Processamento de Imagens

Alunos: Guilherme da Silva Schveitzer, Pedro Tsalikis de Melo, Rafael Pinho Medeiros

Carregar a Imagem - Comum nos códigos

```
// 1. Abrir a imagem
std::string imagePath = "../data/blobs-image.jpg";
cv::Mat originalImage = cv::imread(imagePath, cv::IMREAD_COLOR);

if (originalImage.empty()) {
    std::cerr << "Erro: Nao foi possivel carregar a imagem: " << imagePath << std::endl;
    return -1;
}

std::cout << "Imagem carregada com sucesso!" << std::endl;
std::cout << "Dimensoes: " << originalImage.cols << "x" << originalImage.rows << std::endl;
```

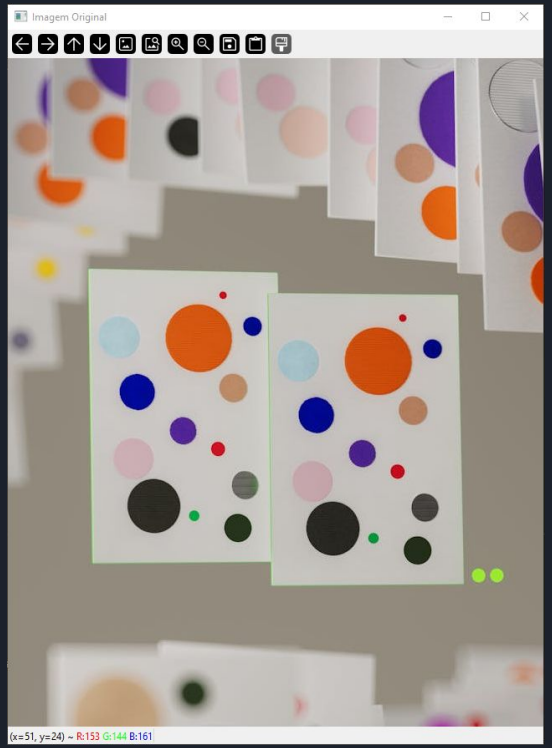
Run

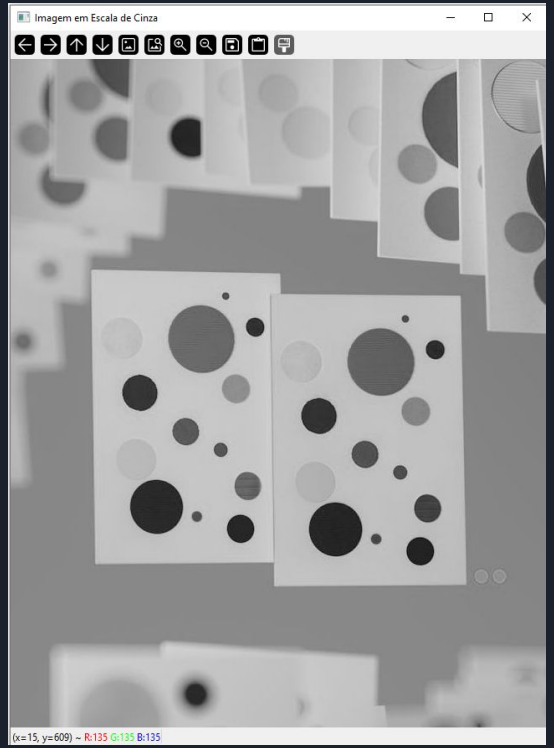
```
void exibir_menu() {
    std::cout << "\n======================================================" << std::endl;
    std::cout << "   PROCESSAMENTO DIGITAL DE IMAGENS - MENU PRINCIPAL" << std::endl;
    std::cout << "======================================================" << std::endl;
    std::cout << "Selecione uma opcao:" << std::endl;
    std::cout << "\n[1] Conversao para Escala de Cinza (Media Aritmetica)" << std::endl;
    std::cout << "[2] Conversao para Escala de Cinza (Media Ponderada)" << std::endl;
    std::cout << "[3] Operacoes Aritmeticas com Escalar" << std::endl;
    std::cout << "[4] Operacoes Aritmeticas entre Imagens" << std::endl;
    std::cout << "[5] Limiarizacao (Thresholding)" << std::endl;
    std::cout << "[6] Isolamento de Canais RGB" << std::endl;
    std::cout << "[7] Histogramas" << std::endl;
    std::cout << "[8] Inverso da Imagem (Negativo)" << std::endl;
    std::cout << "[0] Sair" << std::endl;
    std::cout << "\n======================================================" << std::endl;
    std::cout << "Digite sua opcao: ";
}
```

Conversão para Tons de Cinza(Média Aritmética) **Código**

```
// 3. Percorrer pixel a pixel a imagem
std::cout << "Processando pixels..." << std::endl;

for (int y = 0; y < originalImage.rows; y++) {
    for (int x = 0; x < originalImage.cols; x++) {
        // Obter os valores BGR do pixel atual
        cv::Vec3b pixel = originalImage.at<cv::Vec3b>(y, x);
        uchar blue = pixel[0];
        uchar green = pixel[1];
        uchar red = pixel[2];

        // 4. Calcular para cada pixel a media de valores r, g e b
        uchar grayValue = static_cast<uchar>((red + green + blue) / 3);

        // 5. Salvar o pixel correspondente na imagem de saida
        // Aplicar o valor de cinza nos tres canais para manter a imagem colorida
        resultImage.at<cv::Vec3b>(y, x) = cv::Vec3b(grayValue, grayValue, grayValue);
    }
}

std::cout << "Processamento concluido!" << std::endl;
```

