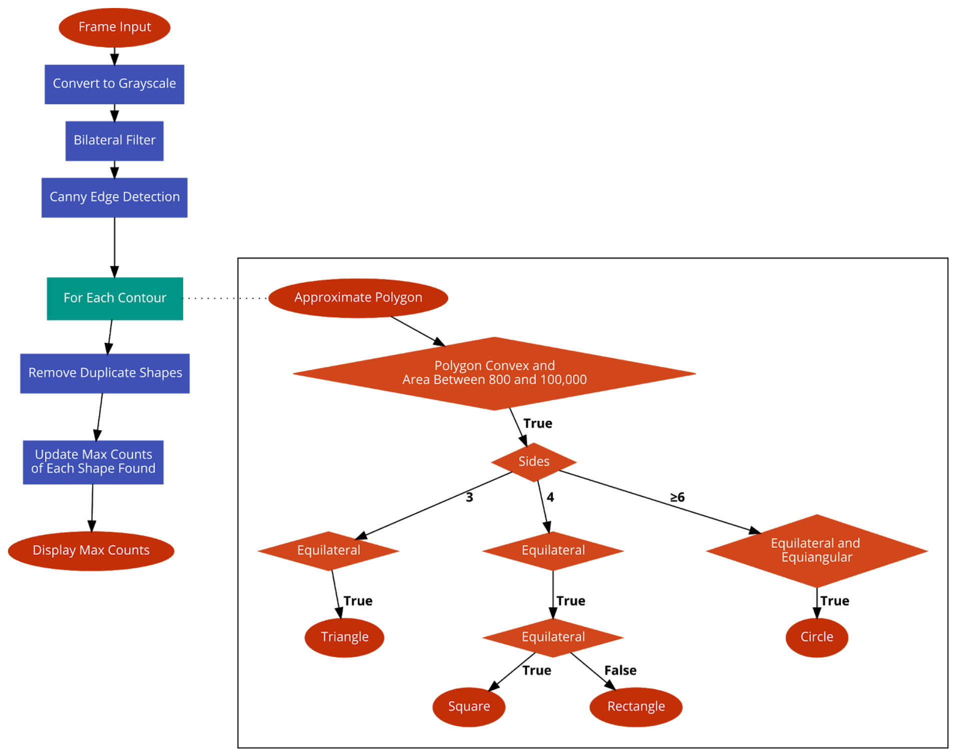
# Fox Enterprises

## Image Recognition Documentation

**Algorithm Description:**

Our benthic species recognizer uses video from one of our cameras via a capture card. A Java program running on a laptop on the surface processes the video. We use the OpenCV library for all image processing. With every frame of video the program converts it to grayscale and applied a bilateral filter. Then the program uses canny edge detection to create contours. The program iterates over all of the contours to create a list of all of the shapes it detects. For each contour it checks whether it is convex and has an area between 800 and 100,000 pixels. If the contour passes these requirements, the program then uses OpenCV's approxPolyDP function to calculate an approximately matching polygon. If the polygon has three sides and is equiangular, the contour is classified as a triangle. If it has four sides and is equiangular, then it is a quadrilateral. If the quadrilateral is equilateral, it's a square; otherwise, it's a rectangle. If the polygon has six or more sides and is equiangular, then it's a circle. Once the program has finished classifying all the polygons it removes duplicates that sometimes appear. If two shapes are too close to each other, one of them is ignored. After the final tally of shapes in that frame is found, the program will update the maximum tallies for each type of shape. This is to insure that all shapes are eventually found because our program airs on the side of less shapes. Finally, the shape counts are displayed.

**Data Flow Diagram:**



**Source Code:**

File: Display.java
```
import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.Point;
import org.opencv.core.Size;
import org.opencv.highgui.HighGui;
import org.opencv.imgcodecs.Imgcodecs;
import org.opencv.imgproc.Imgproc;
import org.opencv.videoio.VideoCapture;

import javax.swing.*;
import java.util.ArrayList;
import java.util.List;

/**
 * The {@code Display} class is the entry point of the program.
 * It manages shape recognition and displays the results.
 */
public class Display {

  public static void main(String[] args) {
    // Load OpenCV
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);

    // Load and resize benthic shapes display image
    Mat shapesImage = Imgcodecs.imread(
        "images/Benthic Shapes.png",
        Imgcodecs.IMREAD_UNCHANGED
    );
    Imgproc.resize(shapesImage, shapesImage,
        new Size(100, 480));
    // Split shapesImage into RGBA
    List<Mat> shapesChannels = new ArrayList<>();
    Core.split(shapesImage, shapesChannels);

    // Start video stream
    VideoCapture capture = new VideoCapture(1);
```

```java
Mat frame = new Mat();

int shapesCols = shapesImage.cols();
int shapesRows = shapesImage.rows();
int[] speciesMax = new int[4];
// Create display window
HighGui.namedWindow("display", HighGui.WINDOW_NORMAL);
HighGui.resizeWindow("display", 1600, 900);
// Main loop
while (capture.isOpened()) {
  // Read and resize frame
  capture.read(frame);
  Imgproc.resize(frame, frame, new Size(1600, 1160));

  // Count shapes on frame
  int[] species = ShapeIdentifier.identifyShapes(frame);
  int sum = 0;
  for (int i = 0; i < species.length; i++) {
    sum += species[i];
    Imgproc.putText(frame, Integer.toString(speciesMax[i]),
        new Point(10, 125 * i + 250),
        Imgproc.FONT_HERSHEY_COMPLEX, 5,
        ShapeIdentifier.RED);
  }

  // Update speciesMax
  if (sum >= 3) {
    for (int i = 0; i < 4; i++) {
      if (species[i] > speciesMax[i]) {
        speciesMax[i] = species[i];
      }
    }
  }

  // Draw shapes image
  for (int x = 0; x < shapesCols; x++) {
    for (int y = 0; y < shapesRows; y++) {
      if (shapesChannels.get(3).get(y, x)[0] > 0) {
        frame.put(y + 150, x + 130, 0, 0, 255);
```

```
          }
        }
      }
      // Display frame
      HighGui.imshow("display", frame);
      HighGui.waitKey(1);
    }
  }
}
```

