Contents

1 eGUI/eframe 完全ガイド - Rust ペイントアプリから学ぶ 2

1.1 目次 . . . 2
1.2 概要 . . . 2
1.2.1 eGUI とは . . . 2
1.2.2 eframe とは . . . 2
1.2.3 Immediate Mode vs Retained Mode . . . 2
1.3 セットアップ . . . 3
1.3.1 1. 基本的なプロジェクトセットアップ . . . 3
1.3.2 2. 最小限のアプリケーション . . . 3
1.3.3 3. 高度な設定例 . . . 3
1.4 概念 . . . 5
1.4.1 Immediate Mode GUI の流れ . . . 5
1.4.2 状態管理のベストプラクティス . . . 5
1.5 レイアウト . . . 6
1.5.1 パネルシステム . . . 6
1.5.2 レイアウトのカスタマイズ . . . 7
1.5.3 レスポンシブデザイン . . . 8
1.6 ウィジェット . . . 9
1.6.1 基本的なウィジェット . . . 9
1.6.2 入力ウィジェット . . . 10
1.6.3 高度なウィジェット . . . 11
1.7 描画 . . . 12
1.7.1 Painter の基本的な使用法 . . . 12
1.7.2 高度な描画テクニック . . . 14
1.7.3 ペイントアプリの描画実装 . . . 16
1.8 イベント . . . 19
1.8.1 キーボードイベント . . . 19
1.8.2 マウスイベント . . . 20
1.8.3 カスタムイベントシステム . . . 21
1.9 実践例 . . . 23
1.9.1 ペイントアプリの完全な実装例 . . . 23
1.9.2 テストの実装 . . . 26
1.10 最適化 . . . 27
1.10.1 パフォーマンス最適化 . . . 27
1.10.2 メモリ最適化 . . . 28
1.11 トラブル . . . 29
1.11.1 よくある問題と解決法 . . . 29
1.11.2 デバッグ手法 . . . 31
1.11.3 パフォーマンス計測 . . . 32
1.12 まとめ . . . 32
1.12.1 学習したポイント . . . 32
1.12.2 推奨される学習の進め方 . . . 33
1.12.3 次のステップ . . . 33

1 eGUI/eframe 完全ガイド - Rust ペイントアプリから学ぶ

1.1 目次

1. eGUI と eframe の概要
2. 基本的なセットアップ
3. Immediate Mode GUI の概念
4. レイアウトシステム
5. ウィジェット詳細解説
6. 描画とペインター
7. イベント処理
8. 実践的なコード例
9. パフォーマンス最適化
10. トラブルシューティング

1.2 概要

1.2.1 eGUI とは

eGUI (egui) は、Rust で書かれた Immediate Mode GUI ライブラリです。

1.2.1.1 主な特徴

- **Immediate Mode**: UI を毎フレーム再構築する方式
- **軽量**: 最小限の依存関係
- **クロスプラットフォーム**: Windows, macOS, Linux, WebAssembly に対応
- **Rust ネイティブ**: メモリ安全性とパフォーマンスを両立
- **直感的 API**: シンプルで理解しやすい設計

1.2.2 eframe とは

eframe は、eGUI の上に構築されたアプリケーションフレームワークです。

1.2.2.1 主な機能

- **ネイティブウィンドウ管理**: デスクトップアプリケーションの作成
- **ウェブデプロイ**: WebAssembly でのブラウザ実行
- **設定管理**: アプリケーション設定の永続化
- **ライフサイクル管理**: アプリケーションの開始・終了処理

1.2.3 Immediate Mode vs Retained Mode

特徴	Immediate Mode (eGUI)	Retained Mode (従来の GUI)
状態管理	アプリケーション側で管理	GUI 側で管理
UI の構築	毎フレーム再構築	一度構築して更新
メモリ使用量	軽量	重い（ウィジェットツリーを保持）
複雑性	シンプル	複雑
デバッグ	容易	困難（状態が分散）

1.3 セットアップ

1.3.1 1. 基本的なプロジェクトセットアップ

```
# Cargo.toml
[dependencies]
eframe = "0.31.1"
egui = "0.31.1"
```

1.3.2 2. 最小限のアプリケーション

```
use eframe::egui;

fn main() -> Result<(), eframe::Error> {
    let options = eframe::NativeOptions {
        viewport: egui::ViewportBuilder::default()
            .with_inner_size([320.0, 240.0]),
        ..Default::default()
    };

    eframe::run_native(
        "My App",
        options,
        Box::new(|_cc| Ok(Box::new(MyApp::default()))),
    )
}

#[derive(Default)]
struct MyApp;

impl eframe::App for MyApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        egui::CentralPanel::default().show(ctx, |ui| {
            ui.heading("Hello, World!");
        });
    }
}
```

