03. レンダリングパイプライン詳解

Matcap Maker の描画エンジンは、**Ping-Pong Framebuffer Architecture** を採用しています。これは画像処理ソフトのレイヤー合成をGPU上で効率的に行うための標準的な手法です。

Ping-Pong FBO 戦略

FBO (Frame Buffer Object) を交互に切り替えることで、メモリ割り当てを最小限に抑えつつ、無限の合成ステップを実現しています。

メモリリソース構成

- **Texture A (`fbo_ping`)**: 前のステップまでの合成結果（Accumulator）。
- **Texture B (`fbo_pong`)**: 現在の合成結果の書き込み先。
- **Texture C (`fbo_layer`)**: 現在のレイヤー単体の描画結果（Temporary）。

レンダリングフロー (疑似コード)

```
def render(layer_stack):
    # 1. バッファのクリア
    clear(fbo_ping, color=transparent)
    clear(fbo_pong, color=transparent)

    # 2. Accumulator (ping) が空の状態からスタート
    current_accumulator = fbo_ping
    next_accumulator = fbo_pong

    for layer in layer_stack:
        if not layer.enabled: continue

        # --- Phase 1: レイヤー個別の描画 ---
        # そのレイヤーのシェーダー（例: Noise Shader）を使って fbo_layer に描画
        bind_framebuffer(fbo_layer)
        layer.render() # Uniform設定 & DrawCall

        # --- Phase 2: 合成 (Blending) ---
        # fbo_layer (Src) と current_accumulator (Dst) を合成して
        # next_accumulator に書き込む
        bind_framebuffer(next_accumulator)
        use_shader(blend_shader)

        set_uniform("uSrc", fbo_layer.texture)
        set_uniform("uDst", current_accumulator.texture)
        set_uniform("uMode", layer.blend_mode) # Add, Multiply, Overlay...

        draw_quad() # 全画面矩形描画

        # --- Phase 3: スワップ ---
        # ping と pong を入れ替える
```

```
        current_accumulator, next_accumulator = next_accumulator,
current_accumulator

    # 3. 最終結果
    return current_accumulator.texture
```

高度なブレンドモードの実装

OpenGLの固定機能である `glBlendFunc` (例: `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA`) は、単純なアルファ合成や加算合成には適していますが、Photoshopにあるような複雑なブレンドモード（オーバーレイ、ソフトライト、覆い焼きカラーなど）は表現できません。

そのため、本ツールでは **Programmable Blending** を採用しています。 つまり、ブレンド処理を完全にフラグメントシェーダー (`blend.frag`) 内の計算式として実装しています。

GLSLによるブレンド実装例

```
// Overlay: 背景が明るいか暗いかでScreenとMultiplyを切り替える
float blendOverlay(float b, float f) {
    return b < 0.5 ? (2.0 * b * f) : (1.0 - 2.0 * (1.0 - b) * (1.0 - f));
}

// Soft Light: Photoshop仕様の近似式
float blendSoftLight(float b, float f) {
    return (f < 0.5)
        ? (2.0 * b * f + b * b * (1.0 - 2.0 * f))
        : (sqrt(b) * (2.0 * f - 1.0) + 2.0 * b * (1.0 - f));
}
```

このアプローチにより、あらゆる数学的な合成ルールを自由に追加することが可能です。アルファの取り扱い（Premultiplied Alpha vs Straight Alpha）についても、シェーダー内で厳密に制御しています。