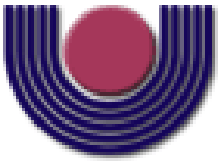


UNIOESTE – Universidade Estadual do Oeste do Paraná

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Aplicação de métodos de tratamento de cores em ampliadores de tela

Alexandre Santetti Scortegagna

CASCAVEL

2013

Alexandre Santetti Scortegagna

APLICAÇÃO DE MÉTODOS DE TRATAMENTO DE CORES EM AMPLIADORES DE TELA

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Marcio Oyamada

CASCAVEL

2013

ALEXANDRE SANTETTI SCORTEGAGNA

APLICAÇÃO DE MÉTODOS DE TRATAMENTO DE CORES EM AMPLIADORES DE TELA

Monografia apresentada como requisito parcial para obtenção do Título de *Bacharel em Ciência da Computação*, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Marcio Oyamada (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Adair Santa Catarina
Colegiado de Ciência da Computação,
UNIOESTE

Profª. Adriana Postal
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 6 de Novembro de 2013.

DEDICATÓRIA

Dedico este trabalho aos meus pais, Armando e Elizabete e às minhas duas irmãs, Daiana e Deise, que estiveram sempre ao meu lado, apoiando-me em todos os momentos de minha vida.

AGRADECIMENTOS

Aos meus pais, Armando Scortegagna e Elizabete Santetti Scortegagna, por todo amor, atenção e cuidado que sempre recebi, e pela preocupação com meu ensino.

Às minha irmãs, Daiana Santetti Scortegagna e Deise Santetti Scortegagna, que sempre me deram força e apoio.

Ao meu orientador, Prof. Márcio Oyamada, pela atenção e ajuda no desenvolvimento deste trabalho, assim como por todo conhecimento passado como professor.

Ao prof. Jorge Bidarra, pela oportunidade de ter participado do projeto xLupa, o qual possibilitou a realização deste trabalho.

A todos os professores que compartilharam conhecimento durante toda a minha vida.

Aos meus amigos, os quais sempre tiveram considerável importância em minha vida.

E finalmente a todas as pessoas que tentam todos os dias fazer do mundo um lugar melhor para se viver.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

RGB	*Red, Green, Blue*
CMYK	*Cyan, Magenta, Yellow, Black*
SSE	*Streaming SIMD Extensions*
SIMD	*Single Instruction, Multiple Data*

SUMÁRIO

RESUMO

Este trabalho descreve o estudo de métodos de tratamento de imagem no xLupa, um ampliador de tela aberto voltado para pessoas com baixa visão e idosas. O presente trabalho tem como foco o problema de ocultação de detalhes da imagem quando o usuário habilita o recurso de alteração de cores de fonte e de fundo, causando perda de informação visual. A solução adotada para minimizar a perda de informação visual foi utilizar um método similar a conversão em escala de cinza. Este método foi implementado e avaliado comparando-o com os resultados gerados pela versão anterior. Adicionalmente, foi realizado a otimização do código visando obter o menor impacto de desempenho possível para o novo método.

Palavras-chave: Tecnologia Assistiva; Ampliador de tela; Software Livre; Processamento de Imagens.

Capítulo 1

Introdução

Ampliadores de telas são tecnologias assistivas importantes para pessoas com baixa visão, mas também para pessoas idosas acometidas pela presbiopia.

De acordo com o Instituto Brasileiro de Geográfia e Estatística IBGE (2010) [1], existem no Brasil 6.585.308 pessoas com deficiência visual. Deste total, 582.624 possuem cegueira e o restante possuem baixa visão. A pesquisa revela ainda que 23% da população brasileira tem algum tipo de deficiência.

O xLupa [2] é um software livre desenvolvido para ampliação de telas em computadores, desenvolvido principalmente em C++ [3] voltado para pessoas com baixa visão, enquadrando-se na categoria de ampliadores de tela. O xLupa possui como características principais a manipulação de imagens, dentre os quais se incluem, além da ampliação, definições de contraste, ajuste de brilho, seleção de cores de fundo, cores de fonte e intensidade de cores. O xLupa também possui um leitor de tela integrado.

