Computer Graphics with Rust

Borworntat Dendumrongkul
and
Teejuta Sriwaranon

Computer Engineering, Chulalongkorn University
October 30, 2025

Rust

C++/OpenGL Pain

C++ and OpenGL contains lots of pains.

- Segmentation fault (core dumped)
- Iterator invalidation
- Data race conditions
- OpenGL's "GLOBAL STATE MACHINE"
- `glBindBuffer(...)`???
- `glEnable(...)`???

C++ is powerful but **unsafe**. OpenGL is familiar but **outdated** and **stateful**.

Philosophy of Rust

Safety First: Rust aims to eliminate entire classes of errors common in other languages, like memory leaks, dangling pointers, and data races.

Zero-Cost Abstraction: Rust strives to provide powerful abstractions without sacrificing performance.

Performance: Rust offering speeds comparable to languages like **C and C++**

Concurrency: Rust's compiler prevent data races by itself.

Modern Tooling: A single command is used to build, run, test and manage dependencies.

Meet Cargo!

Cargo is Rust's build system and package manager. It simplifies the process of managing Rust projects.

- `cargo new <project-name>`: Create a new Rust project.
- `cargo build`: Compile the current project.
- `cargo run`: Build and run the current project.
- `cargo test`: Run tests for the current project.
- `cargo doc --open`: Generate and open documentation for the current project.

Rust IDE/Compiler

We highly recommend using **Rust Rover** by JetBrains or **Visual Studio Code** with the Rust extension for a better development experience.

Your First Rust Program

```
fn main() {
  println!("Hello, World!");
}
```

You can run this program by saving it in a file named `main.rs` and executing `cargo run` in the terminal.

Concept of Ownership

Rust's ownership system is a set of rules that govern how memory is managed in Rust. The three main concepts of ownership are:

- Each value in Rust has a variable that is called its **owner**.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Borrowing and Mutability

In Rust, you can borrow a value using references. There are two types of references: immutable and mutable.

- Immutable references allow you to read the value without modifying it. You can have multiple immutable references to the same value at the same time.
- Mutable references allow you to modify the value. However, you can only have one mutable reference to a value at a time, and you cannot have any immutable references while a mutable reference exists.

Data Types in Rust

Rust has several built-in data types, including:

Data Type	Description
`i32`	32-bit signed integer
`u32`	32-bit unsigned integer
`f32`	32-bit floating-point number
`bool`	Boolean value (true or false)
`char`	Unicode character
`String`	Growable string type

Variables in Rust

```rust
// Variable declaration
let x: i32 = 5; // Immutable variable
let mut y: i32 = 10; // Mutable variable
```

By default, variables in Rust are immutable. To make a variable mutable, you need to use the `mut` keyword.

Keyword	Description
`let`	Declare a variable
`mut`	Make a variable mutable
`const`	Declare a constant value
`static`	Declare a static variable

Control Flow in Rust

```rust
// If-else statement
if x < y {
  println!("x is less than y");
} else {
  println!("x is greater than or equal to y");
}
// Switch statement
match x {
  1 => println!("One"),
  2 => println!("Two"),
  _ => println!("Other"),
}
```

Loops in Rust

```rust
// While loop
let mut count = 0;
while count < 5 {
  println!("Count: {}", count);
  count += 1;
}
// For loop
for i in 0..5 {
  println!("i: {}", i);
}
```

Functions in Rust

Function in Rust are defined using the `fn` keyword. There are two ways to return a value from a function: implicit return and explicit return.

```
// Implicit return
fn add(a: i32, b: i32) -> i32 {
  a + b
}
// Explicit return
fn add_return(a: i32, b: i32) -> i32 {
  return a + b;
}
```

Structs in Rust

Structs are custom data types that allow you to group related data together. You can define a struct using the `struct` keyword.

```
struct Point {
  x: f32,
  y: f32,
}
```

You can create an instance of a struct and access its fields using dot notation.

```
let p = Point { x: 1.0, y: 2.0 };
println!("Point: ({}, {})", p.x, p.y);
```

