Welcome to this exciting tutorial series on creating an image processor using Rust, with a particular focus on parallel computing. Whether you're an experienced programmer or a newbie, this series will provide you with the tools and knowledge needed to explore the world of image processing using Rust, one of the most powerful programming languages available today.

Prerequisites

Before we get started, there are a few prerequisites you should be aware of:

- A basic understanding of programming concepts, such as variables, loops, and functions.
- Familiarity with Rust and its syntax would be helpful, but not strictly necessary. This course will guide you through all the essential concepts.
- Rust should be installed on your computer. If not, visit the official Rust website for installation instructions.
- A code editor and a willingness to learn are also required.

Course Structure

This course is structured to guide you from the basics of setting up the image crate in Rust, to building the foundation of image manipulation, and leveraging parallel computing to speed up processes. The reason we focus on parallel computing is because images are 2D vectors of pixels, so having knowledge of arrays, vectors and working with them would be beneficial for this course.

We'll cover everything from loading and saving images, to applying complex transformations and optimizations. We'll start with an overview of image processing and its applications. Then, we will delve into the practical aspects like installing, configuring, loading, and saving images in different formats.

Basic Image Operations

Basic image operations such as resizing and rotating will be covered. We'll also discuss how to use rayon for parallel computing to enhance performance. Towards the end, we'll explore some advanced use cases and a comprehensive project. By the end of the series, you will have a solid understanding of how to handle image processing in Rust. You'll be able to implement efficient, safe, and parallelized image processing applications.

About Zenva

Before we start, let's talk a little about Zenva. Zenva is an online learning academy with over a million students. We offer a wide range of courses for beginners and for those who want to learn something new. Our courses are versatile, allowing you to learn in the way that suits you best. You can choose to view the video tutorials, read the written summaries, or even follow along with the included project files.

We're excited to guide you through this course. Let's get started.

Image processing is a fascinating field that involves modifying and enhancing digital images using computer algorithms. The primary goal of image processing is to improve the quality of images or to modify them for specific purposes. In certain contexts, such as machine learning, image processing is more about extracting usable information from images. This article will provide a beginner's guide to understanding image processing.

Understanding Image Processing

Think of an image as a giant mosaic composed of tiny colored blocks. Each of these blocks represents a pixel, which carries specific color and intensity information. Image processing involves altering these pixels using mathematical and programming techniques. This could involve adjusting, transforming, or completely reshaping these tiny blocks to enhance the image or extract important data from it. This could involve changing brightness, applying filters, or detecting shapes. In essence, image processing is about tweaking these pixels to perform visual magic.

Applications of Image Processing

Image processing is a powerful technology with applications across multiple industries. Here are a few examples:

- In healthcare, it helps doctors to diagnose diseases with greater accuracy through enhanced medical images.
- On the roads, it powers the eyes of autonomous vehicles, enabling them to see and navigate safely.
- In security, it empowers systems with facial recognition capabilities to identify individuals quickly and accurately.
- It also plays a crucial role in environmental monitoring, the arts and entertainment, showcasing its versatility and broad impact on our daily lives and global infrastructure.

Image Processing Course Outline

For those interested in learning more about image processing, here's a simple and straightforward course outline:

1. Setting up the project
2. Configuring dependencies
3. Loading images
4. Manipulating images
5. Applying parallel processing using Rayon to optimize our algorithm

Before starting this course, ensure you have Rust installed, or you can use online coding platforms like Replet. This course is designed to be beginner-friendly and will introduce you to the fascinating world of image processing.

So, are you ready to dive into the world of image processing? See you in the next session.

In this tutorial, we will explore the Image Crate in Rust. This crate is essential for anyone looking to work with images in Rust, providing extensive functionality for opening, saving, and manipulating images in a variety of formats.

Understanding Crates and Packages in Rust

In Rust, a Crate is the output artifact of the compiler. The compilation model centers on artifacts called crates. Each compilation processes a single crate in source form, and if successful, it produces a single crate in binary form, either an executable or some sort of library. Meanwhile, a package is an artifact managed by Cargo, which is Rust's package manager.

There's a vast array of packages available for integration into our applications, which can be found on the website crates.io. For instance, there are packages under the category of command line utilities that can aid us in working with the terminal, manipulating arguments, and much more.

Working with Cargo

Cargo is Rust's build system and package manager. It allows us to create new projects, configure our applications, and manage Rust applications. When creating a new project, Cargo generates an initializer configuration file known as cargo.toml. This file contains a package and dependencies list, where we can configure our project.

