Contents

1 Rust ペイントアプリ - 詳細コード解説

1.1 概要

本プロジェクトは、Rust 言語と egui/eframe フレームワークを使用したペイント（お絵描き）アプリケーションです。ユーザーが自由に線を描き、色や太さを変更でき、消しゴム機能も備えた本格的なペイントツールとなっています。

1.2 プロジェクト構造

```
rust_painter/
├── Cargo.toml          # プロジェクト設定と依存関係
├── src/
│   ├── main.rs         # エントリーポイントとモジュール宣言
│   ├── app.rs          # PaintApp メイン実装
│   ├── stroke.rs       # Stroke 構造体と描画処理
│   ├── tools.rs        # ツール設定管理
│   ├── font.rs         # フォント初期化
│   └── ui/
│       ├── mod.rs      # UI モジュール統合
│       ├── toolbar.rs  # ツールバー UI
│       ├── sidebar.rs  # サイドパネル UI
```

```
│       └──── canvas.rs    # キャンバス描画処理
├──── fonts/
│    └──── NotoSansJP-Regular.ttf  # 日本語フォント
└──── target/              # ビルド生成物（Rust コンパイラが作成）
```

1.3 依存関係の詳細解説

1.3.1 1. eframe (v0.31.1)

- **役割**: eframe は eGUI 上に構築されたアプリケーションフレームワーク
- **なぜ使用するか**: クロスプラットフォーム対応の GUI アプリケーションを簡単に作成できる
- **具体的な使用箇所**:
  - `eframe::run_native()` - ネイティブウィンドウを作成し、アプリケーションを実行
  - `eframe::NativeOptions` - ウィンドウの初期設定（サイズなど）
  - `eframe::App` trait - アプリケーションのメインループを定義

1.3.2 2. egui (v0.31.1)

- **役割**: Immediate Mode GUI ライブラリ
- **なぜ使用するか**:
  - 状態管理が簡単（従来の GUI ライブラリと異なり、UI の状態を毎フレーム再構築）
  - 高いパフォーマンスと軽量性
  - Rust らしい安全性とパフォーマンス
- **Immediate Mode の利点**:
  - UI の状態とロジックが一箇所に集約される
  - デバッグが容易
  - 複雑な状態管理が不要

1.3.3 3. image (v0.25.6)

- **役割**: 画像処理用ライブラリ
- **現在の使用状況**: まだ実装されていないが、将来的に画像の保存/読み込み機能で使用予定

1.3.4 4. rfd (v0.15.3)

- **役割**: Native File Dialog（ファイル選択ダイアログ）
- **現在の使用状況**: まだ実装されていないが、ファイル保存/読み込みダイアログで使用予定

1.4 リファクタリング後のモジュール構造

本プロジェクトは当初 267 行の単一ファイル（main.rs）でしたが、保守性と拡張性を向上させるため、以下の 8 つのモジュールに分割されました。

1.4.1 モジュール分割の利点

- **責務の分離**: 各モジュールが明確な役割を持つ
- **保守性向上**: バグ修正や機能追加が局所化される
- **テスト容易性**: 各モジュールを独立してテスト可能
- **再利用性**: 他のプロジェクトでの部分的な再利用が可能
- **協業効率**: 複数人での開発時の競合を削減

1.5 各モジュールの詳細解説

1.5.1 1. main.rs - エントリーポイント (21 行)

```
mod app;
mod stroke;
mod tools;
mod font;
mod ui;

use eframe::egui;
use app::PaintApp;

fn main() -> Result<(), eframe::Error> {
    let options = eframe::NativeOptions {
        viewport: egui::ViewportBuilder::default().with_inner_size([800.0, 600.0]),
        ..Default::default()
    };

    eframe::run_native(
        "ペイントアプリ",
        options,
        Box::new(|cc| Ok(Box::new(PaintApp::new(cc)))),
    )
}
```

リファクタリング後の変更点: - **モジュール宣言**: 各機能モジュールを宣言 - **import 最小化**: 必要最小限の import のみ - **main 関数**: 基本的な構造は変更なし

詳細解説: - **モジュールシステム**: Rust の mod キーワードで外部モジュールを宣言 - **戻り値の型**: Result<(), eframe::Error>は、エラーハンドリングのための Rust の標準的なパターン- **NativeOptions 設定**: - viewport: ウィンドウの表示設定を行う - with_inner_size([800.0, 600.0]): 初期ウィンドウサイズを 800x600 ピクセルに設定 - ..Default::default(): その他の設定はデフォルト値を使用（Rust の構造体更新記法）- **run_native 関数**: - 第 1 引数: ウィンドウタイトル - 第 2 引数: ウィンドウ設定 - 第 3 引数: アプリケーションインスタンスを作成するクロージャ - Box::new(|cc| ...): ヒープ上にクロージャを配置 - cc: &eframe::CreationContext: アプリケーション作成時のコンテキスト情報