Struct Methods in Rust

You can define methods for a struct using the `impl` keyword. Methods are functions that are associated with a specific struct.

```
impl Point {
  fn distance_from_origin(&self) -> f32 {
    (self.x.powi(2) + self.y.powi(2)).sqrt()
  }
}
```

You can call a method on an instance of a struct using dot notation.

```
let p = Point { x: 3.0, y: 4.0 };
println!("Distance from origin: {}", p.distance_from_origin());
```

Debug vs Release Builds in Rust

In Rust, there are two main build profiles: **debug** and **release**. The default `cargo run` uses the debug build, while `cargo run --release` uses the release build. **Debug**

Build:

- Includes debug symbols for easier debugging.
- No optimizations applied, so code runs slower.
- Includes runtime checks (e.g., bounds checking) for safety.
- Faster compilation time.

Release Build:

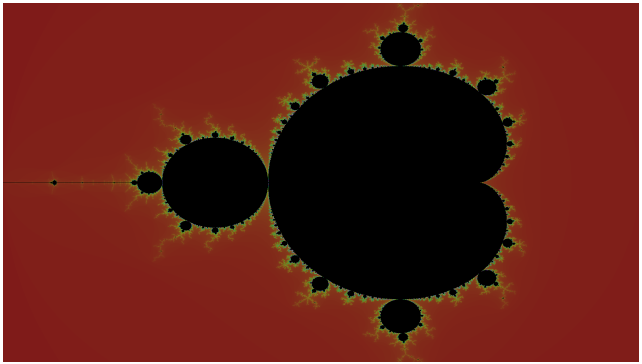
- Optimizations enabled (e.g., inlining, loop unrolling) for maximum performance.
- Debug symbols and checks are removed to improve speed.
- Slower compilation but much faster execution.

For performance-critical applications like rendering, always use `--release` for the best speed!

Lab 8.1: Mandelbrot Set Rendering (Single Thread)

Lab 8.1: Mandelbrot Set Rendering using Rust

From the skeleton code provided, implement a program that renders the Mandelbrot set in single threaded.



Mandelbrot Set Definition

The Mandelbrot set is defined by iterating the function:

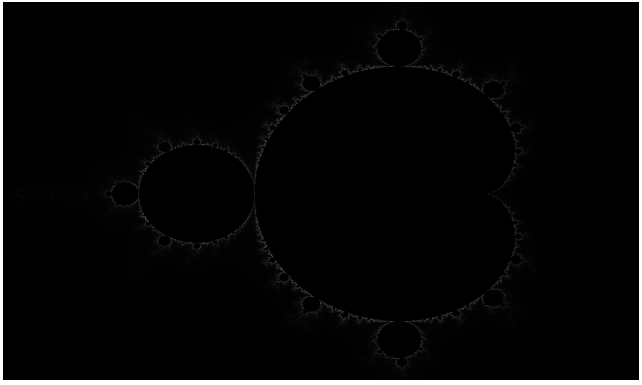
$$Z_{n+1} = Z_n^2 + C$$

where $Z_0 = 0$ and C is a complex number corresponding to a point in the complex plane. A point C is part of the Mandelbrot set if the sequence does not diverge to infinity.

Mandelbrot Set Visualization

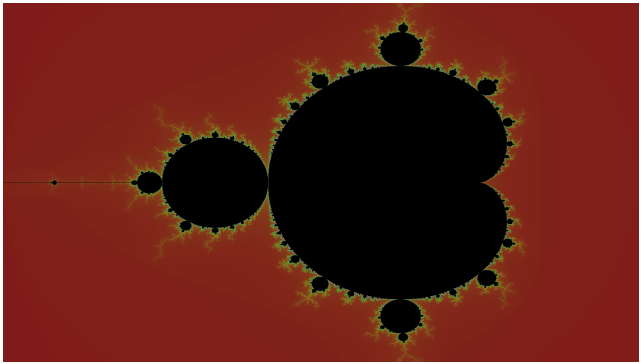
To visualize the Mandelbrot set, we map each pixel in the image to a point in the complex plane. We then iterate the function for each point and determine how quickly the sequence diverges. The number of iterations before divergence is used to assign a color to the pixel.

The resulting image is just a monochrome representation of the Mandelbrot set.