Installing and Using the Image Crate

In this tutorial, we'll be using the Image Crate in Rust. This crate provides native Rust implementations of image encoding and decoding, as well as some basic image manipulation functions.

To verify the installation of the Rust compiler and Cargo, you can use the following commands:

```
rustc --version
cargo --version
```

Once you've verified that Rust and Cargo are correctly installed, you can create a new project using the following command:

```
cargo new image_processor
```

After creating the project, you can add the Image Crate to your dependencies list. This can be done manually in the cargo.toml file, or you can use the terminal command:

```
cargo add image
```

With the Image Crate added to your dependencies list, you can now start building your application. It's a good idea to compile and run your application at this point to ensure everything is working correctly before proceeding.

Compiling and Running the Application

To compile your application, use the command:

```
cargo build
```

And to run your application, use the command:

```
cargo run
```

If your application returns a “Hello World” message, then everything is working correctly. In the next part of this tutorial, we will delve deeper into the Image Crate and its functionalities.

In this lesson, we'll enhance our knowledge of image processing in Rust by learning how to load and save images in various formats. Handling image formats is a critical aspect of any image processing application. Thankfully, Rust simplifies this task with the Image Crate. Let's dive in.

Importing the Package

Firstly, we need to import the package. We will use the open function from the Image Crate to load the image file into our application. The open function is versatile, automatically determining the image's format and handling it accordingly.

```
use image::open;
```

Understanding Image Formats

When processing images, it's important to understand the different image formats you might work with. Each format, such as JPEG, PNG, or BMP, has its own characteristics:

- **JPEG:** This format is highly compressed, making it perfect for web use but not ideal for quality-critical applications.
- **PNG:** PNG offers lossless compression, which is great for preserving detail and supports transparency.
- **BMP:** BMP files are not compressed, providing high quality at the cost of file size.

Loading an Image

To load an image file into our application, we'll use the open function from the Image Crate. We'll declare a variable to hold the image and call the open function, passing the path to the image file. If something goes wrong, we'll call expect and provide a message indicating a failure to load or open the image.

```
let img = image::open("path_to_image_file")
    .expect("Failure to load/open image");
```

Obtaining Image Information

Once the image is loaded into our application, we may need to get information such as the pixels or the width and height of the image. To do this, we need to include the generic image view trait, which allows us to get the dimensions, pixels, or perform bound checking for the image.

```
use image::{open, GenericImageView};

let (width, height) = img.dimensions();
println!("Width: {}, Height: {}", width, height);
```

Saving an Image

Saving an image is straightforward. We can use the save method to save it in the original format, or the save_with_format function if we want to convert it to another format. We'll try to save a sample image in both PNG and WEBP formats.

First, we need to import the ImageFormat enumeration, which provides supported image formats like png, bmp, icon, jpeg, and more. We also need the DynamicImage enumeration, which represents a matrix of pixels convertible from and to an RGBA representation.

```
use image::{open, GenericImageView, ImageFormat, DynamicImage};
```

Next, we'll use the save_with_format method to save the image in the desired format, passing the full path and the image format as arguments. If something goes wrong, we'll use expect to display an error message.

```
img.save_with_format("path_to_image_file.png", ImageFormat::Png)
    .expect("Failed to save image as PNG");
img.save_with_format("path_to_image_file.webp", ImageFormat::WebP)
    .expect("Failed to save image as webp");
```

With these steps, we've successfully loaded an image, obtained its information, and saved it in different formats using Rust and the Image Crate.

In this lesson, we will learn how to perform basic but crucial image operations in Rust. Specifically, we will focus on resizing and rotating images using ImageCrate. These operations form the backbone of many image processing applications, from preparing images for web use to adjusting them for analysis.

Creating Functions for Resizing and Saving Images

First, we will create functions for each specific task to ensure we write clean code. We will create placeholder functions for resizing the image and another function for saving the image.

```
fn resize_image() {

}

fn save_image() {

}
```

In the `resize_image` function, we will open the image and resize it to the desired dimensions. In the `save_image` function, we will save the image to the specified output path in PNG format.

Importing Necessary Libraries

To use these functions, we need to import some classes and libraries. We will use the Image Crate library, specifically: `open` for opening the image and `DynamicImage` for working with the image. We also need `imageops` and `FilterType` for resizing the image.

```
use image::{open, DynamicImage, imageops::FilterType};
```

Resizing the Image

With our resizing function, we first want to set up parameters to collect our image path, width, and height. We can then add our functionality for resizing:

```
fn resize_image(path: &str, width: u32, height: u32) -> DynamicImage {
    let img = open(path).expect("Failed to open image");
    img.resize(width, height, FilterType::Nearest)
}
```

In the above code, we simply open the image and resize it. When resizing, we create a new file out of the pixels, and we need to provide a filtering type to enhance the image. The `FilterType` comes into play here. We use the `Nearest` filter type because it’s the simplest and fastest. However, it may not always yield the best quality.

