Manual de Desenvolvimento de Efeitos RGB para B450M Steel Legend

Este manual foi criado para orientar desenvolvedores a criar e integrar novos efeitos RGB no sistema modular desenvolvido para a placa-mãe ASRock B450M Steel Legend, utilizando o OpenRGB como base de comunicação. Ele descreve a arquitetura do controlador, as funções principais disponíveis para criação de efeitos e exemplos práticos para implementação.

1 — Arquitetura Geral

O sistema é dividido em duas partes principais: o controlador e os efeitos modulares. O arquivo principal `openrgb_control.py` é o núcleo do programa, responsável por se conectar ao servidor OpenRGB, mapear os LEDs da placa-mãe e organizar os efeitos em categorias. Cada efeito é um script Python independente que pode ser executado de forma isolada, contanto que siga o padrão definido pelo controlador.

2 — Funcionamento do Controlador

O controlador inicializa a conexão com o OpenRGB, detecta dispositivos conectados (placa-mãe, RAM, cooler ARGB, etc.) e carrega automaticamente os efeitos da pasta `effects/`. Os efeitos são organizados em categorias: `Backplate`, `Chipset`, `Cooler`, `Memory_RAM` e `Global_Effects`. Cada efeito é exibido na interface gráfica (Tkinter), permitindo iniciar e parar manualmente. Quando o usuário clica em 'Iniciar', o controlador cria uma nova thread e executa a função `run_effect(ctx)` do módulo de efeito selecionado.

3 — Estrutura de Diretórios

O controlador cria automaticamente a estrutura de pastas ao ser executado pela primeira vez:

project-root/
■■ openrgb_control.py
■■ effects/
■ ■■ Backplate/
■ ■■ Chipset/
■ ■■ Cooler/
■ ■■ Memory_RAM/
■ ■■ Global_Effects/

Cada categoria pode conter quantos efeitos forem necessários. Cada efeito deve ser um arquivo `.py` contendo o nome e a função principal `run_effect`.

4 — Estrutura de um Efeito

Cada efeito precisa ter três elementos obrigatórios:
- `EFFECT_NAME`: nome que será exibido na interface gráfica.
- `ASK_COLORS`: define se o usuário poderá escolher cores (True ou False).
- `run_effect(ctx)`: função principal executada em loop até ser interrompida.

Exemplo básico:

```
EFFECT_NAME = "Exemplo Simples"
ASK_COLORS = False

def run_effect(ctx):
    import time
    leds = ctx["leds"]
```

```
    set_led = ctx["set_led"]
    delay = ctx["delay"]
    running = ctx["running"]
    while running():
        for n in leds:
            set_led(n, 255, 0, 0)
        time.sleep(delay(0.3))
        for n in leds:
            set_led(n, 0, 0, 255)
        time.sleep(delay(0.3))
```

5 — O Contexto (ctx)

A função `run_effect(ctx)` recebe um dicionário contendo todas as funções e dados necessários para manipular os LEDs e sincronizar o efeito. A seguir estão as principais chaves disponíveis:

• `leds`: lista de nomes de LEDs da categoria.
• `set_led(name, r, g, b)`: define a cor de um LED.
• `set_leds(list, r, g, b)`: define a cor de vários LEDs ao mesmo tempo.
• `delay(base_delay)`: ajusta a velocidade do efeito com base na escala configurada.
• `running()`: retorna True enquanto o efeito estiver ativo.
• `ram_devices`: lista de módulos de memória RAM detectados.
• `main_device`: dispositivo principal (placa-mãe).
• `client`: instância do OpenRGBClient.
• `piece_color` e `bg_color`: cores escolhidas pelo usuário quando `ASK_COLORS` é True.

Essas funções são suficientes para criar qualquer tipo de efeito de iluminação, desde animações simples até sincronizações complexas entre dispositivos.

6 — Mapa de LEDs da B450M Steel Legend

O controlador utiliza um mapa de LEDs fixo que associa nomes legíveis a índices físicos:

chipset 1–8 → região do chipset principal
backplate 1–10 → LEDs na placa traseira
saida argb 3 pino da placa mae ligou os cooler → saída ARGB para coolers

Esse mapeamento permite que os efeitos usem nomes intuitivos em vez de índices brutos.

