PROBLEM 2 - ▲■

(LINK)

This code creates a dynamic, interactive fractal generator using either squares or triangles, depending on user selection. The visual logic and recursive pattern are inspired by “Recursive Triangle” by Hiraga (link), adapted and extended here to allow both square and triangle tilings with interactive controls. At its core, the script recursively draws grids of either deformed squares or tessellated equilateral triangles, each filled with a complex "frame" pattern created through recursive subdivision.

The process started by playing with a code base that allowed me to see the workings and logic behind fractal generation. (link)

Upon startup, the program defines key global variables such as frac (which sets the recursion depth), scaleX (which determines the deformation of the fractal's inner shapes), and shapeType (which toggles between 'square' and 'triangle' modes via GUI buttons). The canvas is sized to fit the browser window, with color and style settings established in setup(). GUI elements are built with createGUI(), letting users increase or decrease the fractal depth (frac) and switch between square and triangle tilings in real time.

The function setGridParams() calculates the geometry and centering offsets (offsetX, offsetY) needed to fill the canvas with a uniform grid, adjusting for the chosen shape. For squares, the grid is built by fitting a set number of rows (nRows) vertically, then filling horizontally. For triangles, the algorithm computes the side length so that the desired number of triangle rows fits vertically, and then arranges the triangles in a honeycomb pattern, shifting every other row by half a cell for tight tiling.

The run() function clears the canvas and triggers the main drawing process, calling either the square or triangle rendering logic as needed. For squares, nested loops fill the canvas with deformed squares, each recursively subdivided using drawDeformedSquare(). This function creates a "frame" by drawing both the outer and inner square at each recursion, and then fills the gaps with smaller triangles drawn via drawRecursiveTriangle(). The recursion continues until the base case, which simply draws the innermost shape.

For triangles, the drawAllTriangles() function covers the canvas with alternating upward- and downward-pointing equilateral triangles. Each triangle is recursively drawn using drawDeformedTriangle(), which works analogously to the square version, generating inner and outer triangles and recursively subdividing the pattern.

User interaction is seamless: moving the mouse horizontally alters the scaleX value, dynamically changing the deformation of each fractal layer (mouseMoved()), while resizing the window re-triggers the drawing to keep the pattern centered (windowResized()). All code is modular and clearly divided by function, making it easy to extend with new features or shapes in the future. Overall, this code is a concise and flexible demonstration of recursive geometric art, combining mathematical rigor with interactive aesthetics.

PSEUDOCODE

1. Define global variables: frac (recursion depth), scaleX (deformation), side (cell size), nRows (grid rows), nCols (grid columns), offsetX/offsetY (grid offsets), gui_buttons
(array for buttons), shapeType (square or triangle)

2. On program start:
3. - Create canvas that fills the window
4. - Disable shape filling and set stroke color to pale blue
5. - Call setGridParams to calculate grid sizing and offsets
6. - Call createGUI to add interactive buttons for recursion and shape type
7. - Call run to draw the initial fractal grid

8. Function setGridParams:
9. - If shapeType is 'square':
10. -- Calculate cell side so nRows of squares fit vertically
11. -- Compute number of columns so grid fills the canvas
12. -- Compute offsets to center square grid
13. - Else if shapeType is 'triangle':
14. -- Calculate ideal triangle height so nRows fit vertically
15. -- Calculate side length from height (for equilateral triangles)
16. -- Compute number of columns to fill canvas (add buffer columns)
17. -- Compute offsets to center triangle grid

18. Function run:
19. - Clear canvas with black background
20. - Call setGridParams to ensure updated layout
21. - If shapeType is 'square':
22. -- For each row in grid (0 to nRows-1):
23. --- For each column in grid (0 to nCols-1):
24. ---- Compute four corners of current square
25. ---- Call drawDeformedSquare(frac, x0, y0, x1, y1, x2, y2, x3, y3)
26. - Else if shapeType is 'triangle':
27. -- Call drawAllTriangles to fill canvas with tessellated triangles