1.3.3 3. 高度な設定例

```
fn main() -> Result<(), eframe::Error> {
    let options = eframe::NativeOptions {
        viewport: egui::ViewportBuilder::default()
            .with_inner_size([800.0, 600.0])
            .with_min_inner_size([400.0, 300.0])
```

```
            .with_icon(load_icon())
            .with_resizable(true)
            .with_maximize_button(true),

        // ハードウェアアクセラレーション
        renderer: eframe::Renderer::default(),

        // マルチサンプリング（アンチエイリアシング）
        multisampling: 4,

        // 開発時の設定
        run_and_return: false,

        ..Default::default()
    };

    eframe::run_native(
        "Advanced App",
        options,
        Box::new(|cc| {
            // ダークモード設定
            cc.egui_ctx.set_visuals(egui::Visuals::dark());

            // カスタムフォント読み込み
            setup_custom_fonts(&cc.egui_ctx);

            Ok(Box::new(MyApp::new(cc)))
        }),
    )
}

fn setup_custom_fonts(ctx: &egui::Context) {
    let mut fonts = egui::FontDefinitions::default();

    fonts.font_data.insert(
        "my_font".to_owned(),
        egui::FontData::from_static(include_bytes!("../assets/font.ttf")),
    );

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

    ctx.set_fonts(fonts);
}
```

1.4 概念

1.4.1 Immediate Mode GUI の流れ

```
impl eframe::App for MyApp {
    fn update(&mut self, ctx: &egui::Context, frame: &mut eframe::Frame) {
        // 1. UI の構築（毎フレーム実行）
        egui::TopBottomPanel::top("menu_bar").show(ctx, |ui| {
            // メニューバーの構築
            egui::menu::bar(ui, |ui| {
                ui.menu_button("File", |ui| {
                    if ui.button("New").clicked() {
                        // 2. ユーザーアクションの処理
                        self.new_file();
                    }
                    if ui.button("Open").clicked() {
                        self.open_file();
                    }
                });
            });
        });

        egui::CentralPanel::default().show(ctx, |ui| {
            // 3. メインコンテンツの構築
            ui.label(format!("Counter: {}", self.counter));
            if ui.button("Increment").clicked() {
                // 4. 状態の更新
                self.counter += 1;
            }
        });

        // 5. フレーム終了（自動的に再描画がスケジュールされる）
    }
}
```

1.4.2 状態管理のベストプラクティス

```
struct MyApp {
    // アプリケーション状態
    counter: i32,
    text_input: String,
    is_enabled: bool,

    // UI 状態（一時的なもの）
    show_popup: bool,
    selected_item: Option<usize>,
}
```

```rust
impl MyApp {
    // 状態変更は専用メソッドで
    fn increment(&mut self) {
        self.counter += 1;
    }

    fn reset(&mut self) {
        self.counter = 0;
        self.text_input.clear();
    }

    // 複雑な操作は分離
    fn handle_file_operation(&mut self, operation: FileOperation) {
        match operation {
            FileOperation::New => self.new_file(),
            FileOperation::Open => self.open_file(),
            FileOperation::Save => self.save_file(),
        }
    }
}
```

1.5 レイアウト

1.5.1 パネルシステム

eGUI は階層的なパネルシステムを採用しています。

```rust
impl eframe::App for MyApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        // トップレベルパネル
        egui::TopBottomPanel::top("top_panel").show(ctx, |ui| {
            ui.horizontal(|ui| {
                ui.label("Top Panel");
                ui.separator();
                ui.button("Button");
            });
        });

        egui::TopBottomPanel::bottom("bottom_panel").show(ctx, |ui| {
            ui.horizontal(|ui| {
                ui.label("Status: Ready");
                ui.with_layout(egui::Layout::right_to_left(egui::Align::Center), |ui| {
                    ui.label("Right aligned");
                });
            });
        });
```

```
        // サイドパネル
        egui::SidePanel::left("left_panel")
            .resizable(true)
            .default_width(200.0)
            .width_range(100.0..=300.0)
            .show(ctx, |ui| {
                ui.vertical(|ui| {
                    ui.heading("Tools");
                    ui.separator();

                    ui.selectable_value(&mut self.selected_tool, Tool::Brush, "Brush");
                    ui.selectable_value(&mut self.selected_tool, Tool::Eraser, "Eraser");
                    ui.selectable_value(&mut self.selected_tool, Tool::Line, "Line");
                });
            });

        egui::SidePanel::right("right_panel").show(ctx, |ui| {
            ui.vertical(|ui| {
                ui.heading("Properties");
                ui.separator();

                ui.add(egui::Slider::new(&mut self.brush_size, 1.0..=50.0)
                    .text("Size"));
                ui.color_edit_button_srgba(&mut self.color);
            });
        });

        // 中央パネル（最後に配置）
        egui::CentralPanel::default().show(ctx, |ui| {
            // メインコンテンツ
            self.render_canvas(ui);
        });
    }
}
```

1.5.2 レイアウトのカスタマイズ

```
// 水平レイアウト
ui.horizontal(|ui| {
    ui.label("Label 1");
    ui.label("Label 2");
    ui.label("Label 3");
});

// 垂直レイアウト
ui.vertical(|ui| {
```

```
    ui.button("Button 1");
    ui.button("Button 2");
    ui.button("Button 3");
});

// カスタムレイアウト
ui.with_layout(egui::Layout::top_down(egui::Align::Center), |ui| {
    ui.label("Centered");
    ui.button("Centered Button");
});

// グリッドレイアウト
egui::Grid::new("my_grid")
    .num_columns(2)
    .spacing([40.0, 4.0])
    .striped(true)
    .show(ui, |ui| {
        ui.label("Name:");
        ui.text_edit_singleline(&mut self.name);
        ui.end_row();

        ui.label("Age:");
        ui.add(egui::Slider::new(&mut self.age, 0..=120));
        ui.end_row();

        ui.label("Email:");
        ui.text_edit_singleline(&mut self.email);
        ui.end_row();
    });

// カスタム間隔
ui.spacing_mut().item_spacing = egui::vec2(20.0, 10.0);
ui.spacing_mut().button_padding = egui::vec2(8.0, 4.0);
```