Mandelbrot Set Coloring

To enhance the visualization, we can apply coloring based on the number of iterations before divergence. A common approach is to use a color gradient or palette to map iteration counts to colors. This can create visually appealing images that highlight the intricate details of the Mandelbrot set.



Using External Crates

To work with images and complex numbers in Rust, we can use external crates. In this lab, we will use the `image` crate for image manipulation and the `num-complex` crate for complex number operations. To add these dependencies, include the following in your `Cargo.toml` file

```
[dependencies]
image = "0.24.9"
num-complex = "0.4.2"
```

Or just use https://github.com/2110479-Computer-Graphics/Lab08-Rust-Parallel provided!

Parallelism in Rust

Parallelism in Rust

We can achieve parallelism in Rust using several libraries and techniques. One of the most popular libraries for parallelism in Rust is `rayon`, which provides a simple and efficient way to perform data parallelism.

`rayon`: A data parallelism library that allows you to easily parallelize computations over collections.

In our lab, we will be using the `rayon` crate to parallelize the Mandelbrot set rendering process.

Using Rayon for Parallelism

To use `rayon`, you first need to add it as a dependency in your `Cargo.toml` file:

```
[dependencies]
rayon = "1.10.0"
```

from: Reddit r/ProgrammerHumor

Parallel Iteration with Rayon

With `rayon`, you can easily parallelize iterations over collections using the `par_iter()` method. This method allows you to perform operations on each element of a collection in parallel, utilizing multiple threads to improve performance.

```
use rayon::prelude::*;
let numbers = vec![1, 2, 3, 4, 5];
let sum: i32 = numbers.par_iter().map(|&x| x * 2).sum();
```

Why we need Parallelism?

Sometimes single-threaded performance is not enough, especially for computationally intensive tasks like rendering the raytracing or fractals. By leveraging multiple CPU cores, we can significantly reduce the time it takes to complete these tasks.

from: Reddit r/pcmasterrace

Lab 8.2: Mandelbrot Set Rendering (Multi-thread)

Lab 8.2: Mandelbrot Set Rendering using Rust in Multi-threading

From 8.1, extend your program to support multi-threading using `rayon` crate. Also, compare the performance between single-threaded and multi-threaded implementations. (Question set in MCV)

Graphics API

Graphics API Overview

A Graphics API (Application Programming Interface) is a set of functions and protocols that allow developers to create and manipulate graphics in applications. Graphics APIs provide a way to interact with the underlying hardware, such as GPUs (Graphics Processing Units), to render images, animations, and visual effects. Some popular graphics APIs include:

- OpenGL (Khronos Group)
- Vulkan (Khronos Group)
- DirectX (Microsoft)
- Metal (Apple)
- WebGPU (W3C)

Choosing a Graphics API

When choosing a graphics API for your project, consider the following factors:

- **Platform support**: Ensure the API supports the target platforms (Windows, macOS, Linux, Web, etc.).
- **Performance**: Evaluate the performance characteristics of the API for your specific use case.
- **Ease of use**: Consider the learning curve and available documentation and resources.
- **Community and ecosystem**: Look for an active community and a rich ecosystem of libraries and tools.

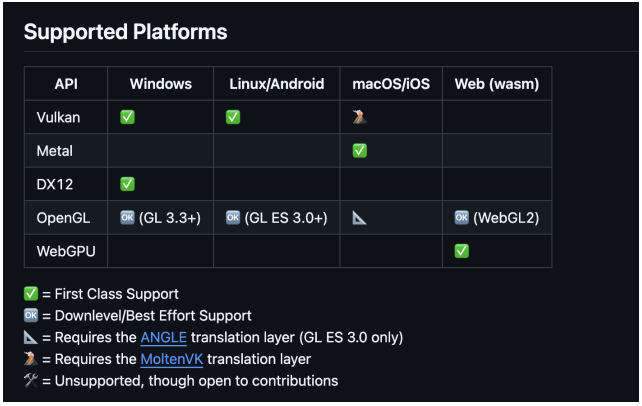
GPU Programming with wgpu

Introduction to wgpu

`wgpu` is a modern, safe, and efficient graphics API for Rust that provides a high-level abstraction over low-level graphics APIs like Vulkan, Metal, and DirectX 12. It is designed to be easy to use while still providing powerful features for GPU programming. Some key features of `wgpu` include:

- Cross-platform support
- Safety and performance
- Modern graphics features
- Integration with Rust's ecosystem

Why wgpu?