28. Function drawAllTriangles:
29. - Compute triangle height (h) from current side length
30. - Compute number of rows/columns to ensure coverage (add buffer)
31. - Compute offsetX/offsetY for centering grid
32. - For each row in grid (0 to nRows2-1):
33. -- Compute y position of row and horizontal shift (sl) for honeycomb effect
34. -- For each column in row (0 to nCols2-1):
35. --- Compute x position with shift
36. --- If triangle (upwards) is within/near canvas:
37. ---- Compute three vertices (left, top, right)
38. ---- Call drawDeformedTriangle(frac, bx, by, ax, ay, cx, cy)
39. --- If triangle (downwards) is within/near canvas:
40. ---- Compute three vertices (bottom, left, right)
41. ---- Call drawDeformedTriangle(frac, ax2, ay2, cx2, cy2, bx2, by2)

42. Function mouseMoved:
43. - Map mouseX to deformation factor scaleX
44. - Call run() to redraw fractal with new deformation

45. Function drawDeformedSquare(n, x0, y0, x1, y1, x2, y2, x3, y3):
46. - If n == 0:
47. -- Draw innermost square using vertices (base case)

48. -- Return
49. - Compute four inner (deformed) corners by interpolating between outer corners (lerp, scaleX)
50. - Draw outer and inner square shapes
51. - Call drawRecursiveTriangle on each of the four "wing" triangles between outer and inner square
52. - Recursively call drawDeformedSquare with inner corners for next step (n-1)

53. Function draw Recursive Triangle(n, x0, y0, x1, y1, x2, y2):
54. - If n <= 0:
55. -- Draw innermost triangle (base case)
56. -- Return
57. - Compute three inner (deformed) corners (lerp, scaleX)
58. - Draw outer and inner triangles
59. - Call drawRecursiveTriangle on each of the three "wing" triangles between outer and inner triangle
60. - Recursively call drawRecursiveTriangle with inner corners for next step (n-1)

61. Function draw Deformed Triangle(n, x0, y0, x1, y1, x2, y2):
62. - If n == 0:
63. -- Draw triangle itself (base case)
64. -- Return
65. - Compute three inner (deformed) corners (lerp, scaleX)
66. - Draw outer and inner triangle shapes
67. - Call drawDeformedTriangle recursively on three "wing" triangles
68. - Recursively call drawDeformedTriangle with inner corners for next step (n-1)

69. Function createGUI:
70. - Create buttons for: recursion +/-, square/triangle switch
71. - Assign mouse callbacks to update frac/shapeType and call run()
72. - Position buttons in lower left of canvas