Conversão para Tons de Cinza(Média Aritmética)
Resultado

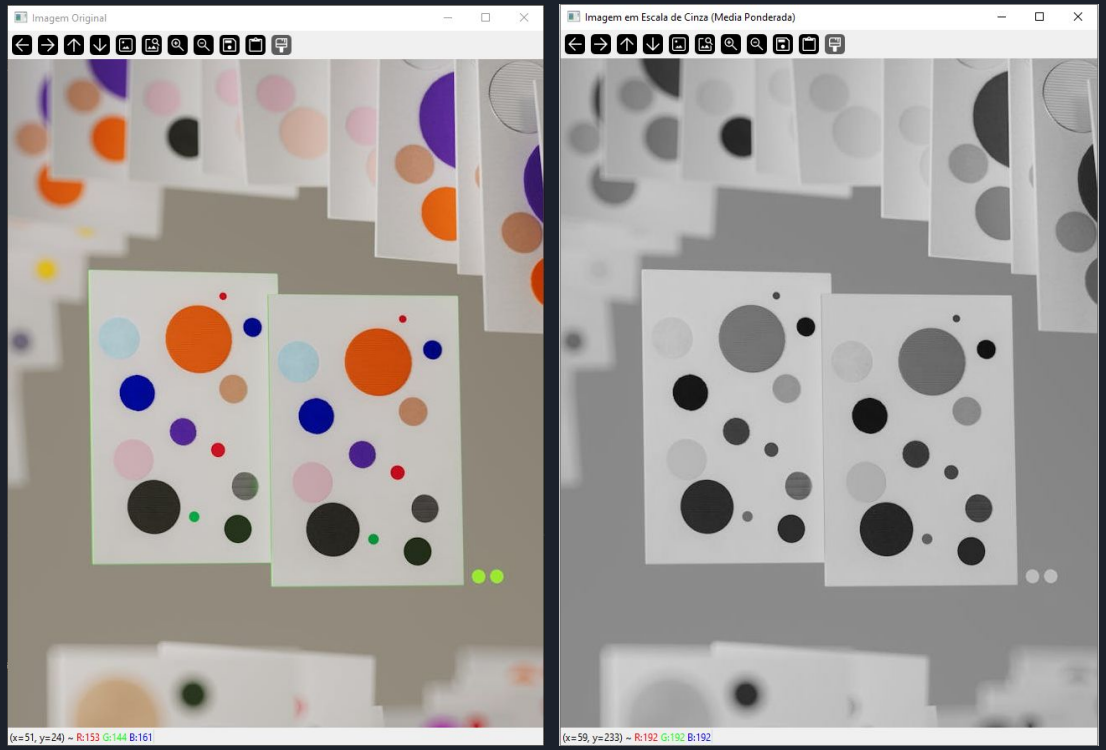




Conversão para Tons de Cinza(Média Ponderada) **Código**

```
// 3. Processar cada pixel com media ponderada
std::cout << "Processando pixels usando Media Ponderada..." << std::endl;

for (int y = 0; y < originalImage.rows; y++) {
    for (int x = 0; x < originalImage.cols; x++) {
        // Obter valores BGR
        cv::Vec3b pixel = originalImage.at<cv::Vec3b>(y, x);
        uchar blue = pixel[0];
        uchar green = pixel[1];
        uchar red = pixel[2];

        // Media ponderada: Gray = 0.299*R + 0.587*G + 0.114*B
        uchar grayValue = static_cast<uchar>(
            0.114f * blue + 0.587f * green + 0.299f * red
        );

        // Preencher todos os canais com o valor de cinza
        resultImage.at<cv::Vec3b>(y, x) = cv::Vec3b(grayValue, grayValue, grayValue);
    }
}

std::cout << "Processamento concluido!" << std::endl;
```

Conversão para Tons de Cinza(Média Ponderada)
Resultado

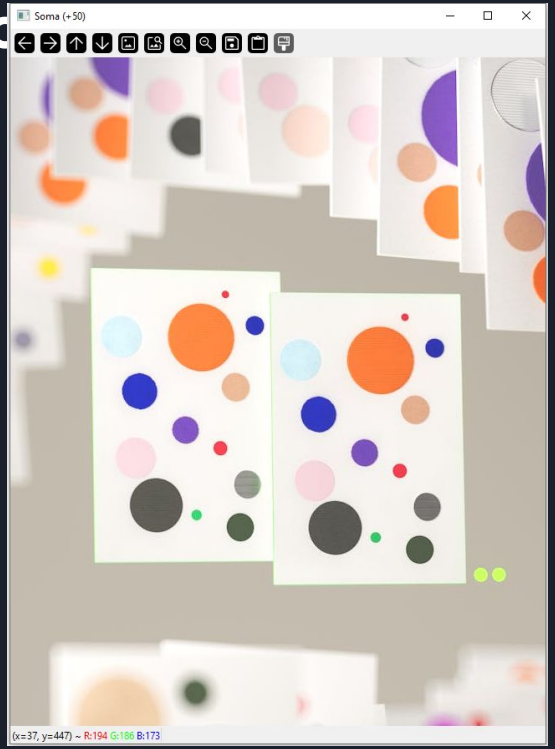


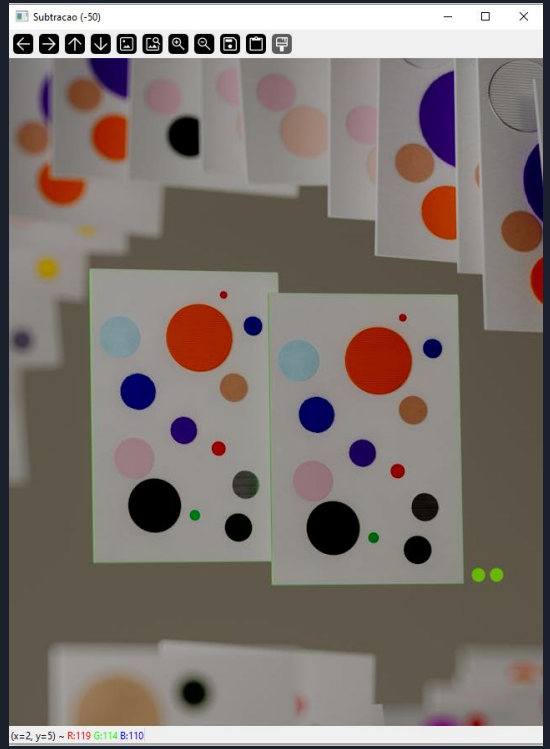
Operações Aritméticas (+ - * /) - Coloridas e tons de cinza
Código

```
// 2. Escalar usado
int valorEscalar = 50;
float fatorMultiplicacao = 1.5f;
float fatorDivisao = 2.0f;

std::cout << "Valor escalar utilizado: " << valorEscalar << std::endl;

// 3. Criar imagens de resultado
cv::Mat somaResult = cv::Mat::zeros(originalImage.size(), CV_8UC3);
cv::Mat subtracaoResult = cv::Mat::zeros(originalImage.size(), CV_8UC3);
cv::Mat multiplicacaoResult = cv::Mat::zeros(originalImage.size(), CV_8UC3);
cv::Mat divisaoResult = cv::Mat::zeros(originalImage.size(), CV_8UC3);
```