Saving the Image

After resizing the image, we call the `save_image` function to save the file. If something goes wrong, we show a message to the user.

```
fn save_image(img: &DynamicImage, output_path: &str) {
    img.save_with_format(output_path, image::ImageFormat::Png).expect("Failed to save image");
}
```

Using the Functions

We can now use these functions in the main function. We load an image, resize it, and save it with a new name.

```
fn main() {
    println!("Image Processing");
    let resized_image = resize_image(
         "D:\\image_processor\\assets\\sample_img.jpg",
         512,
         512
     );

    save_image(
         &resized_image,
         "D:\\image_processor\\assets\\sample_img_RESIZED.png"
     );
}
```

In the above code, we provide the path to the file and the new dimensions to the `resize_image` function. Then we save the resized image to the specified output path.

Improving Image Quality

When we run the code and open the resized image, we may notice that the image quality is not very good. To improve the quality, we can use a different filter type. The `Lanczos3` filter can enhance the image quality significantly.

```
img.resize(width, height, FilterType::Lanczos3)
```

After changing the filter type and running the code again, we should notice that the resized image has much better quality than before. However, we may still encounter issues with the image dimensions. In the next lessons, we will learn how to fix these issues.

In this lesson, we will learn how to rotate images using the Rust programming language. Rotating images is a fundamental operation in image processing. We will rotate an image by 90, 180, and 270 degrees using methods provided by the image crate in Rust. These methods are very efficient and maintain the image's quality during the rotation without any filtering process required. Let's get started.

Creating a Function to Rotate an Image

First, we need to create a function that will rotate an image. This function will take the path of the image file (a string) and the degree of rotation (a U32 value) as parameters, and return a dynamic image.

```
fn rotate_image(path: &str, degrees: u32) -> DynamicImage {
    let img = open(path).expect("Failed to load image");
    match degrees {
        90 => img.rotate90(),
        180 => img.rotate180(),
        270 => img.rotate270(),
        _ => {
            eprintln!("Unsupported rotation angle. Please give 90, 180, 270.");
            img
        }
    }
}
```

In the above code:

- We open the image file using the provided path.
- We use a match statement to check the provided degree of rotation.
- If the degree is 90, we rotate the image by 90 degrees using the rotate90() method.
- If the degree is 180, we rotate the image by 180 degrees using the rotate180() method.
- If the degree is 270, we rotate the image by 270 degrees using the rotate270() method.
- If the degree is anything else, we print an error message and return the image as it is.

Using the Rotate Image Function

Now that we have our rotate_image function, we can use it to rotate an image in our main function. In the below example, we're rotating an image by 90 degrees.

```
let rotated_img = rotate_image("D:\\image_processor\\assets\\sample_img.jpg", 90);
```

Saving the Rotated Image

After rotating the image, we need to save it. We can do this using the save_image function. In this example, we're saving the rotated image to a new file.

```
save_image(&rotated_img, "D:\\image_processor\\assets\\sample_img_ROTATED.png");
```

And that's it! We now have a function that can rotate an image by 90, 180, or 270 degrees. This function is very efficient and maintains the image's quality during the rotation process. In the next lesson, we will cover how to handle the aspect ratio and additional functions that come with the image crate in Rust. We will also learn how to modify images using our own functions and optimize them using the Rayon Parallel Computing Library for faster processing.

In this lesson, we will explore image ratios and other functionalities provided by the image crate in Rust. We will create two additional functions that you need to know. The first one is resizing the image by keeping the ratios in mind and the second one is image cropping using the image crate.

Understanding Image Ratios

The aspect ratio of an image is the ratio of its width to its height. It’s crucial to maintain the aspect ratio when resizing images to prevent distortion. For instance, stretching an image beyond its original aspect ratio can make it look unnaturally wide or tall.