1.1 Identificação do problema

Dentre os itens que necessitavam de revisão e adequação na versão atual do xLupa, encontra-se o tratamento da qualidade do contraste aplicado às imagens ampliadas, especialmente em imagens ampliadas em duas cores (cor de fundo e fonte). A aplicação deste contraste é importante para usuários que possuem deficiências em relação à percepção das cores, e que tem mais facilidade com certas cores do que outras, permitindo um ajuste de acordo com suas necessidades.

As imperfeições presentes no xLupa eram geradas quando o usuário selecionava uma cor para fonte e outra para fundo, e como as cores selecionadas nesta versão não apresentavam variação de tons, todo pixel na tela será da cor 1 (fonte) ou cor 2 (fundo), ou

seja, uma região de intensidade de cor intermediária era oculta por pertencer à um dos dois lados do limiar utilizado.

Quando o xLupa é ativado, cabe ao usuário escolher o fator de ampliação e a cor de fundo que melhor se adapte a sua necessidade. A escolha da cor de fundo é uma das particularidades do xLupa, pois com isso o mesmo pode adaptar-se a diferentes usuários que possuem diferentes necessidades de contraste.

Diferentes fatores de ampliação não afetam o resultado do tratamento de cores, e o cursor do mouse não é afetado nem pelo fator de ampliação, nem pelo tratamento de cores sendo assim, o cursor do mouse será o único elemento visual que poderá apresentar cores que não pertençam à combinação selecionada pelo usuário.

As combinações de cores definidas inicialmente são limitadas, mas durante a execução do xLupa pode-se selecionar qualquer combinação de cores que se queira, bem como alterar o fator de ampliação para um valor não disponível no menu mostrado durante a inicialização do xLupa.

Nem sempre as cores escolhidas produzem um bom contraste. Muitas vezes, a combinação das cores de fundo e da fonte gera imagens pouco nítidas. Menus, botões e alguns textos nas janelas podem apresentar algumas imperfeições gráficas, tornando-se difíceis de serem visualizados prejudicando o trabalho dos usuários.

1.2 Motivações

Como pessoas com baixa visão podem apresentar diferentes reações em relação a determinadas combinações de cores e efeitos de suavização ou realce, busca-se aqui o estudo e aplicação de técnicas que possam atender as necessidades desses usuários, incluindo no xLupa aquelas que apresentarem resultados satisfatórios.

Tendo em vista as possibilidades de aprimoramentos que o tratamento de cores do xLupa ainda pode receber, estudaremos a utilização de algoritmos de processamento de imagens digitais no tratamento de cores do xLupa, buscando um resultado superior à versão atual em termos de qualidade de imagem, baseado na conservação dos detalhes da imagem original quando se aplica o tratamento de cores.

O algoritmo de tratamento de cores do xLupa deixa espaço para a implementação e teste de várias soluções, em especial a quantização do espaço de cores, sendo este o foco deste trabalho.

1.3 Objetivos

O objetivo deste trabalho é dar continuidade ao desenvolvimento do xLupa [1], mais especificamente em relação à melhoria na qualidade da imagem. Para esta etapa do desenvolvimento, novos algoritmos de tratamento de imagem [4] tais como manipulação de cores e contraste foram estudados e/ou aplicados, buscando uma melhoria da qualidade da imagem ampliada, especialmente na preservação da informação visual.

O foco é o aprimoramento da imagem gerada pelo xLupa quando o usuário seleciona combinações de cores para frente e fundo. Esse recurso é disponível no xLupa pois usuários podem ter mais facilidade com certas combinações de cores do que outras. No entanto o algoritmo original apresenta deficiências no resultado visual das combinações.

Desta forma, o objetivo deste trabalho é avaliar e implementar outros métodos de tratamento de cores visando reduzir a perda de detalhes. Adicionalmente, este trabalho tem como objetivo a análise do impacto no desempenho e a otimização visando minimizar o impacto do tratamento de cores.

1.4 Organização do texto

Neste capítulo foram apresentadas informações básicas sobre o xLupa, uma descrição dos problemas encontrados em sua versão atual, as motivações para a continuidade do desenvolvimento do xLupa, bem como os objetivos deste trabalho.

No capítulo 2 apresentamos uma análise subjetiva do resultado gerado pela versão atual do software xLupa, considerando os detalhes da imagem que são ou não omitidos, descrevendo detalhadamente os problemas encontrados em relação ao tratamento de cores.