7 — Criando um Novo Efeito (Passo a Passo)

1. Escolha uma categoria (ex: `Global_Effects`).
2. Crie um novo arquivo `.py` dentro da pasta correspondente.
3. Defina `EFFECT_NAME`, `ASK_COLORS` e `run_effect(ctx)`.
4. Dentro da função, use `ctx['set_led']` ou `ctx['set_leds']` para controlar LEDs.
5. Use `delay()` para controlar a velocidade e `running()` para sair do loop corretamente.
6. Teste o efeito iniciando pelo programa principal `openrgb_control.py`.
7. Se necessário, ajuste o mapeamento de LEDs no controlador para corresponder ao hardware.

8 — Boas Práticas e Dicas

• Sempre verifique `running()` em loops para garantir que o efeito possa ser interrompido.
• Use `delay()` em vez de `time.sleep()` fixo, pois ele respeita a velocidade global.
• Evite loops muito longos ou bloqueios.
• Use try/except ao alterar cores da RAM para evitar travamentos.

• Utilize cores entre 0 e 255.
• Evite modificar variáveis globais fora do escopo do efeito.
• Comente o código brevemente para facilitar colaboração.

9 — Exemplos de Efeitos

Abaixo alguns exemplos de efeitos práticos com diferentes níveis de complexidade.

```
# Exemplo simples — alterna entre vermelho e azul
EFFECT_NAME = "Alternar Cores"
ASK_COLORS = False
def run_effect(ctx):
    import time
    set_led = ctx["set_led"]
    leds = ctx["leds"]
    delay = ctx["delay"]
    running = ctx["running"]
    toggle = True
    while running():
        color = (255,0,0) if toggle else (0,0,255)
        for n in leds: set_led(n, *color)
        toggle = not toggle
        time.sleep(delay(0.5))
```

Exemplo intermediário — Pulso suave

Este exemplo cria um efeito de pulsação suave entre preto e a cor escolhida.

```
EFFECT_NAME = "Pulso Suave"
ASK_COLORS = True
def run_effect(ctx):
    import time
    leds = ctx["leds"]
    set_led = ctx["set_led"]
    delay = ctx["delay"]
    running = ctx["running"]
    piece = ctx.get("piece_color") or (0,150,255)
    while running():
        for t in range(0,101,5):
            color = tuple(int(c*t/100) for c in piece)
            for n in leds: set_led(n, *color)
            time.sleep(delay(0.03))
        for t in range(100,-1,-5):
            color = tuple(int(c*t/100) for c in piece)
            for n in leds: set_led(n, *color)
            time.sleep(delay(0.03))
```

Exemplo avançado — Efeito Tetris Global

Baseado no efeito `tetris_global.py`, este exemplo combina LEDs do backplate, RAM e chipset, simulando blocos caindo e fixando. Ele demonstra o potencial de sincronização entre múltiplos dispositivos RGB.

```
EFFECT_NAME = "Tetris Global Sincronizado"
ASK_COLORS = True
def run_effect(ctx):
    import time
    from openrgb.utils import RGBColor
    leds = ctx["leds"]
    ram = ctx["ram_devices"]
    set_led = ctx["set_led"]
    delay = ctx["delay"]
```

```
    running = ctx["running"]
    piece = ctx.get("piece_color", (0,120,255))
    bg = ctx.get("bg_color", (0,0,0))
    order = []
    for n in leds:
        if "backplate" in n: order.append(("main", n))
    for idx, dev in enumerate(ram):
        for i in range(min(8,len(dev.leds))):
            order.append(("ram",(idx,i)))
    while running():
        for i,(kind,data) in enumerate(order):
            if kind=="main": set_led(data,*piece)
            else: ram[data[0]].leds[data[1]].set_color(RGBColor(*piece))
            time.sleep(delay(0.05))
        for i,(kind,data) in enumerate(order):
            if kind=="main": set_led(data,*bg)
            else: ram[data[0]].leds[data[1]].set_color(RGBColor(*bg))
        time.sleep(delay(0.2))
```

10 — Conclusão

Com este manual, qualquer desenvolvedor pode criar e integrar novos efeitos RGB no sistema modular baseado no OpenRGB. Seguindo o padrão de contexto e boas práticas apresentadas aqui, é possível expandir o conjunto de efeitos para diferentes partes da placa-mãe e dispositivos conectados. Novos efeitos podem ser adicionados facilmente apenas copiando o arquivo para a pasta de categoria correta.