1.5.3 レスポンシブデザイン

```
impl MyApp {
    fn render_responsive_ui(&mut self, ui: &mut egui::Ui) {
        let available_width = ui.available_width();

        if available_width > 800.0 {
            // 大画面レイアウト
            ui.horizontal(|ui| {
                ui.vertical(|ui| {
                    self.render_sidebar(ui);
                });
                ui.separator();
```

```
                ui.vertical(|ui| {
                    self.render_main_content(ui);
                });
            });
        } else {
            // 小画面レイアウト
            ui.vertical(|ui| {
                egui::CollapsingHeader::new("Tools")
                    .default_open(false)
                    .show(ui, |ui| {
                        self.render_sidebar(ui);
                    });
                ui.separator();
                self.render_main_content(ui);
            });
        }
    }
}
```

1.6 ウィジェット

1.6.1 基本的なウィジェット

```
impl MyApp {
    fn render_basic_widgets(&mut self, ui: &mut egui::Ui) {
        // テキスト表示
        ui.label("Simple label");
        ui.heading("Heading");
        ui.monospace("Monospace text");
        ui.small("Small text");

        // ボタン
        if ui.button("Click me").clicked() {
            println!("Button clicked!");
        }

        // カスタムサイズのボタン
        if ui.add(egui::Button::new("Custom Button")
            .min_size(egui::vec2(100.0, 30.0)))
            .clicked() {
            self.handle_custom_action();
        }

        // チェックボックス
        ui.checkbox(&mut self.is_enabled, "Enable feature");
```

```
        // ラジオボタン
        ui.radio_value(&mut self.selected_option, Option::A, "Option A");
        ui.radio_value(&mut self.selected_option, Option::B, "Option B");

        // セレクタブル値
        ui.selectable_value(&mut self.mode, Mode::Edit, "Edit Mode");
        ui.selectable_value(&mut self.mode, Mode::View, "View Mode");
    }
}
```

1.6.2 入力ウィジェット

```
impl MyApp {
    fn render_input_widgets(&mut self, ui: &mut egui::Ui) {
        // テキスト入力
        ui.horizontal(|ui| {
            ui.label("Name:");
            ui.text_edit_singleline(&mut self.name);
        });

        // パスワード入力
        ui.horizontal(|ui| {
            ui.label("Password:");
            ui.add(egui::TextEdit::singleline(&mut self.password)
                .password(true));
        });

        // 複数行テキスト
        ui.label("Description:");
        ui.add(egui::TextEdit::multiline(&mut self.description)
            .desired_rows(5)
            .desired_width(f32::INFINITY));

        // 数値入力
        ui.horizontal(|ui| {
            ui.label("Value:");
            ui.add(egui::DragValue::new(&mut self.value)
                .speed(0.1)
                .clamp_range(0.0..=100.0));
        });

        // スライダー
        ui.add(egui::Slider::new(&mut self.slider_value, 0.0..=1.0)
            .text("Opacity")
            .show_value(true));

        // カラーピッカー
```

```
        ui.horizontal(|ui| {
            ui.label("Color:");
            ui.color_edit_button_srgba(&mut self.color);
        });

        // カスタムカラーピッカー
        egui::color_picker::color_edit_button_hsva(
            ui,
            &mut self.hsva_color,
            egui::color_picker::Alpha::Opaque,
        );
    }
}
```

1.6.3 高度なウィジェット

```
impl MyApp {
    fn render_advanced_widgets(&mut self, ui: &mut egui::Ui) {
        // プログレスバー
        let progress = self.current_progress / self.max_progress;
        ui.add(egui::ProgressBar::new(progress)
            .text(format!("{:.1}%", progress * 100.0)));

        // セパレーター
        ui.separator();

        // 折りたたみヘッダー
        egui::CollapsingHeader::new("Advanced Settings")
            .default_open(false)
            .show(ui, |ui| {
                self.render_advanced_settings(ui);
            });

        // スクロールエリア
        egui::ScrollArea::vertical()
            .max_height(200.0)
            .show(ui, |ui| {
                for i in 0..100 {
                    ui.label(format!("Item {}", i));
                }
            });

        // コンボボックス
        egui::ComboBox::from_label("Select option")
            .selected_text(format!("{:?}", self.selected_option))
            .show_ui(ui, |ui| {
                ui.selectable_value(&mut self.selected_option, Option::A, "Option A");
```

```
                ui.selectable_value(&mut self.selected_option, Option::B, "Option B");
                ui.selectable_value(&mut self.selected_option, Option::C, "Option C");
            });

        // ハイパーリンク
        ui.hyperlink_to("eGUI Repository", "https://github.com/emilk/egui");

        // 画像表示
        if let Some(texture) = &self.image_texture {
            ui.image(texture);
        }

        // カスタムウィジェット
        let response = ui.allocate_response(
            egui::vec2(100.0, 100.0),
            egui::Sense::click(),
        );

        if response.clicked() {
            self.handle_custom_widget_click();
        }

        // カスタム描画
        let painter = ui.painter();
        painter.rect_filled(
            response.rect,
            5.0,
            egui::Color32::BLUE,
        );
    }
}
```