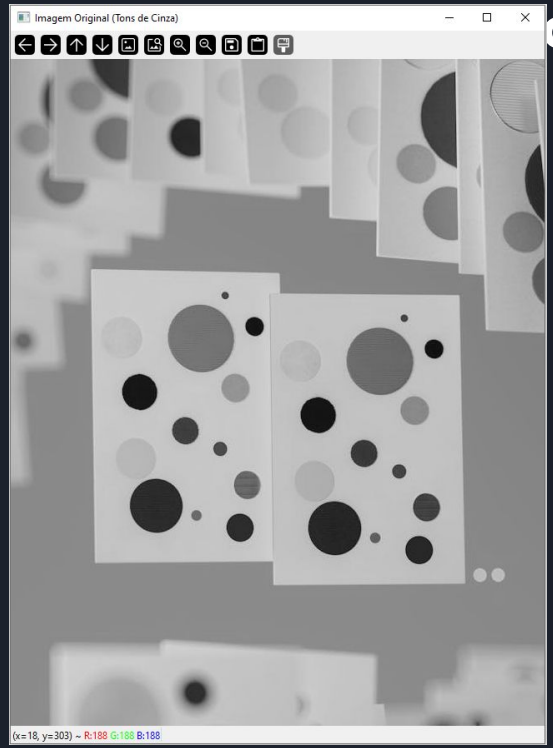
```
std::cout << "Processando operacoes aritmeticas..." << std::endl;

// 4. Percorrer pixels
for (int y = 0; y < originalImage.rows; y++) {
    for (int x = 0; x < originalImage.cols; x++) {
        cv::Vec3b pixel = originalImage.at<cv::Vec3b>(y, x);

        // SOMA
        cv::Vec3b somaPixel;
        somaPixel[0] = saturate_uchar(static_cast<int>(pixel[0]) + valorEscalar);
        somaPixel[1] = saturate_uchar(static_cast<int>(pixel[1]) + valorEscalar);
        somaPixel[2] = saturate_uchar(static_cast<int>(pixel[2]) + valorEscalar);
        somaResult.at<cv::Vec3b>(y, x) = somaPixel;

        // SUBTRACAO
        cv::Vec3b subPixel;
        subPixel[0] = saturate_uchar(static_cast<int>(pixel[0]) - valorEscalar);
        subPixel[1] = saturate_uchar(static_cast<int>(pixel[1]) - valorEscalar);
        subPixel[2] = saturate_uchar(static_cast<int>(pixel[2]) - valorEscalar);
        subtracaoResult.at<cv::Vec3b>(y, x) = subPixel;

        // MULTIPLICACAO
        cv::Vec3b multPixel;
        multPixel[0] = saturate_uchar(pixel[0] * fatorMultiplicacao);
        multPixel[1] = saturate_uchar(pixel[1] * fatorMultiplicacao);
        multPixel[2] = saturate_uchar(pixel[2] * fatorMultiplicacao);
        multiplicacaoResult.at<cv::Vec3b>(y, x) = multPixel;

        // DIVISAO (evitando divisao por zero — opcional aqui, mas mantido por seguranca)
        cv::Vec3b divPixel;
        divPixel[0] = saturate_uchar(pixel[0] / fatorDivisao);
        divPixel[1] = saturate_uchar(pixel[1] / fatorDivisao);
        divPixel[2] = saturate_uchar(pixel[2] / fatorDivisao);
        divisaoResult.at<cv::Vec3b>(y, x) = divPixel;
    }
}
```

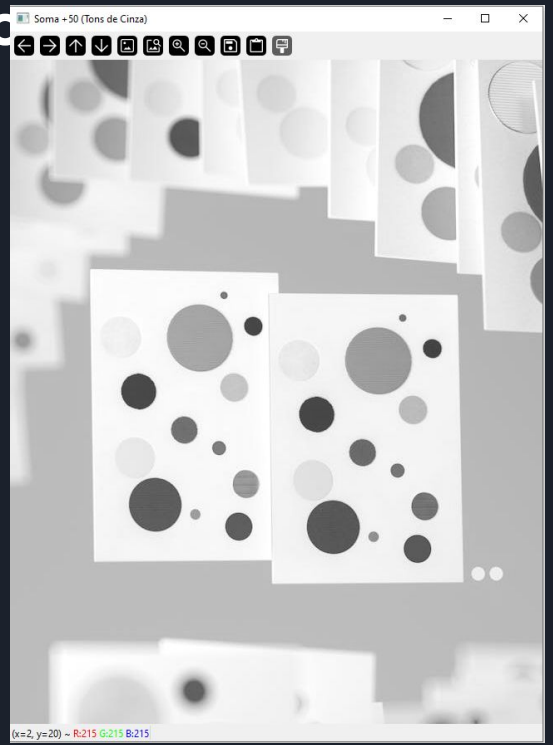
Operações Aritméticas (+ - * /) - Imagem com Valor Escalar

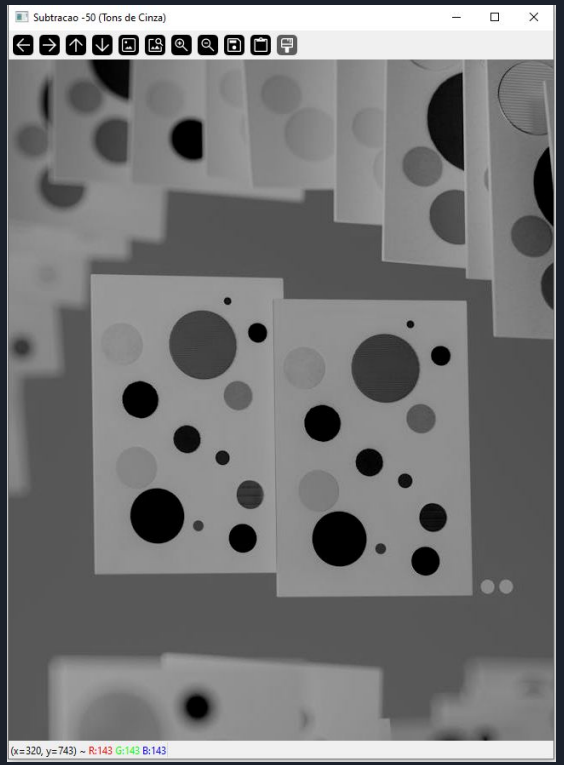




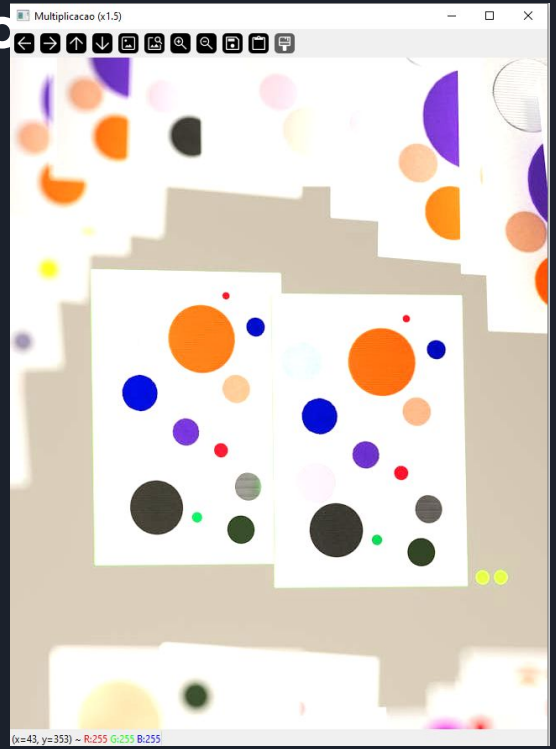
Operações Aritméticas (+ - * /) - Imagem com Valor Escalar



