Platformy programistyczne .Net i Java

Dokumentacja aplikacji okienkowej z użyciem wielowątkowości Javie

Mateusz Marciak 272599

Opis Projektu

Celem projektu jest utworzenie aplikacji okienkowej w języku Java, która pozwala na obróbkę obrazków. Działanie aplikacji opiera się na wgraniu obrazka z rozszerzeniem .jpg, a następnie umożliwienie dokonania na nim operacji takich jak:

- Negatyw
- Progowanie
- Konturowanie
- Obrót i zmiana rozmiaru

Aplikacja została wykonana z użyciem frameworka **JavaFX**.

Implementacja

Na początku skupiono się na utworzeniu okienka, umieszczeniu w nim danych o autorze aplikacji oraz dodaniu możliwości załadowania obrazu z komputera.

```
public class ImageLoader {  1 usage

    public static Image loadJPG(Window parentWindow) {  1 usage
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Wybierz plik JPG");
        fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter( s: "Pliki JPG",  ...strings: "*.jpg")
        );

        File selectedFile = fileChooser.showOpenDialog(parentWindow);
        if (selectedFile != null && selectedFile.getName().toLowerCase().endsWith(".jpg")) {
            return new Image(selectedFile.toURI().toString());
        } else if (selectedFile != null) {
            UIUtils.showAlert(Alert.AlertType.ERROR,  message: "Niedozwolony format pliku");
        }
        return null;
    }
```

Rysunek 1 Załadowanie obrazu do aplikacji

Skorzystaliśmy z klasy **FileChooser**, a następnie ograniczyliśmy jej wybór do plików o rozszerzeniu *.jpg.

Następnie dodaliśmy funkcjonalność zapisywania pliku, w folderze windowsowym **Obrazy.**

```java
public class ImageSaver {  1 usage

    public static boolean saveToPicturesFolder(Image image, String filename) {  1 usage
        try {
            String userPictures = System.getProperty("user.home") + File.separator + "Pictures";
            File outputFile = new File(userPictures,  child: filename + ".jpg");

            if (outputFile.exists()) return false;

            BufferedImage bImage = SwingFXUtils.fromFXImage(image,  bufferedImage: null);
            BufferedImage rgbImage = new BufferedImage(bImage.getWidth(), bImage.getHeight(), BufferedImage.TYPE_INT_RGB);
            rgbImage.getGraphics().drawImage(bImage,  x: 0,  y: 0,  observer: null);

            ImageIO.write(rgbImage,  formatName: "jpg", outputFile);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

Rysunek 2 Zapis obrazu

Kolejnym etapem było dodanie wymienionych w opisie zadania funkcjonalności. Proces przetwarzania obrazu odbywał się przez zastosowaniu na nim wątków.

Użyta biblioteka do zrównoleglenia to **ExecutorService** – wysoko poziomowa biblioteka do zarządzania wątkami.

```java
public class ParallelProcessor {  4 usages

    private static final int THREAD_COUNT = 4;  5 usages
    private static final ExecutorService executor = Executors.newFixedThreadPool(THREAD_COUNT);  2 usages

    public static Image process(Image input, PixelOperation operation) throws InterruptedException, ExecutionException {
        int width = (int) input.getWidth();
        int height = (int) input.getHeight();
        WritableImage output = new WritableImage(width, height);
        PixelReader reader = input.getPixelReader();
        PixelWriter writer = output.getPixelWriter();

        int chunkHeight = height / THREAD_COUNT;
        Future<?>[] tasks = new Future[THREAD_COUNT];

        // zrównoleglenie
        for (int i = 0; i < THREAD_COUNT; i++) {
            final int startY = i * chunkHeight;
            final int endY = (i == THREAD_COUNT - 1) ? height : (i + 1) * chunkHeight;
            tasks[i] = executor.submit(() -> operation.apply(reader, writer, width, startY, endY));
        }

        // zakończenie działania
        for (Future<?> task : tasks) task.get();

        return output;
    }
```

Rysunek 3 Przetwarzanie zrównoleglone

Do przetwarzania użyto 4 wątki. Obraz dzielono na chunki (poziomo linie), każdy wątek wykonywał operacje na chunkach jednocześnie. Następnie czekano na zakończenie pracy każdego wątku i zwracano przetworzony obraz.

Tak zaimplementowaną operację na wątkach, stosowano zależnie od procesu, który chcieliśmy wykonać.

Poniżej przykład wykorzystania dla **negatywu.**

