Contents

1 Rust 2D グラフィックスフレームワーク分析レポート

1.1 ペイントアプリケーションにおけるレイヤー合成機能の詳細比較

1.2 1. macroquad - 2D ゲームフレームワーク

1.2.1 レイヤー合成機能

- **レンダーターゲット対応**: ☒ 完全サポート
- **テクスチャレンダリング**: ☒ `render_target()` 関数でオフスクリーンレンダリング可能
- **アルファブレンディング**: ☒ `draw_texture_ex()` でアルファ合成対応
- **マルチレイヤー**: ☒複数のレンダーターゲットを使用したレイヤー管理可能

```
// レイヤー作成例
let render_target = render_target(320, 150);
set_texture_filter(render_target.texture, FilterMode::Nearest);

// レイヤーに描画
set_camera(Camera2D {
    zoom: vec2(0.01, 0.01),
    target: vec2(0.0, 0.0),
    render_target: Some(render_target),
    ..Default::default()
});

// レイヤーを画面に合成
draw_texture_ex(
    render_target.texture,
    0., 0., WHITE,
    DrawTextureParams {
        dest_size: Some(vec2(1.0, 1.0)),
        flip_y: true, // 上下反転問題の回避
        ..Default::default()
    },
);
```

1.2.2 ペイントアプリケーション適性

- **ブラシストローク**: ☒ 線や図形描画 API 豊富
- **リアルタイム描画**: ☒ 高速な 2D 描画バッチング
- **筆圧対応**: ⚠ 基本的な入力処理、筆圧は追加実装必要
- **実例**: drawing-app - スタイラス対応のペイントアプリ

1.2.3 GUI 統合

- **内蔵 UI**: ☒ `macroquad::ui` モジュール提供
- **外部 GUI**: ☒ eGUI やその他の GUI ライブラリと併用可能
- **ツールバー**: ☒ 簡単な UI 要素でツールバー実装可能

1.2.4 パフォーマンス

- **描画性能**: ☒ 自動ジオメトリバッチング
- **メモリ使用量**: ☒ 軽量設計
- **リアルタイム**: ☒ 60FPS 維持可能

1.2.5 学習コスト

- **eGUI からの移行**: ☒☒☒ 中程度 - 類似 API だが描画パラダイム変更必要
- **ドキュメント**: ☒ 豊富な例とチュートリアル
- **コミュニティ**: ☒ 活発な開発とサポート

1.2.6 プラットフォーム対応

- **デスクトップ**: ☒ Windows/Mac/Linux
- **モバイル**: ☒ iOS/Android
- **Web**: ☒ WebAssembly 対応

1.3 2. miniquad - 低レベルグラフィックス

1.3.1 レイヤー合成機能

- **レンダーターゲット対応**: ☒完全サポート、`examples/offscreen.rs` で実装例
- **マルチパスレンダリング**: ☒ 複数のレンダーパス組み合わせ可能
- **テクスチャ操作**: ☒ 低レベルテクスチャ制御
- **アルファブレンディング**: ☒ OpenGL/Metal/WebGL 全対応

```
// オフスクリーンレンダリング例
let color_img = Image::gen_image_color(OFFSCREEN_SIZE, OFFSCREEN_SIZE, WHITE);
let depth_img = Image::gen_image_color(OFFSCREEN_SIZE, OFFSCREEN_SIZE, WHITE);
let render_texture = Texture::from_rgba8(ctx, OFFSCREEN_SIZE, OFFSCREEN_SIZE, &color_img.byt
let depth_texture = Texture::from_rgba8(ctx, OFFSCREEN_SIZE, OFFSCREEN_SIZE, &depth_img.byte

// レンダーパス設定
let pass = RenderPass::new(ctx, render_texture, depth_texture);
```

1.3.2 ペイントアプリケーション適性

- **柔軟性**: ☒ 低レベル制御で高度なブラシ効果実装可能
- **パフォーマンス**: ☒ 最適化された描画パイプライン
- **カスタマイズ**: ☒ シェーダーレベルでの制御可能
- **複雑さ**: ⚠ 高度な機能は実装コストが高い

1.3.3 GUI 統合

- **GUI 非依存**: ☒ あらゆる GUI フレームワークと統合可能
- **eGUI**: ☒ `egui-miniquad` で簡単統合
- **カスタム UI**: ☒ 完全カスタム UI 実装可能

1.3.4 パフォーマンス

- **最高レベル**: ☒ ハードウェア最適化
- **メモリ効率**: ☒ 細かな制御可能
- **クロスプラットフォーム**: ☒ 統一 API

1.3.5 学習コスト

- **eGUI からの移行**: ☒☒☒☒☒ 高難度 - 完全に異なるパラダイム
- **OpenGL 知識**: 必要 - 3D グラフィックス基礎知識推奨
- **コミュニティ**: ⚠ 小規模だが質の高いサポート