1.5.2 2. app.rs - メインアプリケーションロジック

```
```rust
use eframe::egui;
use crate::tools::ToolSettings;
use crate::ui::canvas::CanvasHandler;
use crate::ui::{render_toolbar, render_sidebar};
use crate::font::setup_fonts;

pub struct PaintApp {
    tools: ToolSettings,
```
```

```
    canvas: CanvasHandler,
}

impl PaintApp {
    pub fn new(cc: &eframe::CreationContext<'_>) -> Self {
        setup_fonts(&cc.egui_ctx);
        Self::default()
    }

    fn clear_canvas(&mut self) {
        self.canvas.clear();
    }
}

impl eframe::App for PaintApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        // UI レンダリングロジック
    }
}
```

リファクタリング後の変更点:
- **構造の簡素化**: `ToolSettings`と`CanvasHandler`に責務を分離
- **フォント設定の分離**: `font.rs`モジュールに移動
- **UI処理の分離**: `ui`モジュール群に移動

設計思想:
- **コンポジション**: 複雑な機能を小さな部品に分解
- **単一責任原則**: 各構造体が明確な責任を持つ
- **依存関係の明確化**: `use`文で依存関係が一目瞭然

3. stroke.rs - 描画ストローク管理

```rust
```rust
use eframe::egui::{self, Color32, Pos2};

#[derive(Clone)]
pub struct Stroke {
    pub points: Vec<Pos2>,
    pub color: Color32,
    pub width: f32,
}

impl Stroke {
    pub fn new(color: Color32, width: f32) -> Self { /* ... */ }
    pub fn add_point(&mut self, point: Pos2) { /* ... */ }
    pub fn draw(&self, painter: &egui::Painter, offset: egui::Vec2) { /* ... */ }
```
```

```
    pub fn draw_smooth_stroke(
        painter: &egui::Painter,
        p1: Pos2,
        p2: Pos2,
        width: f32,
        color: Color32
    ) { /* 補間描画ロジック */ }
}
```

リファクタリング後の変更点:
- **メソッドの追加**: 構造体に関連する処理を集約
- **カプセル化**: データ操作のための専用メソッド
- **描画ロジックの内包**: `draw`メソッドで自己描画が可能

設計の利点:
- **オブジェクト指向的**: データと処理が一体化
- **再利用性**: 他の場所からも簡単に使用可能
- **保守性**: Stroke 関連の修正が局所化される

4. tools.rs - ツール設定管理

```rust
```rust
use eframe::egui::Color32;

#[derive(Debug, Clone, PartialEq)]
pub enum Tool {
    Pen,
    Eraser,
}

pub struct ToolSettings {
    pub current_tool: Tool,
    pub brush_size: f32,
    pub brush_color: Color32,
    pub background_color: Color32,
}

impl ToolSettings {
    pub fn get_current_color(&self) -> Color32 {
        match self.current_tool {
            Tool::Pen => self.brush_color,
            Tool::Eraser => self.background_color,
        }
    }
    // その他のヘルパーメソッド
```
```

```
}
```

リファクタリング後の変更点:
- **ツール概念の導入**: `enum Tool`でツールタイプを明確化
- **設定の集約**: ツール関連の設定を一箇所に集約
- **型安全性の向上**: `bool`から `enum`への変更

設計の利点:
- **拡張性**: 新しいツールの追加が容易
- **型安全性**: コンパイル時にツールタイプをチェック
- **一貫性**: ツール切り替えロジックの統一

5. font.rs - フォント初期化

```rust
```rust
use eframe::egui;
use std::sync::Arc;

pub fn setup_fonts(ctx: &egui::Context) {
    let mut fonts = egui::FontDefinitions::default();
    fonts.font_data.insert(
        "noto_sans_jp".to_owned(),
        Arc::new(egui::FontData::from_static(
            include_bytes!("../fonts/NotoSansJP-Regular.ttf")
        )),
    );

    fonts.families
        .entry(egui::FontFamily::Proportional)
        .or_default()
        .insert(0, "noto_sans_jp".to_owned());

    ctx.set_fonts(fonts);
}
```

リファクタリング後の変更点:
- **関数として分離**: 再利用可能な独立関数
- **責務の明確化**: フォント設定のみに特化
- **テスト可能性**: 単体テストが容易

詳細解説:
- **`include_bytes!`マクロ**:
  - コンパイル時にファイルをバイナリデータとして埋め込み
  - 実行時にファイルアクセスが不要（パフォーマンス向上）
  - 配布時にフォントファイルの同梱が不要
- **`Arc<T>`** (Atomically Reference Counted):
```