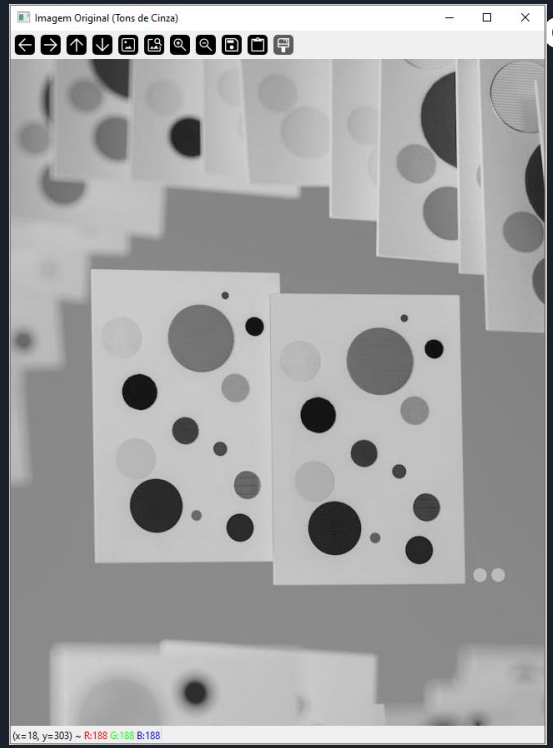


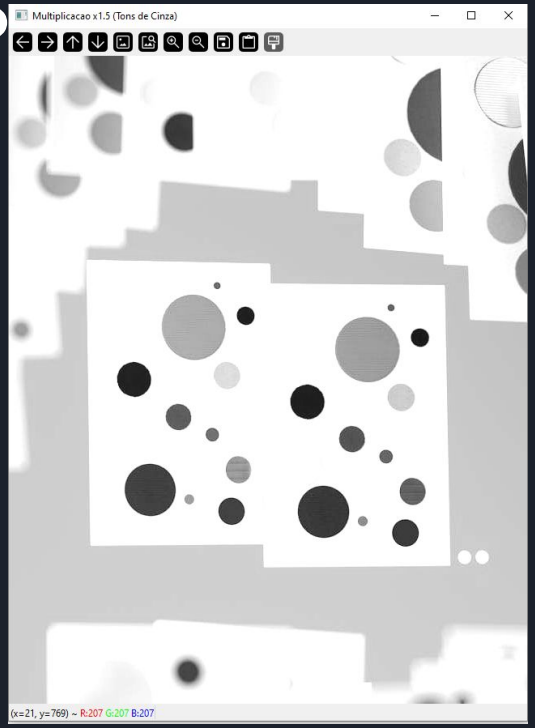


Operações Aritméticas (+ - * /) - Imagem com Valor Escalar



Operações Aritméticas (+ - * /) - Imagem com Valor Escalar





Operações Aritméticas (+ - * /) - Imagem com Imagem
Código

```
// Verificar se as imagens tem as mesmas dimensoes
if (image1.size() != image2.size()) {
    std::cout << "Redimensionando a segunda imagem para coincidir com a primeira..." << std::endl;
    resizeImage(image2, image2, image1.size());
}
```

```
// Percorrer pixel a pixel
for (int y = 0; y < image1.rows; y++) {
    for (int x = 0; x < image1.cols; x++) {
        // Obter os valores BGR dos pixels das duas imagens
        cv::Vec3b pixel1 = image1.at<cv::Vec3b>(y, x);
        cv::Vec3b pixel2 = image2.at<cv::Vec3b>(y, x);

        // SOMA (com promocao de tipos)
        cv::Vec3b somaPixel;
        somaPixel[0] = saturate_uchar(static_cast<int>(pixel1[0]) + static_cast<int>(pixel2[0]));
        somaPixel[1] = saturate_uchar(static_cast<int>(pixel1[1]) + static_cast<int>(pixel2[1]));
        somaPixel[2] = saturate_uchar(static_cast<int>(pixel1[2]) + static_cast<int>(pixel2[2]));
        somaResult.at<cv::Vec3b>(y, x) = somaPixel;

        // SUBTRACAO (diferenca absoluta com promocao de tipos)
        cv::Vec3b subPixel;
        subPixel[0] = saturate_uchar(abs(static_cast<int>(pixel1[0]) - static_cast<int>(pixel2[0])));
        subPixel[1] = saturate_uchar(abs(static_cast<int>(pixel1[1]) - static_cast<int>(pixel2[1])));
        subPixel[2] = saturate_uchar(abs(static_cast<int>(pixel1[2]) - static_cast<int>(pixel2[2])));
        subtracaoResult.at<cv::Vec3b>(y, x) = subPixel;

        // MULTIPLICACAO (normalizada com float para evitar overflow)
        cv::Vec3b multPixel;
        multPixel[0] = saturate_uchar((static_cast<float>(pixel1[0]) * static_cast<float>(pixel2[0])) / 255.0f);
        multPixel[1] = saturate_uchar((static_cast<float>(pixel1[1]) * static_cast<float>(pixel2[1])) / 255.0f);
        multPixel[2] = saturate_uchar((static_cast<float>(pixel1[2]) * static_cast<float>(pixel2[2])) / 255.0f);
        multiplicacaoResult.at<cv::Vec3b>(y, x) = multPixel;

        // DIVISAO (com protecao contra divisao por zero e promocao de tipos)
        cv::Vec3b divPixel;
        divPixel[0] = pixel2[0] == 0 ? 0 : saturate_uchar((static_cast<float>(pixel1[0]) * 255.0f) / static_cast<float>(pixel2[0]));
        divPixel[1] = pixel2[1] == 0 ? 0 : saturate_uchar((static_cast<float>(pixel1[1]) * 255.0f) / static_cast<float>(pixel2[1]));
        divPixel[2] = pixel2[2] == 0 ? 0 : saturate_uchar((static_cast<float>(pixel1[2]) * 255.0f) / static_cast<float>(pixel2[2]));
        divisaoResult.at<cv::Vec3b>(y, x) = divPixel;
    }
}
```