Resizing Image While Maintaining Aspect Ratio

We will modify our previously created resizeImage function to incorporate the concept of aspect ratio. The function will now do some dimension calculations while maintaining the aspect ratio.

```
fn resize_image_maintain_ratio(path: &str, new_width: Option<u32>, new_height: Option
<u32>) -> DynamicImage {
    let img = open(path).expect("Failed to open image");
    let (width, height) = img.dimensions();

    // calculate new dimensions while maintaining the aspect ratio
    let ratio = width as f32 / height as f32;
    let (resize_width, resize_height) = match(new_width, new_height) {
        (Some(w), None) => (w, (w as f32 / ratio).round() as u32),
        (None, Some(h)) => ((h as f32 * ratio).round() as u32, h),
        (Some(w), Some(h)) => (w, h),    // if both dimensions are specified use them
 as is
        (None, None) => (width, height), // if no dimensions are specified use the or
iginal dimensions
    };

    img.resize(resize_width, resize_height, FilterType::Lanczos3)
}
```

In this function, we have the option U32 for the new widths and new height parameters. The option type in Rust encapsulates an optional value. A value can be something or nothing. That’s why we are able to use these some and none options. It indicates that it expects some kind of value or nothing.

First, we get the image dimensions. We then calculate the aspect ratio of the image by dividing the width by height in the floating data type. Next, we calculate the four options that we have:

- If a new width is provided, but height is not, the new height is calculated using the aspect ratio to maintain the proportions.
- If a new height is given, but the width is not, the new width is calculated as new height multiplied by the ratio.
- If both dimensions are specified, use them as is. We don’t need to calculate anything to maintain the aspect ratio.
- If no dimensions are specified, use the original dimensions.

Finally, we return the resized image.

Using the Function in Main

Let’s try to use this function in the main function. We are specifying that the width is 800 pixels and we’re not specifying the height, indicating that the function should calculate the height automatically to preserve the image’s original aspect ratio. This setup allows for flexible image resizing, making it easy to resize images by width, height, both or neither, in which case the original size is maintained while optionally preserving the aspect ratio to avoid visual distortion.

```
fn main() {
    println!("Image Processing");

    let resized_image = resize_image_maintain_ratio(
        "D:\image_processor\assets\sample_img.jpg",
        Some(800),
        None
    );

    save_image(
        &resized_image,
        "D:\image_processor\assets\sample_img_RE_DI.png"
    );
}
```

That’s all for this lesson. In the next lesson, we’ll explore more image processing functionalities in Rust.

In this lesson, we will learn how to crop an image using the Rust programming language. We will create a function that takes an image and the coordinates X and Y where the crop should start, along with the width and height of the crop area. We will then use this function to crop an image and save it.

Creating the Image Cropping Function

First, let’s create the function that will perform the image cropping. We will call this function `crop_image`. This function will take as parameters the image we want to crop and the X and Y coordinates where the cropping should start. It will also take the width and height of the area to be cropped. The function will return the cropped image as a DynamicImage object.

```
fn crop_image(img: &DynamicImage, x: u32, y: u32, width: u32, height: u32) -> Dynamic
Image {
    let cropped_img = crop_imm(img, x, y, width, height);
    DynamicImage::ImageRgba8(cropped_img.to_image())
}
```

In the function body, we create a new variable `cropped_img` and use the `crop_imm` function from the `imageops` module of the `image` crate to perform the cropping. We then return the cropped image as a DynamicImage object.

Using the Image Cropping Function

Now let’s use our `crop_image` function. First, we open an image file and store it in a variable. Then, we call our `crop_image` function and pass it the image and the dimensions for the cropping starting at X and Y coordinates and the required width and height of the cropped area.

```
let img = image::open("D:\\image_processor\\assets\\sample_img.jpg").expect("Failed t
o open image");
let cropped_img = crop_image(&img, 50, 50, 500, 500);
```

Finally, we save the cropped image using the `save` function from the `image` crate.

```
cropped_img.save("D:\\image_processor\\assets\\cropped_image.png").expect("Failed to
save cropped image");
```

Building and Running the Project

After writing the code, we can build and run our project using the `cargo build` and `cargo run` commands respectively. If everything is correct, our program will crop the image as specified and save it.

Remember that you can play with this function as much as you want with different parameters. You can also try to use other functions from the `image` crate to learn about all the possibilities and capabilities of the `image` package in Rust.

Welcome back! In this session, we’ll explore how to use the Rayon Parallel Processing Library for creating paths as a scalable solution for image processing. Parallel computing can significantly speed up tasks, especially when working with large images or performing complex manipulations. By ‘large images’, we mean hundreds of variously sized images.