  - マルチスレッド環境で安全な参照カウント
  - eGUI の内部でフォントデータを複数箇所で共有するため
- **フォント優先度設定**:
  - `insert(0, ...)`: 最高優先度で日本語フォントを設定
  - 日本語文字が適切にレンダリングされる

6. ui/canvas.rs - キャンバス描画処理

```rust
```rust
use eframe::egui::{self, Sense};
use crate::stroke::Stroke;
use crate::tools::ToolSettings;

pub struct CanvasHandler {
    pub strokes: Vec<Stroke>,
    pub current_stroke: Option<Stroke>,
}

impl CanvasHandler {
    pub fn render(&mut self, ui: &mut egui::Ui, tools: &ToolSettings) {
        // キャンバス描画エリアの設定
        let (response, painter) = ui.allocate_painter(
            ui.available_size(),
            Sense::drag(),
        );

        // マウス入力処理
        // 描画レンダリング
        // ステータス表示
    }

    pub fn clear(&mut self) {
        self.strokes.clear();
        self.current_stroke = None;
    }
}
```

詳細アルゴリズム解説:

問題の背景
- マウスイベントは離散的（飛び飛びの点）
- 高速なマウス移動では点間の距離が大きくなる
- 単純に点を結ぶだけでは線が途切れ途切れになる

解決アプローチ
1. **距離計算**: `(p2 - p1).length()`
```

   - ベクトルの長さ（ユークリッド距離）を計算
   - 2 点間の実際の距離を取得

2. **ステップサイズ決定**: `(width / 4.0).max(1.0)`
   - ブラシ幅の 1/4 を基準とした補間間隔
   - `max(1.0)`で最小値を保証（過度に細かい補間を防止）
   - ブラシが太いほど補間間隔も大きくなる（計算効率とのバランス）

3. **線形補間**: `p1 + t * (p2 - p1)`
   - `t`は 0.0 から 1.0 の補間パラメータ
   - 数学的にはベジエ曲線の 1 次形式
   - 2 点間を等間隔で補間

描画方式の選択理由
- **円形ブラシ**: `circle_filled()`を使用
- **なぜ円形か**:
  - 自然な描画感を実現
  - 方向に依存しない一様な太さ
  - アンチエイリアシング効果

7. ui/toolbar.rs & ui/sidebar.rs - UI コンポーネント

toolbar.rs:
```rust
use eframe::egui;
use crate::tools::{Tool, ToolSettings};

pub fn render_toolbar(ui: &mut egui::Ui, tools: &mut ToolSettings) -> bool {
    let response = ui.horizontal(|ui| {
        ui.label("ブラシサイズ:");
        ui.add(egui::Slider::new(&mut tools.brush_size, 1.0..=20.0));

        ui.separator();

        ui.label("色:");
        ui.color_edit_button_srgba(&mut tools.brush_color);

        ui.separator();

        ui.selectable_value(&mut tools.current_tool, Tool::Pen, "ペン");
        ui.selectable_value(&mut tools.current_tool, Tool::Eraser, "消しゴム");

        ui.separator();

        ui.button("クリア").clicked()
    });

    response.inner


```

}

sidebar.rs:

```rust
use eframe::egui;
use crate::tools::ToolSettings;

pub fn render_sidebar(ui: &mut egui::Ui, tools: &mut ToolSettings) {
    ui.heading("レイヤー");
    ui.separator();
    ui.label("レイヤー 1");

    ui.separator();
    ui.heading("背景色");
    ui.color_edit_button_srgba(&mut tools.background_color);
}
```

リファクタリング後の変更点:
- **UI 関数の分離**: 各 UI コンポーネントを独立した関数に
- **戻り値の活用**: ボタンクリック状態を返値で通知
- **借用エラーの解決**: クロージャによる借用競合を回避

設計の利点:
- **再利用性**: 他の場所からも同じ UI を使用可能
- **テスト可能性**: UI 部分の単体テストが容易
- **保守性**: UI 変更の影響範囲が限定的

8. マウス入力処理とレンダリング（統合後）

```rust
if response.dragged() {
    if let Some(pos) = response.interact_pointer_pos() {
        let canvas_pos = (pos - response.rect.min).to_pos2();

        if self.current_stroke.is_none() {
            self.current_stroke = Some(Stroke {
                points: vec![canvas_pos],
                color: if self.is_eraser { self.background_color } else { self.brush_color },
                width: self.brush_size,
            });
        } else if let Some(ref mut stroke) = self.current_stroke {
            stroke.points.push(canvas_pos);
        }
    }
}
```

詳細解説:

1.5.2.1 座標変換の理由

- `pos - response.rect.min`:
  - `pos`: 画面絶対座標
  - `response.rect.min`: キャンバス領域の左上角
  - 減算により、キャンバス相対座標に変換
  - なぜ必要か: UI の他の部分（ツールバー等）の影響を除去