```
public class ImageProcessor {  7 usages

    public static Image generateNegative(Image inputImage) throws InterruptedException, ExecutionException {  1 usage
        return ParallelProcessor.process(inputImage, ( PixelReader reader,  PixelWriter writer,  int width,  int startY,  int endY) -> {
            for (int y = startY; y < endY; y++) {
                for (int x = 0; x < width; x++) {
                    Color color = reader.getColor(x, y);
                    Color negColor = new Color(
                            v: 1.0 - color.getRed(),
                            v1: 1.0 - color.getGreen(),
                            v2: 1.0 - color.getBlue(),
                            v3: 1.0
                    );
                    writer.setColor(x, y, negColor);
                }
            }
        });
    }
```

Rysunek 4 Przetwarzanie negatyw

Na koniec do aplikacji dodano **AppLogger**, który zbierał i wypisywał na bieżąco informacje z użycia aplikacji.

```
public class AppLogger {  14 usages
    private static final String LOG_FILE = "logi_aplikacji.txt";  1 usage
    private static final DateTimeFormatter FORMAT = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  1 u

    public static void log(String level, String message) {  3 usages
        try (PrintWriter writer = new PrintWriter(new FileWriter(LOG_FILE,  append: true))) {
            String time = LocalDateTime.now().format(FORMAT);
            writer.printf("[%s] [%s] %s%n", time, level, message);
        } catch (Exception e) {
            System.err.println("Błąd zapisu logu: " + e.getMessage());
        }
    }

    public static void info(String message) {  13 usages
        log( level: "INFO", message);
    }

    public static void warn(String message) {  no usages
        log( level: "WARN", message);
    }

    public static void error(String message) {  1 usage
        log( level: "ERROR", message);
    }
}
```

Rysunek 5 Logi aplikacji – Implementacja

```
[2025-06-05 17:25:43] [INFO] Uruchomiono aplikację
[2025-06-05 17:27:08] [INFO] Wczytano obraz do aplikacji
[2025-06-05 17:27:32] [INFO] Wykonywanie operacji: Negatyw
[2025-06-05 17:27:34] [INFO] Negatyw został wygenerowany pomyślnie!
[2025-06-05 17:27:43] [INFO] Zmieniono rozmiar na :400x400
[2025-06-05 17:27:47] [INFO] Przywrócono oryginalne wymiary obrazu.
[2025-06-05 17:27:50] [INFO] Użytkownik zresetował obraz do oryginału
```

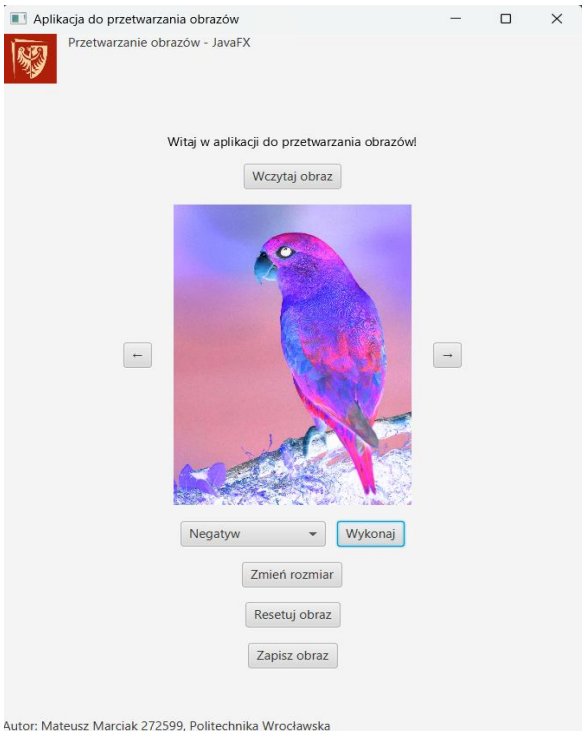
Rysunek 6 Logi aplikacji – przykład

Aplikacja została zabezpieczona, przed niepowołanymi działaniami wyświetlając komunikaty toast, logi i alerty.

```
public class UIUtils {  11 usages
    public static void showAlert(Alert.AlertType type, String message) {
        Alert alert = new Alert(type);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.show();
    }
}
```

Rysunek 7 Przykład implementacji alertów

Rezultat Końcowy



Rysunek 8 Wygląd aplikacji