1.3.6 プラットフォーム対応

- **網羅的**: ☒ Desktop/Web/Mobile/その他全対応
- **安全性**: ☒ unsafe code 不使用

1.4 3. nannou - クリエイティブコーディングフレームワーク

1.4.1 レイヤー合成機能

- **フレームベースレンダリング**: ☒ 高品質な静止画/動画出力
- **WebGPU**: ☒ 最新 GPU 技術活用
- **色彩ブレンディング**: ☒ 高精度浮動小数点色処理
- **シェーダー対応**: ☒ ISF（Interactive Shader Format）対応

1.4.2 ペイントアプリケーション適性

- **アーティスト指向**: ☒ 美術表現に特化した設計
- **ブラシシミュレーション**: ☒ 油絵シミュレーション実績
- **リアルタイム**: ⚠ 高品質優先でリアルタイム性は二次的
- **表現力**: ☒ 最高レベルの視覚表現可能

1.4.3 GUI 統合

- **egui 統合**: ☒ nannou_egui で直接統合
- **マルチウィンドウ**: ☒ 高度なマルチウィンドウ対応
- **UI 優先度**: ☒ UI 統合を重視した設計

1.4.4 パフォーマンス

- **高品質**: ☒ 品質優先
- **GPU アクセラレーション**: ☒ フル GPU 活用
- **ビルド時間**: ⚠ 初回ビルド数分、リビルド 4 秒程度

1.4.5 学習コスト

- **eGUI からの移行**: ☒☒☒ 中程度 - 統合サポートあり
- **アーティスト親和性**: ☒ Processing/OpenFrameworks 経験者に最適
- **ドキュメント**: ☒ 豊富なチュートリアルとコミュニティ

1.4.6 プラットフォーム対応

- **デスクトップ**: ☒ 全プラットフォーム
- **Web**: ☒ WebAssembly 目標（開発中）
- **インストレーション**: ☒ プロダクション対応

1.5 4. piston - モジュラーゲームエンジン

1.5.1 レイヤー合成機能

- **2D グラフィックス**: ☒ 独立した 2D グラフィックスライブラリ
- **マルチバックエンド**: ☒ 複数のレンダリングバックエンド対応
- **モジュラー設計**: ☒ 必要な機能のみ選択可能
- **カスタムバックエンド**: ☒ Graphics トレイトで独自実装可能

```
// 基本描画例
window.draw_2d(&e, |c, g, _device| {
    clear([1.0; 4], g);
    rectangle([1.0, 0.0, 0.0, 1.0], // red
        [0.0, 0.0, 100.0, 100.0],
        c.transform, g);
});
```

1.5.2 ペイントアプリケーション適性

- **イベント駆動**: ☒ リアルタイム入力処理に適している
- **レイヤー管理**: ☒ モジュラー設計でレイヤー実装可能
- **柔軟性**: ☒ ゲームエンジンとしての豊富な機能
- **成熟度**: ⚠ 2014 年開始、安定だが発展停滞気味

1.5.3 GUI 統合

- **即時モード UI**: ☒ 内蔵の即時モード UI
- **独立性**: ☒ ウィンドウバックエンドと 2D グラフィックスの分離
- **カスタム実装**: ☒ 完全なカスタマイゼーション可能

1.5.4 パフォーマンス

- **モジュラー**: ☒ 必要な機能のみでオーバーヘッド最小化
- **最適化**: ☒ ゲームエンジンとしての最適化
- **メモリ効率**: ☒ RAII 設計

1.5.5 学習コスト

- **eGUI からの移行**: ☒☒☒☒ やや高い - 異なる設計思想
- **モジュラー理解**: 必要 - 各モジュールの役割理解が重要
- **ドキュメント**: ⚠ 豊富だが散在している

1.5.6 プラットフォーム対応

- **クロスプラットフォーム**: ☒ 主要デスクトップ対応
- **バックエンド選択**: ☒ プラットフォーム固有最適化可能

1.6 5. raylib-rs - Raylib バインディング

1.6.1 レイヤー合成機能

- **RenderTexture2D**: ☒ 専用のレンダーテクスチャ構造体
- **テクスチャ更新**: ☒ 動的テクスチャ更新対応
- **イメージ処理**: ☒ ピクセルレベルアクセス可能
- **RAII**: ☒ 自動リソース管理

```
// レイヤー作成・描画例
let mut render_texture = rl.load_render_texture(&thread, 800, 600).unwrap();

// レイヤーに描画
{
    let mut texture_draw = rl.begin_texture_mode(&thread, &mut render_texture);
    texture_draw.clear_background(Color::TRANSPARENT);
    texture_draw.draw_circle(100, 100, 50, Color::RED);
}

// レイヤーを画面に合成
let mut d = rl.begin_drawing(&thread);
d.draw_texture(&render_texture, 0, 0, Color::WHITE);
```