Supported Platforms

API	Windows	Linux/Android	macOS/iOS	Web (wasm)
Vulkan	✅	✅	🚀	
Metal			✅	
DX12	✅			
OpenGL	🆗 (GL 3.3+)	🆗 (GL ES 3.0+)	📐	🆗 (WebGL2)
WebGPU				✅

✅ = First Class Support
🆗 = Downlevel/Best Effort Support
📐 = Requires the ANGLE translation layer (GL ES 3.0 only)
🚀 = Requires the MoltenVK translation layer
🛠 = Unsupported, though open to contributions

from: `https://github.com/gfx-rs/wgpu`

Adding wgpu to Your Project

To use `wgpu` in your Rust project, you need to add it as a dependency in your `Cargo.toml` file:

```
[dependencies]
wgpu = "0.17"
```

Basic Structure of a wgpu Application

A basic `wgpu` application typically consists of the following initialization steps:

1. **Instance**: Create a `wgpu::Instance` to represent the WebGPU API instance.
2. **Adapter**: Request a `wgpu::Adapter` from the instance, which represents a physical GPU or graphics device.
3. **Device and Queue**: Request a `wgpu::Device` and `wgpu::Queue` from the adapter. The device manages GPU resources, and the queue handles command submission.
4. **Swap Chain**: Create a swap chain for presenting rendered images to the window.
5. Create shaders, buffers, and other resources.
6. Implement the rendering loop to draw frames.

Example: Hello Triangle with wgpu

Here is a simple example of rendering a triangle using `wgpu`:

- Initialize `wgpu` and create a window.
- Set up the GPU device and swap chain.
- Create vertex and index buffers for the triangle.
- Write shaders to render the triangle.
- Implement the rendering loop to draw the triangle.

Shader Programming in wgpu

In `wgpu`, shaders are typically written in WGSL (WebGPU Shading Language). WGSL is a modern shading language designed specifically for use with WebGPU and `wgpu`. Compared to GLSL or HLSL, WGSL has a simpler syntax and is designed to be more secure and easier to use.

WGSL vs GLSL (Vertex Shader)

WGSL Vertex Shader:

```
// WGSL
@vertex
fn main(@location(0) position: vec4<f32>) -> @builtin(position) vec4<f32> {
  return position;
}
```

GLSL Vertex Shader:

```
// GLSL
#version 450
layout(location = 0) in vec4 position;
void main() {
  gl_Position = position;
}
```

WGSL vs GLSL (Fragment Shader)

WGSL Fragment Shader:

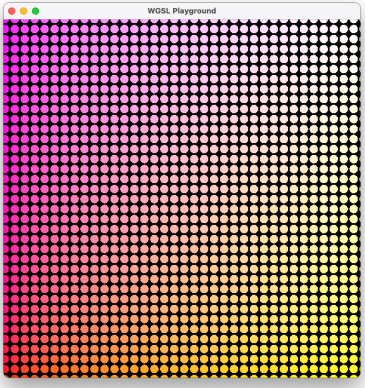
```
// WGSL
@fragment
fn main() -> @location(0) vec4<f32>{
  return vec4<f32>(1.0, 0.0, 0.0, 1.0); // Red color
}
```