Preparing the File

We’ll be focusing specifically on creating our custom function this lesson. As such, we’re going to start things off with a premade template. This template will import the necessary libraries for us, have a function to load an image, and also load that image in our main function.

```
use image::{DynamicImage, ImageBuffer, Rgba};

fn load_image(filepath: &str) -> Result<DynamicImage, image::ImageError> {
    image::open(filepath)
}

fn rotate_image_90_clockwise() {

}

fn main() {
    println!("Image Processing - Enumerating Pixels");

    let img_path = "assets/sample_img.png";
    let img = load_image(img_path).expect("Failed to load image");

}
```

Setting Up the Rayon Crate

First, make sure you add the Rayon crate along with the image crate in your config file. To do that, you can open your terminal and write cargo add rayon. The Rayon library will enable us to easily implement parallel processing in our project using threads. This setup is crucial for optimizing our image processing tasks in the coming function implementation process.

Creating a Function for Image Rotation

We have seen image rotation in the previous sessions using the rotate90 and rotate180 functions to rotate the images. This is highly abstracted away that we can’t see what’s going on really. I mean, in the background, but this time we will create our own rotate_image_90_clockwise function by using loops and iterations. And then we will optimize it by using Rayon.

So, let’s start writing our code now.

```
fn rotate_image_90_clockwise(img: &ImageBuffer<Rgba<u8>, Vec<u8>>,) -> ImageBuffer<Rg
ba<u8>, Vec<u8>> {
    let (width, height) = img.dimensions();

    let mut new_img = ImageBuffer::new(height, width); // new image dimensions are sw
apped

    img.enumerate_pixels().for_each(|(x, y, pixel)| {
        let new_x = height - y - 1; // new x is (height-y-1)
        let new_y = x; // new y is the old x

        new_img.put_pixel(new_x, new_y, *pixel);
    });

    new_img
}
```

This function takes in an image buffer as a parameter and returns an image buffer. The dimensions of the new image buffer are swapped from the original image buffer. Then, it enumerates over each pixel in the original image. For each pixel, it calculates the new x and y positions based on the formulae provided. The new pixel is then placed in the new image buffer at the calculated position. Finally, the new image buffer is returned.

In the main function, we call the rotate_image_90_clockwise function to rotate the image 90 degrees clockwise. The rotated image is then saved to a new file.

```
fn main() {
    println!("Image Processing - Enumerating Pixels");

    let img_path = "assets/sample_img.png";
    let img = load_image(img_path).expect("Failed to load image");

    let rotated_img = rotate_image_90_clockwise(&img.to_rgba8()); // rotate 90 degree
s clockwise

    rotated_img
        .save("assets/processed_img.png")
        .expect("Failed to save processed image.");
}
```

While the current implementation uses a straightforward approach with enumerate_pixels, this is not parallelized. This can be adapted for parallel execution using Rayon for better performance on large images. By using Rayon, we’ll be able to use multiple CPU cores to process the image. We will explore this in the next session.

In this lesson, we will learn how to process images in parallel using the rayon thread pool in Rust. The advantage of parallel processing is that if you have multiple images to process, you can use the power of multiple CPU cores to speed up the process. For example, if you have 10 images in a folder that you need to rotate, instead of processing them one by one which could take a significant amount of time, we can use the rayon library to process all of them in parallel, making our application faster and more scalable.

Adding the Required Packages

The first step is to add the necessary packages to our Rust file. We will use the dynamic image, image buffer, rgba, rayon prelude, fs (file system), and std path libraries. The rayon prelude library allows us to use all its functions while the fs and std path libraries help us to work with the file system and paths respectively.

```
use image::{DynamicImage, ImageBuffer, Rgba};
use rayon::prelude::*;
use std::fs;
use std::path::Path;
```

Loading Images from a Directory

Our goal is to process all images in a specific directory. For this, we need to read the directory, filter out everything that isn’t an image file, and collect the paths of these image files into a vector. Below is the implementation:

```
let image_dir = Path::new("parallel");
let image_paths = fs::read_dir(image_dir)
    .expect("Failed to read directory")
    .into_iter()
    .filter_map(Result::ok)
    .filter(|entry| entry.file_type().is_ok() && entry.file_type().unwrap().is_file()
)
    .map(|entry| entry.path())
    .collect::<Vec<_>>();
```

Processing Images in Parallel

Now that we have the paths of all the images, we can process them in parallel using the rayon thread pool. We will use the par_iter function from the rayon library to create a parallel iterator over the image paths. For each image path, we load the image, rotate it, and save it to a new location. The rotate_image_90_clockwise function is a custom function that rotates an image 90 degrees clockwise.

```
image_paths.par_iter().for_each(|img_path| {
    let img = load_image(img_path.to_str().unwrap()).expect("Failed to load image");
    let rotated_img = rotate_image_90_clockwise(&img.to_rgba8());

    let processed_path = format!("parallel/processed_{}", img_path.file_name().unwrap
().to_str().unwrap());
    rotated_img
```

```
        .save(processed_path)
        .expect("Failed to save processed file.");
});
```

And that’s it! You have created a simple image processing application in Rust that uses parallel processing to rotate multiple images. This is a powerful technique that can greatly speed up your application when dealing with a large number of images.