1.7 描画

1.7.1 Painter の基本的な使用法

```
impl MyApp {
    fn render_custom_drawing(&mut self, ui: &mut egui::Ui) {
        // 描画エリアの確保
        let (response, painter) = ui.allocate_painter(
            egui::vec2(400.0, 300.0),  // サイズ
            egui::Sense::drag(),       // インタラクション
        );

        // 背景の描画
        painter.rect_filled(
```

```
            response.rect,
            5.0,  // 角の半径
            egui::Color32::from_rgb(240, 240, 240),
        );

        // 基本図形の描画
        self.draw_shapes(&painter, response.rect);

        // マウスインタラクションの処理
        if response.dragged() {
            if let Some(pointer_pos) = response.interact_pointer_pos() {
                self.handle_drawing(pointer_pos, response.rect);
            }
        }
    }

    fn draw_shapes(&self, painter: &egui::Painter, rect: egui::Rect) {
        let center = rect.center();

        // 円の描画
        painter.circle_filled(
            center,
            50.0,
            egui::Color32::RED,
        );

        // 円の輪郭
        painter.circle_stroke(
            center,
            60.0,
            egui::Stroke::new(2.0, egui::Color32::BLACK),
        );

        // 四角形
        painter.rect_filled(
            egui::Rect::from_center_size(
                center + egui::vec2(100.0, 0.0),
                egui::vec2(50.0, 50.0),
            ),
            0.0,
            egui::Color32::GREEN,
        );

        // 線の描画
        painter.line_segment(
            [
                center + egui::vec2(-100.0, -50.0),
```

```
                center + egui::vec2(100.0, 50.0),
            ],
            egui::Stroke::new(3.0, egui::Color32::BLUE),
        );

        // テキストの描画
        painter.text(
            center + egui::vec2(0.0, 100.0),
            egui::Align2::CENTER_CENTER,
            "Custom Drawing",
            egui::FontId::default(),
            egui::Color32::BLACK,
        );
    }
}
```

1.7.2 高度な描画テクニック

```
impl MyApp {
    fn render_advanced_drawing(&mut self, ui: &mut egui::Ui) {
        let (response, painter) = ui.allocate_painter(
            ui.available_size(),
            egui::Sense::drag(),
        );

        // グラデーション背景
        self.draw_gradient_background(&painter, response.rect);

        // アニメーション
        self.draw_animated_elements(&painter, response.rect, ui.ctx());

        // インタラクティブ要素
        self.handle_interactive_drawing(&response, &painter);
    }

    fn draw_gradient_background(&self, painter: &egui::Painter, rect: egui::Rect) {
        // 垂直グラデーション
        let colors = [
            (0.0, egui::Color32::from_rgb(135, 206, 235)),  // Sky blue
            (1.0, egui::Color32::from_rgb(255, 255, 255)),  // White
        ];

        for y in 0..rect.height() as i32 {
            let t = y as f32 / rect.height();
            let color = self.interpolate_color(&colors, t);

            painter.hline(
```

```
                rect.min.x..=rect.max.x,
                rect.min.y + y as f32,
                egui::Stroke::new(1.0, color),
            );
        }
    }

    fn draw_animated_elements(&mut self, painter: &egui::Painter, rect: egui::Rect, ctx: &egui::Conte
        // 時間ベースのアニメーション
        let time = ctx.input(|i| i.time) as f32;

        // 回転する要素
        let center = rect.center();
        let radius = 100.0;
        let angle = time * 2.0;

        let pos = center + egui::vec2(
            angle.cos() * radius,
            angle.sin() * radius,
        );

        painter.circle_filled(pos, 10.0, egui::Color32::RED);

        // 継続的な再描画をリクエスト
        ctx.request_repaint();
    }

    fn interpolate_color(&self, colors: &[(f32, egui::Color32)], t: f32) -> egui::Color32 {
        if colors.len() < 2 {
            return colors.first().map(|(_, c)| *c).unwrap_or(egui::Color32::BLACK);
        }

        let t = t.clamp(0.0, 1.0);

        for i in 0..colors.len() - 1 {
            let (t1, c1) = colors[i];
            let (t2, c2) = colors[i + 1];

            if t >= t1 && t <= t2 {
                let local_t = (t - t1) / (t2 - t1);
                return self.lerp_color(c1, c2, local_t);
            }
        }

        colors.last().unwrap().1
    }
```

```
    fn lerp_color(&self, a: egui::Color32, b: egui::Color32, t: f32) -> egui::Color32 {
        egui::Color32::from_rgba_unmultiplied(
            (a.r() as f32 * (1.0 - t) + b.r() as f32 * t) as u8,
            (a.g() as f32 * (1.0 - t) + b.g() as f32 * t) as u8,
            (a.b() as f32 * (1.0 - t) + b.b() as f32 * t) as u8,
            (a.a() as f32 * (1.0 - t) + b.a() as f32 * t) as u8,
        )
    }
}
```