GLSL Fragment Shader:

```
// GLSL
#version 450
layout(location = 0) out vec4 color;
void main() {
  color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

WGSL Playground

To experiment with WGSL shaders, you can use the WGSL Playground, an online tool that allows you to write and test WGSL shaders in real-time.



from: `https://github.com/paulgb/wgsl-playground`

wgpu Pipelines: Graphics vs Compute

`wgpu` supports two main types of pipelines:

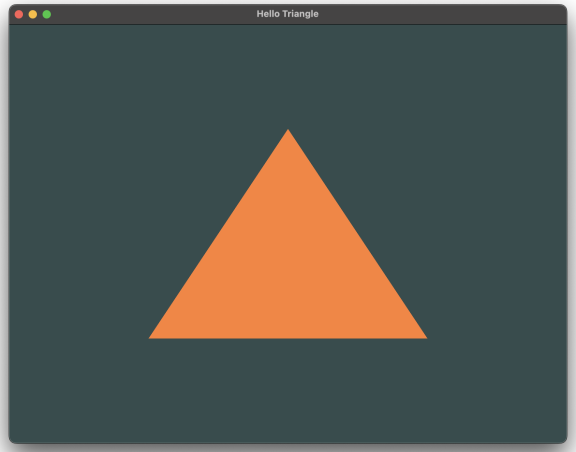
- **Graphics/Render Pipeline**: Used for traditional rendering tasks like drawing triangles, applying shaders, and outputting to the screen. This is what you'll use in Lab 8.3 for rendering shapes.
- **Compute Pipeline**: Used for general-purpose GPU computing tasks like mathematical calculations, simulations, or image processing. This is what you'll use in Lab 8.4 for computing the Mandelbrot set.

The graphics pipeline processes vertices through stages (vertex shader $\rightarrow$ fragment shader), while the compute pipeline runs parallel compute shaders on GPU threads.

Lab 8.3: Hello N-Gon using wgpu

Lab 8.3: Hello N-Gon using wgpu

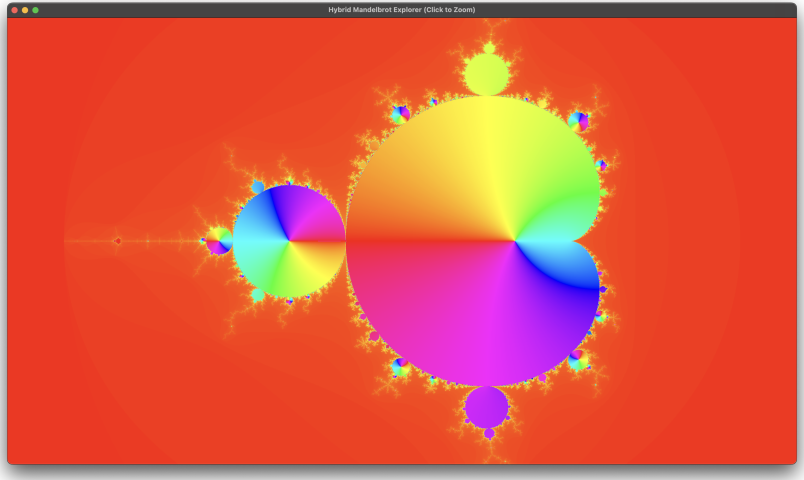
Try to render your Lab 1 (N-gon rendering) using `wgpu` crate.



Lab 8.4: Mandelbrot Set using wgpu

Lab 8.4: Mandelbrot Set using wgpu

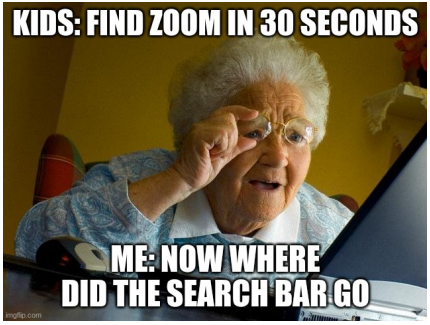
We will try to render Mandelbrot again!, using `wgpu` crate.



Lab 8.4: Mandelbrot Set using wgpu

Requirements:

- Render Mandelbrot set using GPU (wgpu crate)
- (Optional) Implement zoom-in functionality using left mouse click



from: Imgflip

cargo run –release

Bibliography I

- Clean Code Studio (n.d.). *The Philosophy of Rust*. Accessed: 2025-10-24. URL: https://www.cleancode.studio/rust/the-philosophy-of-rust (visited on 10/24/2025).
- Geier, Zachary D. (n.d.). *An absolute beginners guide to WGPU*. Accessed: 2025-10-24. URL: https://zdgeier.com/wgpuintro.html (visited on 10/24/2025).
- Klabnik, Steve, Carol Nichols, and Chris Krycho (n.d.). *The Rust Programming Language*. Accessed: 2025-10-24. URL: https://doc.rust-lang.org/book/ (visited on 10/24/2025).