In this article, we will be exploring the basics of image loading and saving, along with the functions from image, image ops, and rayon crates in the Rust programming language. Our ultimate challenge will be to apply a Gaussian blur to an input image. This task will not only consolidate your understanding of the basics but will also provide you with a practical and usable application.

Task Overview

Our goal is to develop a program in Rust that blurs an input image using a Gaussian blur. The Gaussian blur is a type of image-blurring filter that uses a Gaussian function. Here's a basic outline of how our program should work:

1. Load the input image.
2. Apply the Gaussian blur function to the image.
3. Save the blurred image.

Guidelines

To begin with, create an image using the 'image' crate. This crate provides functions for image processing. You will specifically need to use the 'blur' function from this crate for our task.

If you need further information on how to blur an image in Rust, you can always refer to the official documentation at docs.rs. Here, you will find the 'image' crate and can explore its functions in detail: https://docs.rs/image/latest/image/imageops/fn.blur.html

In this lesson, we're going to discuss how to implement image blurring in Rust. We'll walk you through the solution for the image blurring task, explaining each step along the way. We'll cover how to import the necessary libraries, open the image file, use the blur function, and save the blurred image. We have also provided a smaller and compressed image file, blur.jpg, to make the processing faster.

Imports

Firstly, we need to import the necessary libraries. In Rust, we do this using the `use` keyword. Here are the imports for our image blurring task:

```
use image::{self, RgbImage};
use image::imageops;
```

Image Blurring Function

Next, we create a blur function that takes in an image and a kernel size for blurring it. We use the blur function from the `imageops` library, passing in the image and kernel size as arguments.

```
fn blur(image: &RgbImage, kernel_size: f32) -> RgbImage {
    let blurred = imageops::blur(image, kernel_size);
    blurred
}
```

Using the Blur Function

In the main function, we open the image file and pass it to the blur function. We also specify the kernel size for the blur.

```
fn main() {
    let img = image::open("assets/blur.jpg").unwrap().to_rgb8();
    let blurred = blur(&img, 3.0);
    blurred.save("assets/blurred.jpg").unwrap();
}
```

Running the Script

Before running the script, you need to move the blur.jpg file to the source folder. Then, you can open the terminal, build the project using `cargo build`, and run it using `cargo run`. After running the script, you should find a new file named Blur.jpg in the assets folder. This file is the result of the image blurring process.

Feel free to play around with this script by providing different parameters to the blur function. This will help you understand how the function works and how changing the kernel size affects the amount of blur applied to the image.

Welcome to the conclusion of our journey through the Rust language and its capabilities in image processing. We embarked on a fascinating exploration of this powerful tool and its potential for creating and manipulating images. This course was designed to provide you with the foundations for understanding the language and using it to tackle a wide range of image processing tasks confidently.

Key Learnings

- We began with an introduction to Rust language and its image crate. This provided us with a comprehensive understanding of how to manipulate images including basic transformations like rotations and resizing to more intricate aspects like maintaining aspect ratios and optimizing performance.
- Our journey then took us to the rayon parallel processing library. We harnessed its potential to unlock the true power of concurrency in our image manipulation tasks. This allowed us to distribute the workload across multiple cores, leading to significant performance improvements.
- Recognizing the importance of scalability and efficiency in modern computing, we also explored how to parallelize our image processing algorithms using the Rayon library. This transformed our sequential code into a parallel one, further enhancing its performance.

Reflections and Next Steps

As we conclude our course, it’s essential to reflect on the knowledge gained and the skills acquired. With a solid foundation in Rust and the image crate, you now possess the tools to tackle a wide range of image processing tasks with confidence. This course was one of your first steps in this journey and we hope that you will continue to explore and learn more with Zenva Academy.

Zenva Academy is an online learning platform with over a million learners. We offer a wide range of courses for beginners and those wanting to learn something new. We recommend checking out other interesting courses available on Zenva Academy to further enhance your skills and knowledge.

Once again, thank you for joining us on this journey. We hope that you found this course informative and enjoyable. My name is Muminjon and it was my pleasure to guide you through this course. Happy learning!