73. Function windowResized:
74. - Resize canvas to new window size
75. - Call run() to redraw everything

CODE

```
// Depth of recursion for the fractal structure (number of fractal steps)
let frac = 4;

// Deformation factor (how "tight" the inner shapes are drawn)
let scaleX = 0.75;

// Size of each module (square or triangle)
let side;

// Number of rows for the grid (affects both squares and triangles)
let nRows = 4;

// Variables for the number of columns and offsets to center the grid
let nCols, offsetX, offsetY;

// Array for storing GUI buttons
let gui_buttons = [];

// Shape selector: can be 'square' or 'triangle' (switchable by GUI)
let shapeType = "square";

// Setup function: called once at the start, sets up the canvas, GUI and runs the drawing logic
function setup() {
  createCanvas(windowWidth, windowHeight); // Make the canvas fit the window
  noFill();
  stroke(240, 250, 255, 90);               // Light blueish-white stroke for lines
  setGridParams();                         // Calculate parameters for the current grid
  createGUI();                             // Add interactive buttons for the user
  run();                                   // Start the first draw
}

// Function to calculate grid parameters for squares or triangles
function setGridParams() {
  if (shapeType === 'square') {
    // For squares: side is set so that nRows fit vertically, columns calculated accordingly
    side = min(width, height) / nRows;
    nCols = ceil(width / side);
    // Center the grid in the canvas
    offsetX = (width - nCols * side) / 2;
    offsetY = (height - nRows * side) / 2;
  } else if (shapeType === 'triangle') {
    // For triangles: calculate the side so nRows of triangles fit vertically
    let h_ideal = height / nRows;
    side = h_ideal * 2 / sqrt(3); // The formula for equilateral triangle height
    offsetY = (height - nRows * h_ideal) / 2;
    nCols = ceil(width / side) + 2; // Add 2 to ensure full coverage
    offsetX = (width - (nCols * side)) / 2;
  }
}

// Main function that actually draws the grid of squares or triangles, depending on selection
function run() {
  background(0);       // Clear the canvas with black
  setGridParams();     // (Re)calculate grid parameters
  if (shapeType === 'square') {
    // Draw a grid of squares, each of which is recursively decorated with fractal pattern
    for (let y = 0; y < nRows; y++) {
      for (let x = 0; x < nCols; x++) {
        // Compute the four corners of the square at position (x, y)
        let x0 = offsetX + x * side;
        let y0 = offsetY + y * side;
        let x1 = x0 + side;
        let y1 = y0;
        let x2 = x1;
        let y2 = y0 + side;
        let x3 = x0;
        let y3 = y2;
        // Draw the recursive, deformed square
        drawDeformedSquare(frac, x0, y0, x1, y1, x2, y2, x3, y3);
      }
    }
  } else if (shapeType === 'triangle') {
    // Fill the canvas with a tessellation of triangles
    drawAllTriangles();
  }
}

// This function covers the whole canvas with a honeycomb-like tiling of up and down-pointing triangles
function drawAllTriangles() {
  // Calculate the height of an equilateral triangle with the given side
  let h = side * sqrt(3) / 2;
  // Number of rows and columns needed to fully cover the canvas (with buffer)
  let nRows2 = ceil(height / h) + 2;
  let nCols2 = ceil(width / side) + 2;
  // Offset values for centering the grid
  let offsetX = (width - (nCols2 * side)) / 2;
  let offsetY = (height - nRows2 * h) / 2;

  // Loop through all rows (each row is shifted horizontally by half a side for the honeycomb effect)
  for (let row = 0; row < nRows2; row++) {
    let y = offsetY + row * h; // Y position of the current row
    let sl = (row % 2) * (side / 2); // Odd rows are horizontally shifted

    for (let col = 0; col < nCols2; col++) {