1.7.3 ペイントアプリの描画実装

```
impl CanvasHandler {
    pub fn render(&mut self, ui: &mut egui::Ui, tools: &ToolSettings) {
        let available_size = ui.available_size();

        // キャンバスの描画エリア
        let (response, painter) = ui.allocate_painter(
            available_size,
            egui::Sense::drag(),
        );

        // 背景を描画
        painter.rect_filled(
            response.rect,
            0.0,
            tools.background_color,
        );

        // マウス入力の処理
        self.handle_mouse_input(&response, tools);

        // 全てのストロークを描画
        self.render_strokes(&painter, response.rect.min.to_vec2());

        // ステータス情報の表示
        self.render_status(ui, available_size);
    }

    fn handle_mouse_input(&mut self, response: &egui::Response, tools: &ToolSettings) {
        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() {
                    // 新しいストロークを開始
                    self.current_stroke = Some(Stroke::new(
```

```
                        tools.get_current_color(),
                        tools.brush_size,
                    ));
                }

                if let Some(ref mut stroke) = self.current_stroke {
                    stroke.add_point(canvas_pos);
                }
            }
        }

        if response.drag_stopped() {
            // ストロークを完了
            if let Some(stroke) = self.current_stroke.take() {
                if stroke.len() > 1 {
                    self.strokes.push(stroke);
                }
            }
        }
    }

    fn render_strokes(&self, painter: &egui::Painter, offset: egui::Vec2) {
        // 完了したストロークを描画
        for stroke in &self.strokes {
            stroke.draw(painter, offset);
        }

        // 現在描画中のストロークを描画
        if let Some(ref stroke) = self.current_stroke {
            stroke.draw(painter, offset);
        }
    }
}

impl Stroke {
    pub fn draw(&self, painter: &egui::Painter, offset: egui::Vec2) {
        if self.points.is_empty() {
            return;
        }

        if self.points.len() == 1 {
            // 単一点の場合
            painter.circle_filled(
                self.points[0] + offset,
                self.width / 2.0,
                self.color,
            );
```

```
        } else {
            // 複数点の場合、滑らかに補間
            painter.circle_filled(
                self.points[0] + offset,
                self.width / 2.0,
                self.color,
            );

            for i in 1..self.points.len() {
                Self::draw_smooth_stroke(
                    painter,
                    self.points[i - 1] + offset,
                    self.points[i] + offset,
                    self.width,
                    self.color,
                );
            }
        }
    }

    pub fn draw_smooth_stroke(
        painter: &egui::Painter,
        p1: egui::Pos2,
        p2: egui::Pos2,
        width: f32,
        color: egui::Color32,
    ) {
        let distance = (p2 - p1).length();
        let step_size = (width / 4.0).max(1.0);
        let steps = (distance / step_size).ceil() as usize;

        if steps <= 1 {
            painter.circle_filled(p2, width / 2.0, color);
        } else {
            for i in 0..=steps {
                let t = i as f32 / steps as f32;
                let interpolated_pos = p1 + t * (p2 - p1);
                painter.circle_filled(interpolated_pos, width / 2.0, color);
            }
        }
    }
}
```

1.8 イベント

1.8.1 キーボードイベント

```
impl eframe::App for MyApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        // キーボード入力の処理
        ctx.input(|i| {
            // 個別キーの状態
            if i.key_pressed(egui::Key::Space) {
                self.toggle_play_pause();
            }

            if i.key_pressed(egui::Key::Delete) {
                self.delete_selected();
            }

            // 修飾キーとの組み合わせ
            if i.modifiers.ctrl && i.key_pressed(egui::Key::S) {
                self.save_file();
            }

            if i.modifiers.ctrl && i.key_pressed(egui::Key::Z) {
                self.undo();
            }

            if i.modifiers.ctrl && i.modifiers.shift && i.key_pressed(egui::Key::Z) {
                self.redo();
            }

            // 数字キー
            for (i, key) in [
                egui::Key::Num1, egui::Key::Num2, egui::Key::Num3,
                egui::Key::Num4, egui::Key::Num5,
            ].iter().enumerate() {
                if i.key_pressed(*key) {
                    self.select_tool(i);
                }
            }
        });

        // UI の構築
        egui::CentralPanel::default().show(ctx, |ui| {
            // メインコンテンツ
        });
    }
}
```

1.8.2 マウスイベント

```
impl MyApp {
    fn handle_mouse_events(&mut self, ui: &mut egui::Ui) {
        let (response, painter) = ui.allocate_painter(
            ui.available_size(),
            egui::Sense::click_and_drag(),
        );

        // 基本的なマウスイベント
        if response.clicked() {
            println!("Clicked at: {:?}", response.interact_pointer_pos());
        }

        if response.double_clicked() {
            self.handle_double_click(response.interact_pointer_pos());
        }

        if response.triple_clicked() {
            self.handle_triple_click();
        }

        // ドラッグイベント
        if response.dragged() {
            if let Some(pos) = response.interact_pointer_pos() {
                self.handle_drag(pos, &response);
            }
        }

        if response.drag_started() {
            self.start_drag_operation();
        }

        if response.drag_stopped() {
            self.end_drag_operation();
        }

        // ホバーイベント
        if response.hovered() {
            ui.output_mut(|o| o.cursor_icon = egui::CursorIcon::Crosshair);

            if let Some(pos) = response.hover_pos() {
                self.show_tooltip_at(ui, pos);
            }
        }

        // 右クリックメニュー
```

```
        response.context_menu(|ui| {
            if ui.button("Copy").clicked() {
                self.copy_selection();
                ui.close_menu();
            }
            if ui.button("Paste").clicked() {
                self.paste();
                ui.close_menu();
            }
            ui.separator();
            if ui.button("Delete").clicked() {
                self.delete_selection();
                ui.close_menu();
            }
        });

        // スクロールホイール
        ui.ctx().input(|i| {
            if !i.scroll_delta.is_zero() {
                self.handle_scroll(i.scroll_delta);
            }
        });
    }

    fn show_tooltip_at(&self, ui: &mut egui::Ui, pos: egui::Pos2) {
        egui::show_tooltip_at_pointer(ui.ctx(), egui::Id::new("tooltip"), |ui| {
            ui.label(format!("Position: {:.1}, {:.1}", pos.x, pos.y));
            ui.label(format!("Tool: {:?}", self.current_tool));
        });
    }
}
```