File: Polygon.java
```java
import org.opencv.core.MatOfPoint2f;
import org.opencv.core.Point;

/**
 * The {@code Species} enum represents possible shapes of a
 * benthic species.
 */
enum Species {
    TRIANGLE,
    SQUARE,
    RECTANGLE,
    CIRCLE
}

/**
 * The {@code Polygon} class represents polygons identified by
 * the program and provides methods to calculate properties of
 * the polygon.
 */
class Polygon {
    private MatOfPoint2f mat;
    private Species species;

    /**
     * Creates a new {@code Polygon} with a given contour
     * and species.
     *
     * @param mat      the contour as a mat of points
     * @param species  the benthic species recognised
     */
    Polygon(MatOfPoint2f mat, Species species) {
        this.mat = mat;
        this.species = species;
    }

    /**
     * Gets a numerical representation of the species.
     *
 * @return benthic species ID
 */
int getID() {
    switch (species) {
        case TRIANGLE:
            return 0;
        case SQUARE:
            return 1;
        case RECTANGLE:
            return 2;
        case CIRCLE:
            return 3;
    }
    return -1;
}

/**
 * Gets the center of the polygon
 *
 * @return center point
 */
Point getCenter() {
    Point[] points = mat.toArray();
    Point center = new Point();
    for (Point point : points) {
        center.x += point.x;
        center.y += point.y;
    }
    center.x /= points.length;
    center.y /= points.length;

    return center;
}

/**
 * Determines whether the polygon reaches a certain
 * threshold of equilateral-ness.
 *
 * @param deviation the maximum deviation in side length
 * from the average
 * @return {@code true} if equilateral enough,
 * {@code false} otherwise
*/
boolean isEquilateral(double deviation) {
    Point[] points = mat.toArray();
    double[] sides = new double[points.length];

    for (int i = 0; i < points.length; i++) {
        Point a = points[i];
        Point b;
        if (i == points.length - 1) {
            b = points[0];
        } else {
            b = points[i + 1];
        }

        sides[i] = Math.hypot(a.x - b.x, a.y - b.y);
    }

    double aveSide = 0;
    for (double side : sides) {
        aveSide += side;
    }
    aveSide /= points.length;

    for (double side : sides) {
        if (Math.abs(side - aveSide) > deviation) {
            return false;
        }
    }

    return true;
}

/**
* Determines whether the polygon reaches a certain threshold
* of equiangular-ness.
* The polygon's angles must not deviate more than pi/8
```

```
 * radians from the average.
 *
 * @return {@code true} if equiangular enough,
 * {@code false} otherwise
 */
boolean isEquiangular() {
    Point[] points = mat.toArray();
    double[] angles = new double[points.length];
    for (int i = 0; i < points.length; i++) {
        Point a = points[i];
        Point b;
        if (i == points.length - 1) {
            b = points[0];
        } else {
            b = points[i + 1];
        }
        Point c;
        if (i > points.length - 3) {
            c = points[i - points.length + 2];
        } else {
            c = points[i + 2];
        }

        double ab = Math.hypot(a.x - b.x, a.y - b.y);
        double bc = Math.hypot(b.x - c.x, b.y - c.y);
        double ac = Math.hypot(a.x - c.x, a.y - c.y);
        angles[i] = Math.acos(
            (Math.pow(bc, 2) + Math.pow(ab, 2)
                - Math.pow(ac, 2))
            / (2 * bc * ab)
        );
    }

    double aveAngle = 0;
    for (double angle : angles) {
        aveAngle += angle;
    }
    aveAngle /= points.length;
```

```
        for (double angle : angles) {
            if (Math.abs(angle - aveAngle) > Math.PI / 8) {
                return false;
            }
        }

        return true;
    }

    boolean isParalellagram() {
        Point[] points = mat.toArray();
        double[] sides = new double[points.length];

        for (int i = 0; i < points.length; i++) {
            Point a = points[i];
            Point b;
            if (i == points.length - 1) {
                b = points[0];
            } else {
                b = points[i + 1];
            }

            sides[i] = Math.hypot(a.x - b.x, a.y - b.y);
        }

        return Math.abs(sides[0] - sides[2]) < 20 &&
                Math.abs(sides[1] - sides[3]) < 20;
    }
}
```