Operações Aritméticas (+ - * /) - Imagem com Imagem
Resultado (Imagens Originais)

Operações Aritméticas (+ - * /) - Imagem com Imagem
Resultado (Imagens Originais)

Operações Aritméticas (+ - * /) - Imagem com Imagem
Resultado: Soma e Subtração

Operações Aritméticas (+ - * /) - Imagem com Imagem
Resultado: Soma e Subtração

Operações Aritméticas (+ - * /) - Imagem com Imagem
Resultado: Multiplicação e Divisão

Operações Aritméticas (+ - * /) - Imagem com Imagem
Resultado: Multiplicação e Divisão

Limiarização
Código

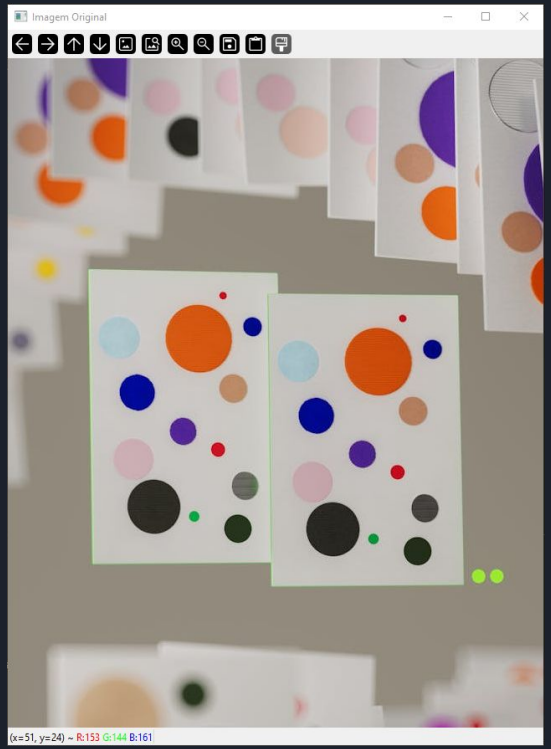
```
// 2. Definir valor de limiar (threshold)
int limiar = 128; // Valor tipico (0-255)
std::cout << "Valor de limiar utilizado: " << limiar << std::endl;
```

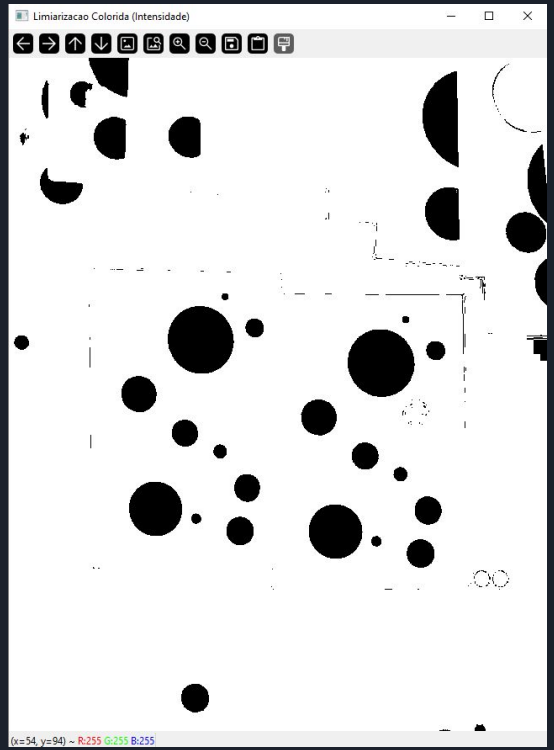
```
// 3. Converter para tons de cinza para limiarizacao em escala de cinza
cv::Mat grayImage;
convertBGRToGray(originalImage, grayImage);
```

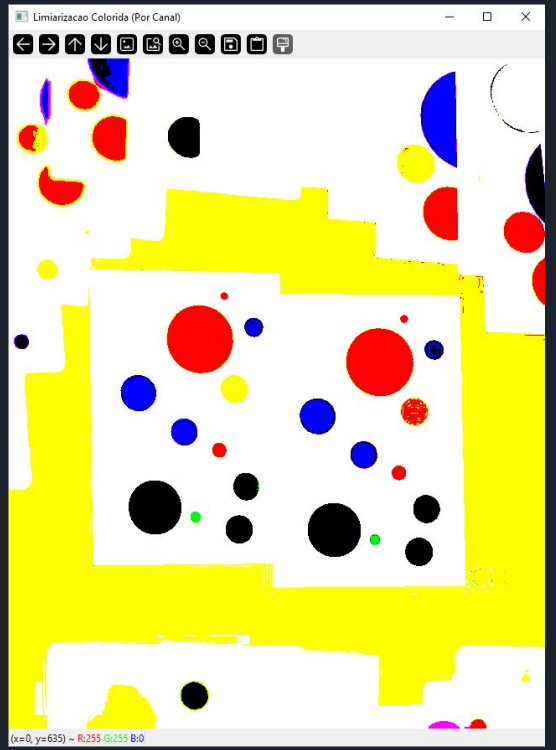
```
// 5. Limiarizacao em tons de cinza
for (int y = 0; y < grayImage.rows; y++) {
    for (int x = 0; x < grayImage.cols; x++) {
        uchar grayPixel = grayImage.at<uchar>(y, x);

        // Limiarizacao binaria: pixel >= limiar = 255, senao = 0
        if (grayPixel >= limiar) {
            binaryResult.at<uchar>(y, x) = 255;
            binaryInverseResult.at<uchar>(y, x) = 0;
        } else {
            binaryResult.at<uchar>(y, x) = 0;
            binaryInverseResult.at<uchar>(y, x) = 255;
        }
    }
}
```

```
// 6. Limiarizacao colorida (metodo 1: por intensidade media)
for (int y = 0; y < originalImage.rows; y++) {
    for (int x = 0; x < originalImage.cols; x++) {
        cv::Vec3b pixel = originalImage.at<cv::Vec3b>(y, x);

        // Calcular intensidade media do pixel
        int intensidadeMedia = static_cast<int>((pixel[0] + pixel[1] + pixel[2]) / 3);

        // Aplicar limiarizacao baseada na intensidade media
        if (intensidadeMedia >= limiar) {
            colorBinaryResult.at<cv::Vec3b>(y, x) = cv::Vec3b(255, 255, 255); // Branco
        } else {
            colorBinaryResult.at<cv::Vec3b>(y, x) = cv::Vec3b(0, 0, 0); // Preto
        }
    }
}
```