1.8.3 カスタムイベントシステム

```
#[derive(Debug, Clone)]
pub enum AppEvent {
    FileNew,
    FileOpen(String),
    FileSave(String),
    ToolChanged(Tool),
    ColorChanged(egui::Color32),
    CanvasCleared,
    StrokeCompleted(Stroke),
}

impl MyApp {
    fn handle_event(&mut self, event: AppEvent) {
```

```
        match event {
            AppEvent::FileNew => {
                self.canvas.clear();
                self.file_path = None;
                self.is_modified = false;
            },
            AppEvent::FileOpen(path) => {
                if let Ok(data) = self.load_file(&path) {
                    self.canvas = data;
                    self.file_path = Some(path);
                    self.is_modified = false;
                }
            },
            AppEvent::ToolChanged(tool) => {
                self.tools.current_tool = tool;
            },
            AppEvent::StrokeCompleted(stroke) => {
                self.canvas.add_stroke(stroke);
                self.is_modified = true;
            },
            _ => {}
        }
    }

    fn update_with_events(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        // イベントの収集
        let mut events = Vec::new();

        // UI 構築とイベント生成
        egui::TopBottomPanel::top("menu").show(ctx, |ui| {
            egui::menu::bar(ui, |ui| {
                ui.menu_button("File", |ui| {
                    if ui.button("New").clicked() {
                        events.push(AppEvent::FileNew);
                    }
                    if ui.button("Open").clicked() {
                        if let Some(path) = self.open_file_dialog() {
                            events.push(AppEvent::FileOpen(path));
                        }
                    }
                });
            });
        });

        // イベントの処理
        for event in events {
            self.handle_event(event);
```

```
        }
    }
}
```

1.9 実践例

1.9.1 ペイントアプリの完全な実装例

以下は、本プロジェクトのモジュール分割された構造を使った実践的な例です：

```
// main.rs
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])
            .with_title("Professional Paint App"),
        ..Default::default()
    };

    eframe::run_native(
        "Paint App",
        options,
        Box::new(|cc| Ok(Box::new(PaintApp::new(cc)))),
    )
}

// app.rs - メインアプリケーション
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,
    show_debug: bool,
}
```

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

        Self {
            tools: ToolSettings::default(),
            canvas: CanvasHandler::default(),
            show_debug: false,
        }
    }

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

impl eframe::App for PaintApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        // ショートカット処理
        self.handle_shortcuts(ctx);

        // メニューバー
        egui::TopBottomPanel::top("menu_bar").show(ctx, |ui| {
            egui::menu::bar(ui, |ui| {
                ui.menu_button("File", |ui| {
                    if ui.button("New").clicked() {
                        self.clear_canvas();
                    }
                    ui.separator();
                    if ui.button("Exit").clicked() {
                        ctx.send_viewport_cmd(egui::ViewportCommand::Close);
                    }
                });

                ui.menu_button("View", |ui| {
                    ui.checkbox(&mut self.show_debug, "Show Debug Info");
                });
            });
        });

        // ツールバー
        egui::TopBottomPanel::top("toolbar").show(ctx, |ui| {
            if render_toolbar(ui, &mut self.tools) {
                self.clear_canvas();
            }
        });
```

```
        // サイドバー
        egui::SidePanel::left("sidebar").show(ctx, |ui| {
            render_sidebar(ui, &mut self.tools);

            if self.show_debug {
                ui.separator();
                ui.heading("Debug Info");
                ui.label(format!("Strokes: {}", self.canvas.stroke_count()));
                ui.label(format!("Tool: {:?}", self.tools.current_tool));
                ui.label(format!("FPS: {:.1}", ctx.input(|i| 1.0 / i.stable_dt)));
            }
        });

        // メインキャンバス
        egui::CentralPanel::default().show(ctx, |ui| {
            self.canvas.render(ui, &self.tools);
        });
    }

    fn save(&mut self, storage: &mut dyn eframe::Storage) {
        // アプリケーション状態の保存
        eframe::set_value(storage, "tools", &self.tools);
        eframe::set_value(storage, "show_debug", &self.show_debug);
    }

    fn auto_save_interval(&self) -> std::time::Duration {
        std::time::Duration::from_secs(30)
    }
}

impl PaintApp {
    fn handle_shortcuts(&mut self, ctx: &egui::Context) {
        ctx.input_mut(|i| {
            if i.consume_key(egui::Modifiers::CTRL, egui::Key::N) {
                self.clear_canvas();
            }

            if i.consume_key(egui::Modifiers::NONE, egui::Key::B) {
                self.tools.current_tool = crate::tools::Tool::Pen;
            }

            if i.consume_key(egui::Modifiers::NONE, egui::Key::E) {
                self.tools.current_tool = crate::tools::Tool::Eraser;
            }
        });
    }
}
```

1.9.2 テストの実装