      // X position of the current triangle (with row shift applied)
      let x = offsetX + col * side + sl;

      // --------- Upward-pointing triangle ---------
      // Only draw if the triangle is within or close to the visible canvas area
      if (y >= -h && y <= height + h && x >= -side && x <= width + side) {
        // Compute the vertices: left base, top, right base
        let ax = x;
        let ay = y + h;
        let bx = x + side / 2;
        let by = y;
        let cx = x + side;
        let cy = y + h;
        // Recursively draw a deformed upward triangle
        drawDeformedTriangle(frac, bx, by, ax, ay, cx, cy);
      }

      // --------- Downward-pointing triangle ---------
      // Only draw if the bottom point is within or near the visible area
      if (y + h < height + h && x >= -side && x <= width + side) {
        // Compute the vertices: bottom, left base, right base
        let ax2 = x + side / 2;
        let ay2 = y + h * 2;
        let bx2 = x;
        let by2 = y + h;
        let cx2 = x + side;
        let cy2 = y + h;
        // Recursively draw a deformed downward triangle
        drawDeformedTriangle(frac, ax2, ay2, cx2, cy2, bx2, by2);
      }
    }
  }
}

// This function adjusts the deformation factor according to the mouse position and redraws everything
function mouseMoved() {
  scaleX = map(mouseX, 0, width, 0.2, 0.85); // Map mouse X to deformation
  run();
}

// Recursively draws a "fractal frame" square, with inner and outer shapes
function drawDeformedSquare(n, x0, y0, x1, y1, x2, y2, x3, y3) {
  if (n == 0) {
    // Base case: draw only the innermost square
    beginShape();
    vertex(x0, y0); vertex(x1, y1);
    vertex(x2, y2); vertex(x3, y3);
    endShape(CLOSE);
    return;
  }
  // Compute the four corners of the inner (deformed) square
  let nx0 = lerp(x0, x1, scaleX);
  let ny0 = lerp(y0, y1, scaleX);
  let nx1 = lerp(x1, x2, scaleX);
  let ny1 = lerp(y1, y2, scaleX);
  let nx2 = lerp(x2, x3, scaleX);
  let ny2 = lerp(y2, y3, scaleX);
  let nx3 = lerp(x3, x0, scaleX);
  let ny3 = lerp(y3, y0, scaleX);

  // Draw the main square and its inner deformed square (the "frame" effect)
  beginShape();
  vertex(x0, y0); vertex(x1, y1); vertex(x2, y2); vertex(x3, y3);
  endShape(CLOSE);
  beginShape();
  vertex(nx0, ny0); vertex(nx1, ny1); vertex(nx2, ny2); vertex(nx3, ny3);
  endShape(CLOSE);

  // Recursively draw fractal triangles in each of the four "wings" of the frame
  drawRecursiveTriangle(n - 1, x0, y0, nx0, ny0, nx3, ny3);
  drawRecursiveTriangle(n - 1, x1, y1, nx1, ny1, nx0, ny0);
  drawRecursiveTriangle(n - 1, x2, y2, nx2, ny2, nx1, ny1);
  drawRecursiveTriangle(n - 1, x3, y3, nx3, ny3, nx2, ny2);

  // Recursively continue with the inner square
  drawDeformedSquare(n - 1, nx0, ny0, nx1, ny1, nx2, ny2, nx3, ny3);
}

// Recursively draws fractal triangles with the same "frame" logic (for the wings and center)
function drawRecursiveTriangle(n, x0, y0, x1, y1, x2, y2) {
  if (n <= 0) {
    // Base case: draw only the innermost triangle
    beginShape();
    vertex(x0, y0); vertex(x1, y1); vertex(x2, y2);
    endShape(CLOSE);
    return;
  }
  // Compute deformed (inner) triangle vertices
  let nx0 = lerp(x0, x1, scaleX);
  let ny0 = lerp(y0, y1, scaleX);
  let nx1 = lerp(x1, x2, scaleX);
  let ny1 = lerp(y1, y2, scaleX);
  let nx2 = lerp(x2, x0, scaleX);
  let ny2 = lerp(y2, y0, scaleX);

  // Draw main triangle and its inner triangle (the "frame" effect)
  beginShape();
  vertex(x0, y0); vertex(x1, y1); vertex(x2, y2);
  endShape(CLOSE);
  beginShape();
  vertex(nx0, ny0); vertex(nx1, ny1); vertex(nx2, ny2);
  endShape(CLOSE);

  // Recursively draw the three "wings" of the frame
  drawRecursiveTriangle(n - 1, x0, y0, nx0, ny0, nx2, ny2);