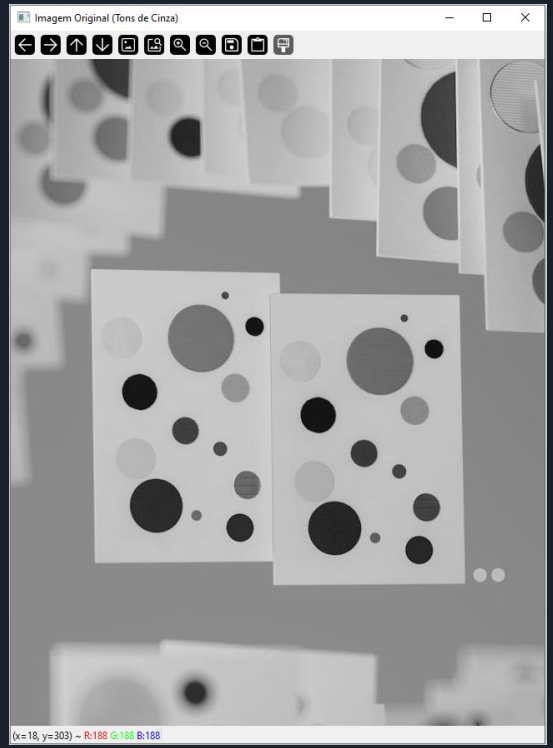
Limiarização
Exemplo Colorido

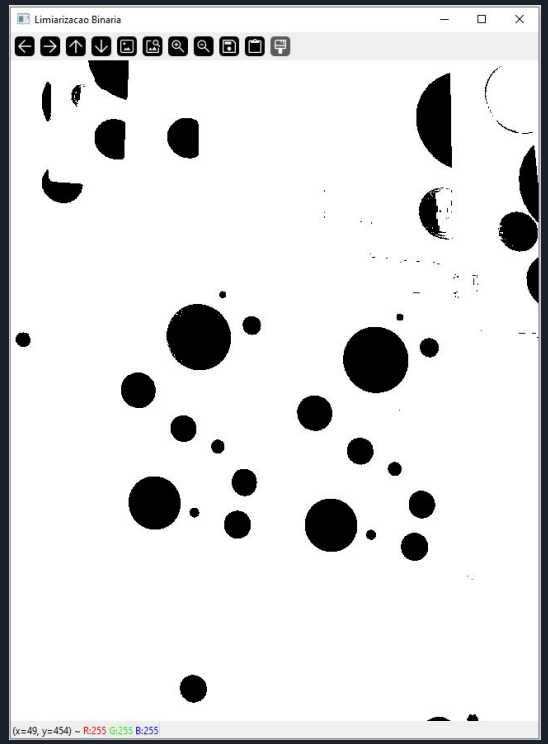


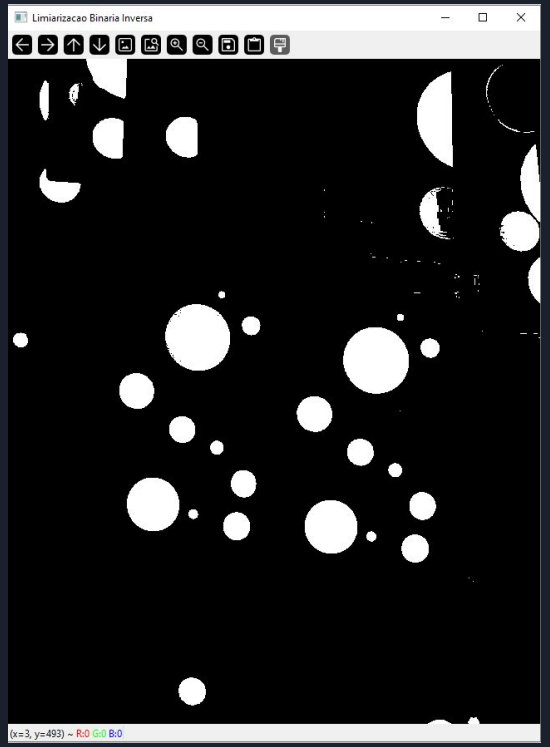




Limiarização
Exemplo Cinza







Histograma
Código

```cpp
// Funcao para desenhar histograma
cv::Mat drawHistogram(const std::vector<int>& histogram, const cv::Scalar& color, const std::string& title) {
    int hist_w = 512, hist_h = 400;
    int bin_w = cvRound((double) hist_w / 256);

    cv::Mat histImage(hist_h, hist_w, CV_8UC3, cv::Scalar(0, 0, 0));

    // Encontrar valor maximo para normalizar
    int max_val = *std::max_element(histogram.begin(), histogram.end());

    // Desenhar as barras do histograma
    for (int i = 1; i < 256; i++) {
        cv::line(histImage,
                 cv::Point(bin_w * (i-1), hist_h - cvRound(((double)histogram[i-1] / max_val) * hist_h)),
                 cv::Point(bin_w * i, hist_h - cvRound(((double)histogram[i] / max_val) * hist_h)),
                 color, 2, 8, 0);
    }

    // Adicionar titulo
    cv::putText(histImage, title, cv::Point(10, 30), cv::FONT_HERSHEY_SIMPLEX, 0.8, cv::Scalar(255, 255, 255), 2);

    return histImage;
}
```

```cpp
// Funcao para computar histograma manualmente
std::vector<int> computeHistogram(const cv::Mat& image, int channel = 0) {
    std::vector<int> histogram(256, 0);

    for (int y = 0; y < image.rows; y++) {
        for (int x = 0; x < image.cols; x++) {
            if (image.channels() == 1) {
                // Imagem em escala de cinza
                uchar intensity = image.at<uchar>(y, x);
                histogram[intensity]++;
            } else {
                // Imagem colorida
                cv::Vec3b pixel = image.at<cv::Vec3b>(y, x);
                histogram[pixel[channel]]++;
            }
        }
    }

    return histogram;
}
```