```
#[cfg(test)]
mod tests {
    use super::*;
    use eframe::egui;

    #[test]
    fn test_stroke_creation() {
        let stroke = Stroke::new(egui::Color32::RED, 5.0);
        assert_eq!(stroke.len(), 0);
        assert_eq!(stroke.color, egui::Color32::RED);
        assert_eq!(stroke.width, 5.0);
    }

    #[test]
    fn test_stroke_add_points() {
        let mut stroke = Stroke::new(egui::Color32::BLACK, 2.0);
        stroke.add_point(egui::pos2(0.0, 0.0));
        stroke.add_point(egui::pos2(10.0, 10.0));

        assert_eq!(stroke.len(), 2);
        assert_eq!(stroke.points[0], egui::pos2(0.0, 0.0));
        assert_eq!(stroke.points[1], egui::pos2(10.0, 10.0));
    }

    #[test]
    fn test_tool_settings() {
        let mut tools = ToolSettings::default();
        assert_eq!(tools.current_tool, Tool::Pen);

        tools.set_tool(Tool::Eraser);
        assert_eq!(tools.current_tool, Tool::Eraser);
        assert_eq!(tools.get_current_color(), tools.background_color);
    }

    // UI テストの例（実際のテストでは eframe のテストハーネスを使用）
    #[test]
    fn test_canvas_clear() {
        let mut canvas = CanvasHandler::new();

        // ストロークを追加
        let stroke = Stroke::new(egui::Color32::RED, 2.0);
        canvas.strokes.push(stroke);
        assert_eq!(canvas.stroke_count(), 1);

        // クリア
```

```
        canvas.clear();
        assert_eq!(canvas.stroke_count(), 0);
        assert!(canvas.current_stroke.is_none());
    }
}
```

1.10 最適化

1.10.1 パフォーマンス最適化

```
impl MyApp {
    fn optimized_rendering(&mut self, ui: &mut egui::Ui) {
        // 描画範囲の最適化
        let visible_rect = ui.clip_rect();

        // 必要な場合のみ再描画
        if self.needs_repaint() {
            ui.ctx().request_repaint();
        }

        // 大きなリストの仮想化
        self.render_virtualized_list(ui);

        // 重い処理の分散
        self.process_heavy_work_incrementally();
    }

    fn render_virtualized_list(&mut self, ui: &mut egui::Ui) {
        let row_height = 20.0;
        let visible_rows = (ui.available_height() / row_height).ceil() as usize;
        let total_rows = self.items.len();

        egui::ScrollArea::vertical().show_rows(
            ui,
            row_height,
            total_rows,
            |ui, row_range| {
                for row in row_range {
                    if let Some(item) = self.items.get(row) {
                        ui.horizontal(|ui| {
                            ui.label(format!("Item {}: {}", row, item.name));
                        });
                    }
                }
            },
        );
```

```
    }

    fn process_heavy_work_incrementally(&mut self) {
        const MAX_WORK_PER_FRAME: usize = 10;

        for _ in 0..MAX_WORK_PER_FRAME {
            if let Some(work_item) = self.work_queue.pop_front() {
                self.process_work_item(work_item);
            } else {
                break;
            }
        }
    }
}
```

1.10.2 メモリ最適化

```
impl CanvasHandler {
    // ストロークの効率的な管理
    fn optimize_strokes(&mut self) {
        // 古いストロークの削除
        if self.strokes.len() > MAX_STROKES {
            self.strokes.drain(0..self.strokes.len() - MAX_STROKES);
        }

        // ストロークの簡略化
        for stroke in &mut self.strokes {
            stroke.simplify();
        }
    }
}

impl Stroke {
    // 点の間引きによる簡略化
    fn simplify(&mut self) {
        if self.points.len() < 3 {
            return;
        }

        let epsilon = 1.0; // 許容誤差
        let simplified = self.douglas_peucker(&self.points, epsilon);
        self.points = simplified;
    }

    fn douglas_peucker(&self, points: &[egui::Pos2], epsilon: f32) -> Vec<egui::Pos2> {
        // Douglas-Peucker アルゴリズムの実装
        if points.len() < 3 {
```

```
            return points.to_vec();
        }

        // 実装の詳細は省略...
        points.to_vec()
    }
}
```

1.11 トラブル

1.11.1 よくある問題と解決法

```
// 問題のあるコード
impl eframe::App for MyApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        // 状態を変更したが再描画されない
        if self.some_condition {
            self.value += 1;
            // ctx.request_repaint(); // これが必要
        }
    }
}

// 修正版
impl eframe::App for MyApp {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        if self.some_condition {
            self.value += 1;
            ctx.request_repaint(); // 明示的に再描画をリクエスト
        }
    }
}
```

1.11.1.1 1. UI が更新されない

```
// 問題のあるコード
let response = ui.button("Click me");
if response.clicked() {  // クリックが検出されない場合
    // ...
}