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

Main.rs

Found in Project/Challenge

The code applies a Gaussian blur to an image. It accomplishes this by first loading a specified image ("blur.jpg"), applying the Gaussian blur with a specified kernel size (3.0), and then saving the result to a new image file ("blurred.jpg").

```
use image::{self, RgbImage};
use image::imageops;

// Apply Gaussian blur to an image and save the result
fn blur(image: &RgbImage, kernel_size: f32) -> RgbImage {
    // Apply Gaussian blur with specified kernel size
    let blurred = imageops::blur(image, kernel_size);

    blurred
}

fn main() {
    // Load an image from a file
    let img = image::open("assets/blur.jpg").unwrap().to_rgb8();

    // Apply Gaussian blur with kernel size 3.0
    let blurred = blur(&img, 3.0);

    // Save the result
    blurred.save("assets/blurred.jpg").unwrap();
}
```

0_main.rs

Found in Project/src

The code opens JPEG image from a specific directory, then saves the image in two different formats – PNG and WEBP, into the same directory. There are commented lines of code which, if uncommented, would print the dimensions of the image.

```
use image::{open, GenericImageView, ImageFormat, DynamicImage};

fn main() {

    let img = open("D:\\image_processor\\assets\\sample_img.jpg").expect("Failure to
load/open image");
```

```
    // let (width, height) = img.dimensions();

    // println!("Width: {}, Height: {}", width, height);

    // saving with specific format png
    img.save_with_format("D:\\image_processor\\assets\\sample_img.png", ImageFormat::
Png).expect("Failed to save image as PNG");

    // saving with specific format webp
    img.save_with_format("D:\\image_processor\\assets\\sample_img.webp", ImageFormat:
:WebP).expect("Failed to save image as webp");

}
```

1_main.rs

Found in Project/src

This code is for an image processing utility. The utility offers functions to resize and rotate images, and to save them in PNG format. The main function runs code to load an image, rotate it 90°, and save it with a new name.

```
use image::{open, DynamicImage, imageops::FilterType::Lanczos3, GenericImageView};

fn resize_image(path: &str, width: u32, height: u32) -> DynamicImage {
    let img = open(path).expect("Failed to open image");
    img.resize(width, height, FilterType::Lanczos3)
}

fn save_image(img: &DynamicImage, output_path: &str) {
    img.save_with_format(output_path, image::ImageFormat::Png).expect("Failed to save
 image");
}

fn rotate_image(path: &str, degrees: u32) -> DynamicImage {
    let img = open(path).expect("Failed to load image");
    match degrees {
        90 => img.rotate90(),
        180 => img.rotate180(),
        270 => img.rotate270(),
        _ => {
            eprintln!("Unsupported rotation angle. Please give 90, 180, 270.");
            img
        }
    }
}

fn main() {
    println!("Image Processing");

    let rotated_img = rotate_image(
        "D:\\image_processor\\assets\\sample_img.jpg",
        90
    );

    save_image(
        &rotated_img,
        "D:\\image_processor\\assets\\sample_img_ROTATED.png"
    );

    // let resized_image = resize_image(
    //     "D:\\image_processor\\assets\\sample_img.jpg",
    //     512,
    //     512
    // );

    // save_image(
    //     &resized_image,
    //     "D:\\image_processor\\assets\\sample_img_RESIZED.png"
    // );


}
```

2_main.rs

Found in Project/src

This code is a simple image processing application in Rust that performs image manipulation tasks such as opening, cropping, resizing while maintaining the aspect ratio and saving an image. The main function demonstrates opening an image file, cropping the image, and saving the cropped image. The commented block in the main function shows resizing the image maintaining its aspect ratio and then saving the resized image.

```
use image::{open, DynamicImage, imageops::FilterType, ImageFormat, imageops::crop_imm
};
use image::GenericImageView; // Required for the dimensions() method

fn save_image(img: &DynamicImage, output_path: &str) {
    // Specify the format directly in the save method
    img.save_with_format(output_path, ImageFormat::Png).expect("Failed to save image"
);
}

fn resize_image_maintain_ratio(path: &str, new_width: Option<u32>, new_height: Option
<u32>) -> DynamicImage {
    let img = open(path).expect("Failed to open image");
    let (width, height) = img.dimensions();

    // calculate new dimensions while maintaining the aspect ratio
    let ratio = width as f32 / height as f32;
    let (resize_width, resize_height) = match(new_width, new_height) {
        (Some(w), None) => (w, (w as f32 / ratio).round() as u32),
        (None, Some(h)) => ((h as f32 * ratio).round() as u32, h),
        (Some(w), Some(h)) => (w, h),    // if both dimensions are specified use them
 as is
```

```
        (None, None) => (width, height), // if no dimensions are specified use the or
iginal dimensions
    };

    img.resize(resize_width, resize_height, FilterType::Lanczos3)
}

fn crop_image(img: &DynamicImage, x: u32, y: u32, width: u32, height: u32) -> Dynamic
Image {
    let cropped_img = crop_imm(img, x, y, width, height);
    DynamicImage::ImageRgba8(cropped_img.to_image())
}

fn main() {
    println!("Image Processing");

    let img = open("D:\\image_processor\\assets\\sample_img.jpg").expect("Failed to o
pen image");
    let cropped_img = crop_image(&img, 100, 2000, 2500, 2500);

    save_image(
        &cropped_img,
        "D:\\image_processor\\assets\\cropped_image.png"
    );

    // let resized_image = resize_image_maintain_ratio(
    //     "D:\\image_processor\\assets\\sample_img.jpg",
    //     Some(800),
    //     None
    // );

    // save_image(
    //     &resized_image,
    //     "D:\\image_processor\\assets\\sample_img_RE_DI.png"
    // );
}
```