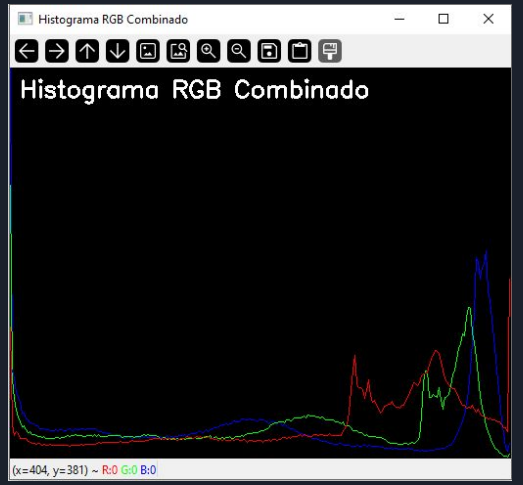
Histograma
Código

```
// Histograma da imagem em escala de cinza
std::vector<int> grayHistogram = computeHistogram(grayImage);

// Histogramas dos canais RGB da imagem colorida
std::vector<int> blueHistogram = computeHistogram(originalImage, 0);   // Canal B
std::vector<int> greenHistogram = computeHistogram(originalImage, 1);  // Canal G
std::vector<int> redHistogram = computeHistogram(originalImage, 2);    // Canal R

std::cout << "Histogramas computados com sucesso!" << std::endl;

// 4. Criar visualizacoes dos histogramas
cv::Mat grayHistImage = drawHistogram(grayHistogram, cv::Scalar(255, 255, 255), "Histograma - Escala de Cinza");
cv::Mat blueHistImage = drawHistogram(blueHistogram, cv::Scalar(255, 0, 0), "Histograma - Canal Azul");
cv::Mat greenHistImage = drawHistogram(greenHistogram, cv::Scalar(0, 255, 0), "Histograma - Canal Verde");
cv::Mat redHistImage = drawHistogram(redHistogram, cv::Scalar(0, 0, 255), "Histograma - Canal Vermelho");
```

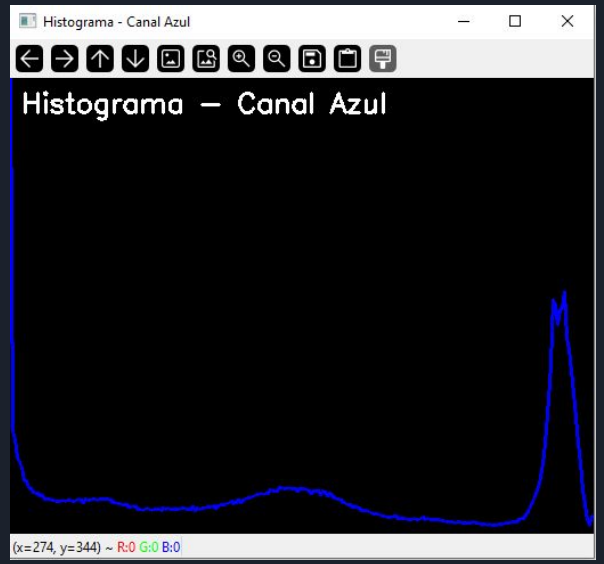
Histograma
Código

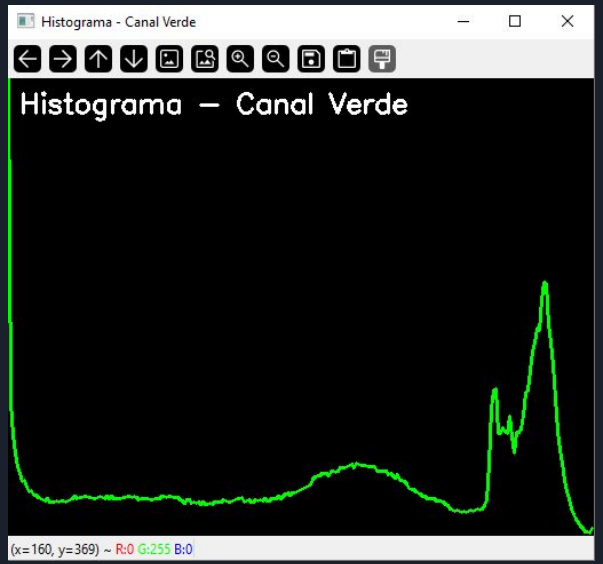
```
// 5. Criar histograma combinado RGB
cv::Mat combinedHistImage(400, 512, CV_8UC3, cv::Scalar(0, 0, 0));
int bin_w = cvRound((double) 512 / 256);

// Encontrar valor maximo para normalizar
int max_blue = *std::max_element(blueHistogram.begin(), blueHistogram.end());
int max_green = *std::max_element(greenHistogram.begin(), greenHistogram.end());
int max_red = *std::max_element(redHistogram.begin(), redHistogram.end());
int max_combined = std::max({max_blue, max_green, max_red});

// Desenhar histogramas combinados
for (int i = 1; i < 256; i++) {
    // Canal Azul
    cv::line(combinedHistImage,
            cv::Point(bin_w * (i-1), 400 - cvRound(((double)blueHistogram[i-1] / max_combined) * 400)),
            cv::Point(bin_w * i, 400 - cvRound(((double)blueHistogram[i] / max_combined) * 400)),
            cv::Scalar(255, 0, 0), 1, 8, 0);

    // Canal Verde
    cv::line(combinedHistImage,
            cv::Point(bin_w * (i-1), 400 - cvRound(((double)greenHistogram[i-1] / max_combined) * 400)),
            cv::Point(bin_w * i, 400 - cvRound(((double)greenHistogram[i] / max_combined) * 400)),
            cv::Scalar(0, 255, 0), 1, 8, 0);

    // Canal Vermelho
    cv::line(combinedHistImage,
            cv::Point(bin_w * (i-1), 400 - cvRound(((double)redHistogram[i-1] / max_combined) * 400)),
            cv::Point(bin_w * i, 400 - cvRound(((double)redHistogram[i] / max_combined) * 400)),
            cv::Scalar(0, 0, 255), 1, 8, 0);
}

cv::putText(combinedHistImage, "Histograma RGB Combinado", cv::Point(10, 30), cv::FONT_HERSHEY_SIMPLEX, 0.8, cv::Scalar(255, 255, 255), 2);

std::cout << "Visualizacoes dos histogramas criadas!" << std::endl;
```