// 修正版
if ui.button("Click me").clicked() {  // より確実な書き方
    // ...
}
```

```
// または
let response = ui.add(egui::Button::new("Click me"));
if response.clicked() {
    // ...
}
```

1.11.1.2 2. レスポンスが機能しない

```
impl MyApp {
    // 問題のあるコード：無制限にデータが蓄積
    fn add_item(&mut self, item: Item) {
        self.items.push(item); // 制限なし
    }

    // 修正版：制限を設ける
    fn add_item(&mut self, item: Item) {
        self.items.push(item);

        const MAX_ITEMS: usize = 1000;
        if self.items.len() > MAX_ITEMS {
            self.items.drain(0..self.items.len() - MAX_ITEMS);
        }
    }
}
```

1.11.1.3 3. メモリリークの回避

```
// カスタムフォントが表示されない場合
fn setup_fonts(ctx: &egui::Context) {
    let mut fonts = egui::FontDefinitions::default();

    // フォントデータの追加
    fonts.font_data.insert(
        "my_font".to_owned(),
        egui::FontData::from_static(include_bytes!("../fonts/my_font.ttf")),
    );

    // 重要：フォントファミリーへの追加
    fonts.families
        .entry(egui::FontFamily::Proportional)
        .or_default()
        .insert(0, "my_font".to_owned()); // 最高優先度で追加

    // フォントの適用
    ctx.set_fonts(fonts);
```

```
}
```

1.11.1.4 4. フォントの問題

1.11.2 デバッグ手法

```
impl eframe::App for MyApp {
    fn update(&mut self, ctx: &egui::Context, frame: &mut eframe::Frame) {
        // デバッグ情報の表示
        if self.show_debug {
            egui::Window::new("Debug")
                .default_open(false)
                .show(ctx, |ui| {
                    ui.label(format!("FPS: {:.1}", 1.0 / ctx.input(|i| i.stable_dt)));
                    ui.label(format!("Mouse pos: {:?}", ctx.input(|i| i.pointer.hover_pos())));
                    ui.label(format!("Memory usage: {}MB", self.get_memory_usage()));

                    ui.separator();

                    // 状態の詳細表示
                    ui.collapsing("App State", |ui| {
                        ui.label(format!("Items count: {}", self.items.len()));
                        ui.label(format!("Current tool: {:?}", self.current_tool));
                    });

                    // リアルタイム値の監視
                    ui.add(egui::plot::Plot::new("value_plot")
                        .height(100.0)
                        .show_axes([true, true])
                        .show(|plot_ui| {
                            let points: Vec<_> = self.debug_values
                                .iter()
                                .enumerate()
                                .map(|(i, &v)| [i as f64, v as f64])
                                .collect();

                            plot_ui.line(egui::plot::Line::new(points));
                        }));
                });
        }

        // メイン UI
        egui::CentralPanel::default().show(ctx, |ui| {
            // ...
        });
    }
}
```

1.11.3 パフォーマンス計測

```rust
use std::time::Instant;

impl MyApp {
    fn performance_monitoring(&mut self, ctx: &egui::Context) {
        let start_time = Instant::now();

        // 重い処理
        self.heavy_computation();

        let duration = start_time.elapsed();

        // 閾値を超えた場合に警告
        if duration.as_millis() > 16 {  // 60fps 維持のため
            eprintln!("Warning: Frame took {}ms", duration.as_millis());
        }

        // 統計情報の更新
        self.frame_times.push(duration.as_secs_f32());
        if self.frame_times.len() > 60 {  // 直近 60 フレーム分のみ保持
            self.frame_times.remove(0);
        }
    }

    fn get_average_frame_time(&self) -> f32 {
        if self.frame_times.is_empty() {
            return 0.0;
        }

        self.frame_times.iter().sum::<f32>() / self.frame_times.len() as f32
    }
}
```

1.12 まとめ

このガイドでは、eGUI と eframe を使用した Rust ペイントアプリケーションの実装を通じて、以下の重要なトピックを学習しました：

1.12.1 学習したポイント

1. **Immediate Mode GUI の概念**
   - 状態管理の簡素化
   - 毎フレーム再構築の利点
   - デバッグの容易さ
2. **eGUI/eframe の基本**
   - アプリケーションのセットアップ

    - レイアウトシステム
    - ウィジェットの使用法
3. **高度な機能**
    - カスタム描画
    - イベント処理
    - パフォーマンス最適化
4. **実践的な実装**
    - モジュール分割
    - テスト手法
    - デバッグ技術

1.12.2　推奨される学習の進め方

1. **基本から始める**: 簡単なアプリケーションから始めて、徐々に機能を追加
2. **公式ドキュメント**: eGUI 公式ドキュメントを参照
3. **サンプルコード**: eGUI リポジトリの examples を研究
4. **コミュニティ**: Discord や GitHub Discussions を活用

1.12.3　次のステップ

- **WebAssembly**: ブラウザでの実行
- **カスタムウィジェット**: 独自の UI コンポーネント作成
- **プラグインシステム**: 拡張可能なアーキテクチャ
- **3D 統合**: threeD との組み合わせ

eGUI と eframe は、Rust エコシステムにおいて非常に強力で使いやすい GUI ソリューションです。このガイドが、皆さんの創造的なアプリケーション開発の助けになることを願っています。