1.6.2 ペイントアプリケーション適性

- **簡単さ**: ☒ 学習コストが低い

- **リアルタイム**: ☒ 高速 2D 描画
- **マウス描画例**: ☒ 公式にマウス描画サンプルあり
- **テクスチャ操作**: ☒ ピクセル単位の操作可能

1.6.3 GUI 統合

- **内蔵 GUI**: ☒ 基本的な GUI 要素提供
- **外部統合**: ⚠ 限定的だが可能
- **即時モード**: ☒ 即時モード GUI 対応

1.6.4 パフォーマンス

- **軽量**: ☒ C 言語ベースで高速
- **メモリ安全**: ☒ Rust の安全性 + RAII
- **スレッドセーフ**: ☒ Rust の変更可能性ルールにより保証

1.6.5 学習コスト

- **eGUI からの移行**: ☒☒ 低い - 類似の即時モード思想
- **C API 風**: ⚠ C 言語の raylib 知識が役立つ
- **ドキュメント**: ☒ raylib 公式ドキュメント + Rust バインディング

1.6.6 プラットフォーム対応

- **幅広い対応**: ☒ Windows/Mac/Linux/Web/Mobile
- **自動リンク**: ☒ ビルドシステム自動化

1.7 6. skia-safe - Skia グラフィックスライブラリバインディング

1.7.1 レイヤー合成機能

- **プロフェッショナル**: ☒ Chrome/Android/Flutter で使用される実績
- **Surface/Canvas**: ☒ 高度なレイヤー管理システム
- **エフェクト/シェーダー**: ☒ 豊富な視覚効果
- **ベクターサポート**: ☒ ベクターグラフィックス完全対応

```
// 高度なレイヤー合成例
let mut surface = surfaces::raster_n32_premul((512, 512)).unwrap();
let canvas = surface.canvas();

// レイヤー作成
let layer_rect = Rect::from_wh(400.0, 300.0);
canvas.save_layer(&SaveLayerRec::default().bounds(&layer_rect));

// 描画操作
let mut paint = Paint::default();
paint.set_color(Color::BLUE);
```

```
canvas.draw_circle((100.0, 100.0), 50.0, &paint);

canvas.restore();
```

1.7.2 ペイントアプリケーション適性

- **最高品質**: ☒ プロ仕様の描画品質
- **テキスト処理**: ☒ HarfBuzz/ICU 統合
- **ブラシエンジン**: ☒ 高度なペイント効果可能
- **パフォーマンス**: ☒ GPU 最適化済み

1.7.3 GUI 統合

- **専門性**: ⚠ 描画エンジンとしての性格、GUI 統合は別途必要
- **カスタム UI**: ☒ 完全なカスタム UI 実装可能
- **ハイエンド**: ☒ 最高品質の UI 表現可能

1.7.4 パフォーマンス

- **最適化**: ☒ Google による産業レベル最適化
- **GPU アクセラレーション**: ☒ Vulkan/Metal/OpenGL/D3D 対応
- **メモリ効率**: ☒ 大規模アプリケーション対応

1.7.5 学習コスト

- **eGUI からの移行**: ☒☒☒☒☒ 最高難度 - 完全に異なるアプローチ
- **C++ API 理解**: 必要 - 元の Skia C++ API 知識が重要
- **専門知識**: 必要 - 2D グラフィックス深い理解推奨

1.7.6 プラットフォーム対応

- **完全対応**: ☒ デスクトップ/モバイル/Web 全対応
- **産業品質**: ☒ 商用アプリケーション実績豊富

1.8 推奨フレームワーク

1.8.1 1 位: macroquad（総合バランス最優秀）

理由: - レイヤー合成機能が完備 - eGUI からの移行コストが適度 - 実際のペイントアプリ実装例あり - 良好なパフォーマンスと使いやすさのバランス

1.8.2 2 位: raylib-rs（学習コスト最小）

理由: - 最も学習しやすい - 十分なレイヤー機能 - 公式マウス描画サンプルあり - 軽量で高速

1.8.3 3 位: miniquad（高度なカスタマイゼーション）

理由: - 最高の柔軟性 - eGUI 統合サポート - 低レベル制御可能 - 学習コストは高いが将来性抜群

1.9 移行推奨ステップ

1. **プロトタイプ段階**: raylib-rs で基本機能実装
2. **機能拡張段階**: macroquad で高度なレイヤー管理
3. **最適化段階**: 必要に応じて miniquad で最適化

各フレームワークとも、現在の eGUI ベースのアプリケーションから段階的に移行可能ですわ。