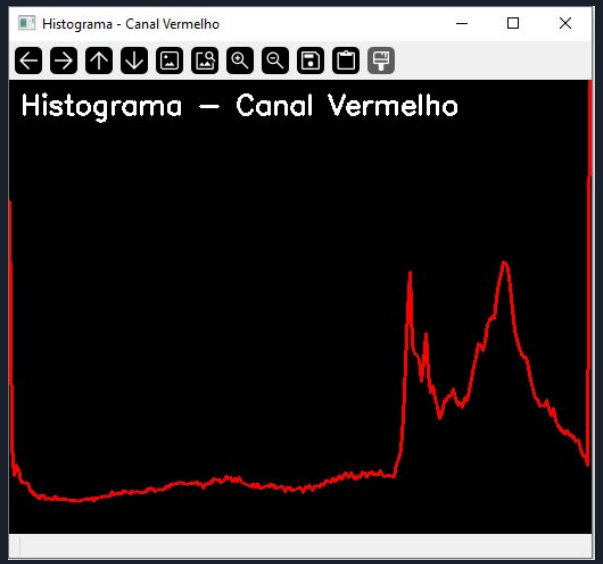
Histograma
Exemplo



Histograma
Exemplo



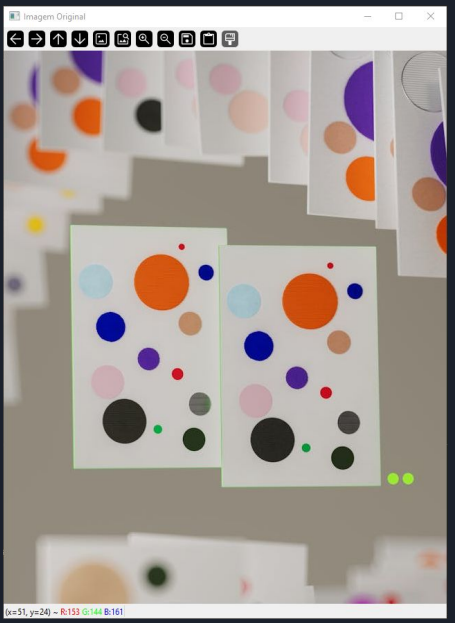


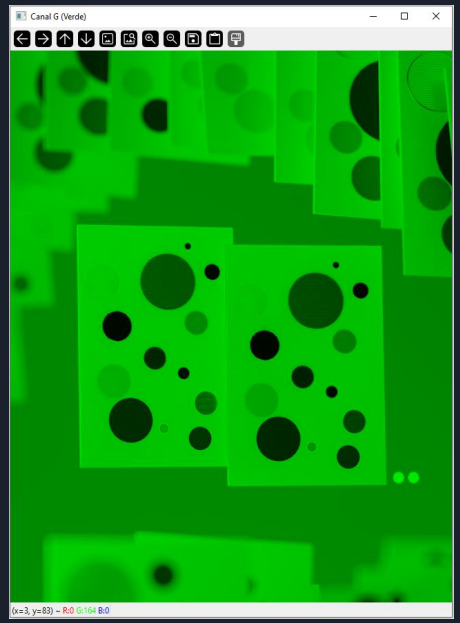


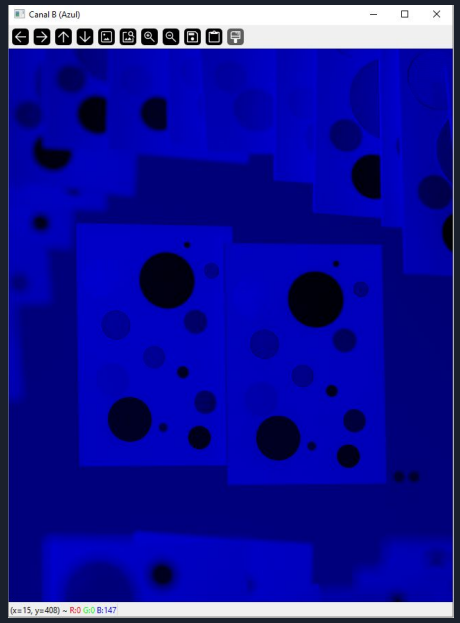
Isolar Canais de Cores
Código

```
// 4. Processar cada pixel para isolar os canais
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        // OpenCV usa formato BGR (Blue, Green, Red)
        cv::Vec3b pixel = originalImage.at<cv::Vec3b>(y, x);

        uchar blue = pixel[0];   // Canal B
        uchar green = pixel[1];  // Canal G
        uchar red = pixel[2];    // Canal R

        // Canal R isolado: apenas o vermelho, outros zerados
        redChannel.at<cv::Vec3b>(y, x) = cv::Vec3b(0, 0, red);

        // Canal G isolado: apenas o verde, outros zerados
        greenChannel.at<cv::Vec3b>(y, x) = cv::Vec3b(0, green, 0);

        // Canal B isolado: apenas o azul, outros zerados
        blueChannel.at<cv::Vec3b>(y, x) = cv::Vec3b(blue, 0, 0);
    }
}
```

Isolar Canais de Cores
Exemplo





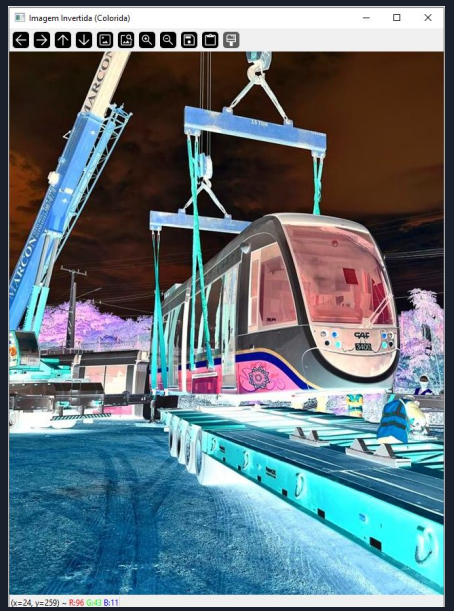


Inverso da imagem
Código

```
// 3. Criar matriz para a imagem invertida
cv::Mat invertedImage = cv::Mat::zeros(height, width, originalImage.type());

std::cout << "Processando pixels para inverter a imagem..." << std::endl;
```

```
// 4. Processar cada pixel para fazer o inverso (negativo)
if (channels == 3) {
    // Imagem colorida (BGR)
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            cv::Vec3b pixel = originalImage.at<cv::Vec3b>(y, x);

            // Inverso: novo_valor = 255 - valor_original
            uchar blue = 255 - pixel[0];   // Inverter canal B
            uchar green = 255 - pixel[1];  // Inverter canal G
            uchar red = 255 - pixel[2];    // Inverter canal R

            invertedImage.at<cv::Vec3b>(y, x) = cv::Vec3b(blue, green, red);
        }
    }
} else if (channels == 1) {
    // Imagem em escala de cinza
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            uchar pixel = originalImage.at<uchar>(y, x);

            // Inverso: novo_valor = 255 - valor_original
            uchar invertedPixel = 255 - pixel;

            invertedImage.at<uchar>(y, x) = invertedPixel;
        }
    }
}
```

Inverso da imagem
Exemplo



Link para o código:
https://github.com/GuiSchveitzer/Trabalho_Processamento_de_imagens_M1PT1/tree/main