1.5.2.2 描画状態管理

- **新規ストローク開始**:
  - `current_stroke.is_none()`: 現在描画中でない場合
  - 新しい `Stroke` インスタンスを作成
  - 初期点を設定

1.5.2.3 消しゴム実装の工夫

- **色による実装**: `self.background_color` を使用
- **利点**:
  - シンプルな実装
  - 既存の描画ロジックをそのまま使用可能
- **制限**:
  - 背景色変更時に過去の消しゴム部分は追従しない
  - より高度な実装では、レイヤーシステムや実際の削除が必要

統合された描画レンダリング処理:

```
for stroke in &self.strokes {
    if stroke.points.len() == 1 {
        painter.circle_filled(stroke.points[0] + offset, stroke.width / 2.0, stroke.color);
    } else if stroke.points.len() > 1 {
        painter.circle_filled(stroke.points[0] + offset, stroke.width / 2.0, stroke.color);
        for i in 1..stroke.points.len() {
            Self::draw_smooth_stroke(
                &painter,
                stroke.points[i - 1] + offset,
                stroke.points[i] + offset,
                stroke.width,
                stroke.color,
            );
        }
    }
}
```

詳細解説:

1.5.2.4 パフォーマンス最適化

- **描画の分岐処理**:
  - 単一点: 1 回の円描画で済む
  - 複数点: 補間描画により滑らかな線を実現
- **不要な処理の回避**:
  - `points.len() == 0` の場合は何も描画しない

- 条件分岐により無駄な処理を削減

1.5.2.5 メモリ効率

- **借用の活用**: `&self.strokes`
  - イテレーション中にデータをコピーしない
  - 大量のストロークがある場合のメモリ使用量を削減

1.6 リファクタリングの設計思想

1.6.1 1. モジュラーアーキテクチャの採用

- **単一責任原則**: 各モジュールが明確な責任を持つ
- **疎結合**: モジュール間の依存関係を最小限に
- **高凝集**: 関連する機能を同じモジュールに集約

1.6.2 2. Rust の言語特性の活用

- **所有権システム**: メモリ安全性の保証
- **型安全性**: コンパイル時のエラー検出
- **ゼロコスト抽象化**: 高レベル API でありながら高性能
- **モジュールシステム**: プライベート/パブリックの適切な使い分け

1.6.3 3. 拡張性とメンテナンス性の向上

- **責務分離**: 各モジュールが独立して進化可能
- **テスト容易性**: 各モジュールの単体テストが可能
- **並行開発**: チームでの開発時の競合を削減
- **部分的再利用**: 他プロジェクトでの部分的な流用が可能

1.6.4 4. Immediate Mode GUI との相性

- **状態管理の簡素化**: UI 状態とアプリケーション状態が同一フレーム内で処理
- **デバッグの容易さ**: UI の動作を逐次追跡可能
- **パフォーマンス**: 必要な部分のみ再描画

1.7 モジュール分割による開発効率の向上

1.7.1 1. 開発段階での利点

- **並行開発**: 複数人で異なるモジュールを同時開発可能
- **影響範囲の限定**: バグ修正や機能追加が局所化
- **コードレビュー**: 変更箇所が明確で レビューが容易
- **テスト戦略**: モジュール単位でのテストが可能

1.7.2 2. 保守段階での利点

- **デバッグ効率**: 問題の原因特定が容易
- **リファクタリング**: 段階的な改善が可能
- **機能追加**: 既存コードへの影響を最小限に
- **ドキュメント**: モジュール単位での文書化

1.7.3 3. 将来の改善ポイント

1.7.3.1 機能面の拡張

- **レイヤーシステム**: `ui/layers.rs` として独立モジュール化
- **アンドゥ/リドゥ**: `commands.rs` でコマンドパターン実装
- **ファイル入出力**: `io.rs` で保存/読み込み機能
- **プラグインシステム**: 動的な機能拡張

1.7.3.2 アーキテクチャの発展

- **イベントドリブン**: メッセージパッシング方式の導入
- **状態管理**: より高度な状態管理パターン
- **設定システム**: 外部設定ファイルの対応
- **多言語対応**: 国際化（i18n）モジュール

1.7.4 4. 学習リソースとしての価値

このリファクタリングされたペイントアプリケーションは以下の学習に最適です：

- **Rust モジュールシステム**: 実践的なモジュール分割例
- **eGUI/eframe フレームワーク**: 各 UI コンポーネントの使用法
- **ソフトウェア設計**: SOLID 原則の実践例
- **リファクタリング技法**: 段階的な改善手法
- **協業開発**: チーム開発での構造化手法

モジュール分割により、267 行の単一ファイルから、保守しやすく拡張可能な 8 つのモジュール構造に進化し、実際の開発現場で求められるコード品質を実現しています。