3_main.rs

Found in Project/src

This code is an image processing program in Rust. It loads an image from a file, rotates it 90 degrees clockwise, and then saves the rotated image back to a file. The rotation function works by swapping the image dimensions and then repositioning each pixel.

```
use image::{DynamicImage, ImageBuffer, Rgba};

fn load_image(filepath: &str) -> Result<DynamicImage, image::ImageError> {
    image::open(filepath)
}

fn rotate_image_90_clockwise(img: &ImageBuffer<Rgba<u8>, Vec<u8>>,) -> ImageBuffer<Rg
ba<u8>, Vec<u8>> {
    let (width, height) = img.dimensions();

    let mut new_img = ImageBuffer::new(height, width); // new image dimensions are sw
apped

    img.enumerate_pixels().for_each(|(x, y, pixel)| {
        let new_x = height - y - 1; // new x is (height-y-1)
        let new_y = x; // new y is the old x

        new_img.put_pixel(new_x, new_y, *pixel);
    });

    new_img
}


fn main() {
    println!("Image Processing - Enumerating Pixels");

    let img_path = "assets/sample_img.png";
    let img = load_image(img_path).expect("Failed to load image");

    let rotated_img = rotate_image_90_clockwise(&img.to_rgba8()); // rotate 90 degree
s clockwise

    rotated_img
        .save("assets/processed_img.png")
        .expect("Failed to save processed image.");

}
```

Final_main.rs

Found in Project/src

The code executes an image processing task in parallel using the Rayon library. It loads images from a directory, rotates each image 90 degrees clockwise, and saves the processed images to a new location. The code uses image processing functions from the image library and the rotate_image_90_clockwise function to rotate the images.

```
use image::{DynamicImage, ImageBuffer, Rgba};
use rayon::prelude::*;
use std::fs;
use std::path::Path;


fn load_image(filepath: &str) -> Result<DynamicImage, image::ImageError> {
    image::open(filepath)
}

fn rotate_image_90_clockwise(img: &ImageBuffer<Rgba<u8>, Vec<u8>>,) -> ImageBuffer<Rg
ba<u8>, Vec<u8>> {
    let (width, height) = img.dimensions();
    let mut new_img = ImageBuffer::new(height, width); // new image dimensions are sw
apped

    img.enumerate_pixels().for_each(|(x, y, pixel)| {
        let new_x = height - y - 1; // new x is (height-y-1)
        let new_y = x; // new y is the old x

        new_img.put_pixel(new_x, new_y, *pixel);
    });

    new_img
}


fn main() {
    println!("Image Processing - Parallel Processing with Threat Pool - rayon");

    // get all images paths in the parallel directory
    let image_dir = Path::new("parallel");
    let image_paths = fs::read_dir(image_dir)
        .expect("Failed to read directory")
        .into_iter()
        .filter_map(Result::ok)
        .filter(|entry| entry.file_type().is_ok() && entry.file_type().unwrap().is_fi
le())
        .map(|entry| entry.path())
        .collect::<Vec<_>>();

    // process images in parallel using rayon thread pool
    image_paths.par_iter().for_each(|img_path| {
        let img = load_image(img_path.to_str().unwrap()).expect("Failed to load image
");
        let rotated_img = rotate_image_90_clockwise(&img.to_rgba8());

        let processed_path = format!("parallel/processed_{}", img_path.file_name().un
wrap().to_str().unwrap());
        rotated_img
            .save(processed_path)
            .expect("Failed to save processed file.");
    });

}
```