkbox is selected, you should draw the turtle at its current location at the end of every frame. A `DrawTurtle` function is provided for you.

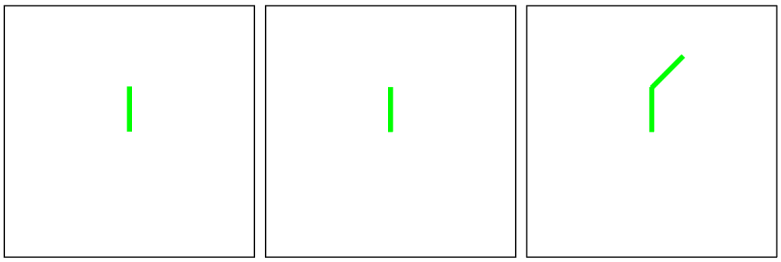

Figure 2: The first three frames of the L-system "A>B>A>B" being draw. **Left**: Frame 1 of the L-system. Only the A character is processed so a line segment is drawn. **Center:** Frame 2 of the L-system. Starting with a fresh image, A is processed and the line drawn again. Then, > is processed and the turtle turns left (but nothing is drawn). **Right**: Frame 3 of the L-system. From a fresh image, A is drawn, the turtle turns, then B is drawn.

3.2 Threading

You should perform all L-system generation and drawing for your frames on their own threads. Then, when the image is finished and sent to the display, repaint

the frame back on Swing's EventDispatchThread using `SwingUtilities.invokeLater();`

4 Interacting with the Viewer

You should add two `EventListeners` that will allow for mouse click and mouse wheel inputs. When either occurs, the image should update automatically. These should be added to the `JPanel` displaying the system.

4.1 Panning

Use a `MouseMotionListener` to implement panning on the image. When the user moves their mouse, a `MouseEvent` is passed to the functions inside the listener. Of interest to us is the `mouseDragged` method which runs when the user moves the mouse while pressed. We will want to keep track of how the mouse is moving between executions of this function and transform our turtle accordingly when drawing the image (i.e., translating it by some amount proportional to the amount of movement).

Panning should work independent of scale. At high levels of zooming, the image should still move the same distance relative to the mouse. It should not move at an accelerated or decelerated rate.

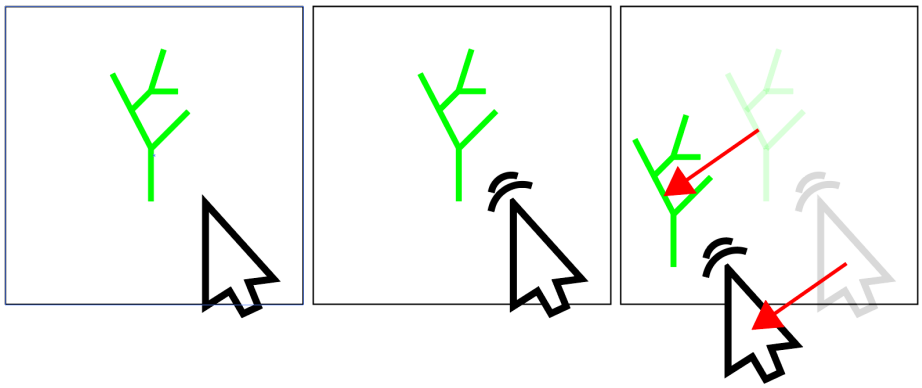

Figure 3: How panning should function. The user will be able to mouse over the image, click, and drag to move the drawing. The drawing should move the same distance as the drag.

4.2 Zooming

Use a `MouseWheelListener` to implement zooming on the image. The zoom should center on the **middle** of the image. The precise values/method is up to you. Try different methods (adding/subtracting a constant amount of scaling, multiplying the previous scale values by some constant values, etc.). Your zoom should be smooth and happen as many times as the user wants (i.e., **you cannot just zoom once** to a smaller/larger scale, the user should be able to zoom as small or large as they want).

When the user scrolls their mouse wheel (or zooms in and out on a trackpad), a `MouseWheelEvent` is passed to the functions inside the listener. The only method for a `MouseWheelListener` is the `mouseWheelMoved` method which runs when the user scrolls their mouse wheel or uses the trackpad to zoom. We can see the direction of scrolling by checking data in the event, and then scale up or down our turtle accordingly.

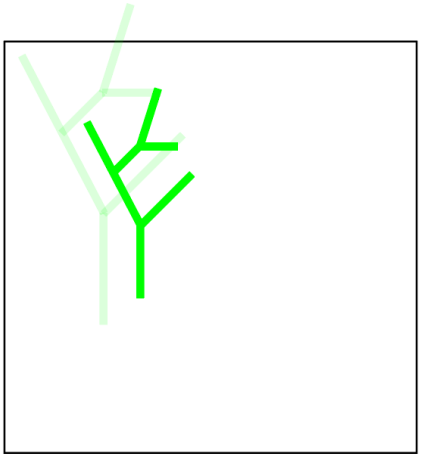

Figure 4: How zooming should function. When the user scrolls, the drawing is scaled around the center of the image. Here, the pale green shows an L-system drawing before zooming out, and the bright green after zooming out.

5 Required Functions

You must implement a required function, which will be tested by graders. This should also be used in the implementation of the walker. The signature for this function is included in the code base, and should not be modifed.

- `static String Generate_LSystem(int iterations, String start, HashMap<Character,String> rules)`: This function takes in the number of iterations to perform on the starting axiom, a String representing the starting axiom, and a HashMap mapping characters to their rules. It calculates and returns the resulting L-system String.

6 Submission and Grading

6.1 Submission

You should submit a zipped folder named in the format `LASTNAME_FIRSTNAME_PROJECT3.zip`. For example, a student named John Doe should submit a zipped folder called `DOE_JOHN_PROJECT2.zip`. Inside the folder should be your updated `Project3_Codebase.java` and `Project3_Functions.java` files. You may also submit an optional `README.txt` with any sources referenced or other information that would be useful to the graders. Besides these three files, nothing else should be in the folder. Up to **10 points** may be deducted for failure to follow proper naming requirements.

6.2 Grading

This project will be graded by both visual inspection of the generated images and unit testing of the required functions. We will run your project with different configurations, and visually verify that the output looks correct. **Points may be deducted if your code runs incorrectly, fails to run, takes an exceedingly long time to execute, or if a specific requirement is not met.**

Feature	Criteria	Points
Animated L-system Drawing	The correct L-system animation feature is demonstrated by running the application. **If animation is not featured, half the points (25) can be earned for displaying a static image.**	50
Panning	The correct panning behavior is implemented. The image moves the same relative distance as the mouse. **Half the points can be earned for behavior that is close to correct, but slightly errored (7.5) (e.g., the panning is accelerated/slowed based on the current level of zoom.**	15
Zooming	The correct zooming behavior is implemented. Scrolling zooms in on the center of the image, and the scroll happens as far in or far out as the user wants. **Half the points can be earned for behavior that is close to correct, but slightly errored (7.5).**	15
Threading	Threading is correctly used in the program. Your program and UI should remain responsive when drawing or animating the system.	10
Unit Tests - Required Functions	Tests will compare the output of your code to the correct expected values. Unlike Projects 1 and 2, there is no required method to use to achieve this. However, if your tests take an abnormally long amount of points to run **you may lose points.**	10