  drawRecursiveTriangle(n - 1, x1, y1, nx1, ny1, nx0, ny0);
  drawRecursiveTriangle(n - 1, x2, y2, nx2, ny2, nx1, ny1);

  // Recursively continue with the inner triangle
  drawRecursiveTriangle(n - 1, nx0, ny0, nx1, ny1, nx2, ny2);
}

// Recursively draws the deformed triangle for use in grid filling (the base for the fractal tiling)
function drawDeformedTriangle(n, x0, y0, x1, y1, x2, y2) {
  if (n == 0) {
    // Base case: draw the triangle itself
    beginShape();
    vertex(x0, y0); vertex(x1, y1); vertex(x2, y2);
    endShape(CLOSE);
    return;
  }
  // Compute deformed (inner) triangle vertices
  let nx0 = lerp(x0, x1, scaleX), ny0 = lerp(y0, y1, scaleX);
  let nx1 = lerp(x1, x2, scaleX), ny1 = lerp(y1, y2, scaleX);
  let nx2 = lerp(x2, x0, scaleX), ny2 = lerp(y2, y0, scaleX);

  // Draw main triangle and its inner triangle (the "frame" effect)
  beginShape();
  vertex(x0, y0); vertex(x1, y1); vertex(x2, y2);
  endShape(CLOSE);
  beginShape();
  vertex(nx0, ny0); vertex(nx1, ny1); vertex(nx2, ny2);
  endShape(CLOSE);

  // Recursively draw the three "wings" of the frame
  drawDeformedTriangle(n - 1, x0, y0, nx0, ny0, nx2, ny2);
  drawDeformedTriangle(n - 1, x1, y1, nx1, ny1, nx0, ny0);
  drawDeformedTriangle(n - 1, x2, y2, nx2, ny2, nx1, ny1);

  // Recursively continue with the inner triangle
  drawDeformedTriangle(n - 1, nx0, ny0, nx1, ny1, nx2, ny2);
}

// GUI creation for interactive buttons (recursion depth and shape selection)
function createGUI() {
  let gui_w = "25px";
  let gui_h = "25px";
  let blankY = 10;
  let runningX = width * 0.05;

  let l_plus_b = createButton("+");
  l_plus_b.mousePressed(function() {
    if (frac < 20) frac++;
    run();
  });
  l_plus_b.style("width", gui_w);
  l_plus_b.style("height", gui_h);
  l_plus_b.style("background", "  #333");
  l_plus_b.style("color", "  #fff");
  l_plus_b.position(runningX, height * 0.95 - blankY);
  gui_buttons.push(l_plus_b);

  let l_minus_b = createButton("-");
  l_minus_b.mousePressed(function() {
    if (frac > 0) frac--;
    run();
  });
  l_minus_b.style("width", gui_w);
  l_minus_b.style("height", gui_h);
  l_minus_b.style("background", "  #333");
  l_minus_b.style("color", "  #fff");
  l_minus_b.position(runningX + 40, height * 0.95 - blankY);
  gui_buttons.push(l_minus_b);

  // Button for squares
  let squareBtn = createButton("■");
  squareBtn.mousePressed(function() {
    shapeType = 'square';
    run();
  });
  squareBtn.style("width", gui_w);
  squareBtn.style("height", gui_h);
  squareBtn.style("background", "  #333");
  squareBtn.style("color", "  #fff");
  squareBtn.position(runningX + 90, height * 0.95 - blankY);
  gui_buttons.push(squareBtn);

  // Button for triangles
  let triBtn = createButton("▲");
  triBtn.mousePressed(function() {
    shapeType = 'triangle';
    run();
  });
  triBtn.style("width", gui_w);
  triBtn.style("height", gui_h);
  triBtn.style("background", "  #333");
  triBtn.style("color", "  #fff");
  triBtn.position(runningX + 130, height * 0.95 - blankY);
  gui_buttons.push(triBtn);
}

// This function ensures the canvas always fits the window, and redraws the fractal accordingly
function windowResized() {
  resizeCanvas(windowWidth, windowHeight);
  run();
}

/*
-------------------------------------------------------------
drawAllTriangles SUMMARY:
-------------------------------------------------------------
This function completely fills the canvas with an alternating "honeycomb" tiling of equilateral triangles, both upward and downward.
Each triangle is then recursively decorated with a fractal "frame" effect (drawDeformedTriangle), producing a complex, detailed pattern.
Offset and coverage calculations ensure there are **no empty spaces** at the edges, regardless of canvas size.
-------------------------------------------------------------
*/
```