Ainda no capítulo 2, com o conhecimento prévio das deficiências do xLupa, apresentaremos a solução para o problema do tratamento de cores, exibindo as alterações no código e a análise subjetiva do resultado gerado pela aplicação da solução escolhida.

O capítulo 3 aborda o desempenho da solução para o problema das cores, apresentando otimizações no código e comparando o desempenho entre a versão original do xLupa e as versões geradas pela aplicação de métodos de otimização, e no capítulo 4 temos a conclusão.

Capítulo 2

Estudo e Implementação

2.1 Representação de imagens

O xLupa manipula imagens em representação matricial [4], onde cada elemento da matriz é denominado "pixel" (*picture element*). Esta matriz de pixels é denominada de "*bit-map*" [4] ou "Mapa de Bits".

Este mapa de bits é um reticulado onde cada pixel possui uma cor associada. Uma determinada imagem possuirá também uma "resolução" associada, caracterizada pelo número de pixels na horizontal e na vertical. Cada elemento da imagem possuirá uma localização, que é definida pela suas coordenadas.

No xLupa a captura desta matriz de pixels é feita através da biblioteca GMAG [5]. Os dados capturados estarão disponíveis para manipulação e cada pixel pode ser acessado e modificado.

A matriz de pixels é capturada e processada. Os processos aplicados manipulam a forma e cor de cada pixel, e em seguida a tela toda é substituída pelos valores dos pixels tratados pelo xLupa.

2.2 Espaço de cores

Dentre os modelos de cores [4] mais conhecidos, estão o sistema RGB e CMYK [4] que consistem na combinação de canais de cores sendo eles o vermelho, verde e azul para RGB (*red, green, blue*) e ciano, magenta, amarelo e preto (*cyan, magenta, yellow, black*) para o sistema CMYK.

Cada pixel capturado durante o processo de tratamento de imagem do xLupa tem uma cor associada à ele, sendo esta cor formada por uma tupla de valores que representam os seus canais, sendo neste caso o modelo RGB.

O modelo de cores RGB é provavelmente o mais usado entre os modelos de cores, especialmente para dados de 8 bits. A teoria do espaço RGB (vermelho-verde-azul), de Thomas Young (1773-1829) [4], é baseada no princípio de que diversos efeitos cromáticos são obtidos pela projeção da luz branca através dos filtros vermelho, verde e azul.

No xLupa os valores dos canais RGB variam entre 0 e 255, e uma tupla destes valores representam uma cor. Quando o usuário seleciona uma combinação de duas cores, ele estará selecionando duas tuplas, e cada pixel na tela terá o valor de uma destas duplas, dependendo da intensidade luminosa do pixel estar ou não acima do limiar 127.

Esta intensidade luminosa é determinada pelo valor correspondente do pixel na escala de cinza, e é calculada para todo pixel toda vez que o método de tratamento de cores é utilizado pelo xLupa, através da equação 2.1, onde K é a intensidade do pixel e R, G e B são os valores de tupla da cor associada ao pixel.

$$K = 0.299*R + 0.587*G + 0.114*B \quad \text{(Equação 2.1)}$$

2.3 Algoritmos de manipulação de cores

Para o desenvolvimento da solução para os problemas encontrados no xLupa estudamos a conversão de cores para a escala de cinza e o modelo de cores RGB, criando um método que mistura os dois conhecimentos.

O cálculo da intensidade do pixel (equação 2.1) teve papel fundamental na solução para o problema das cores, pois foi mesclado com o algoritmo original do xLupa. As intensidades dos pixels são utilizadas para criar uma escala de cores a partir daquelas selecionadas pelo usuário. Estes resultados foram obtidos durante o projeto de iniciação científica e apresentados no EAIC 2012 [6].

Na solução original do xLupa quando o usuário seleciona azul para fonte e amarelo para fundo no xLupa, o resultado é uma tela onde cada pixel assume a cor azul ou amarelo. Na solução baseada na escala de cinza, o pixel assume um tom de azul ou de amarelo proporcional à intensidade do pixel, como na figura 2.1, que mostra um desenho simples em sua cor original e em combinação de azul e amarelo.

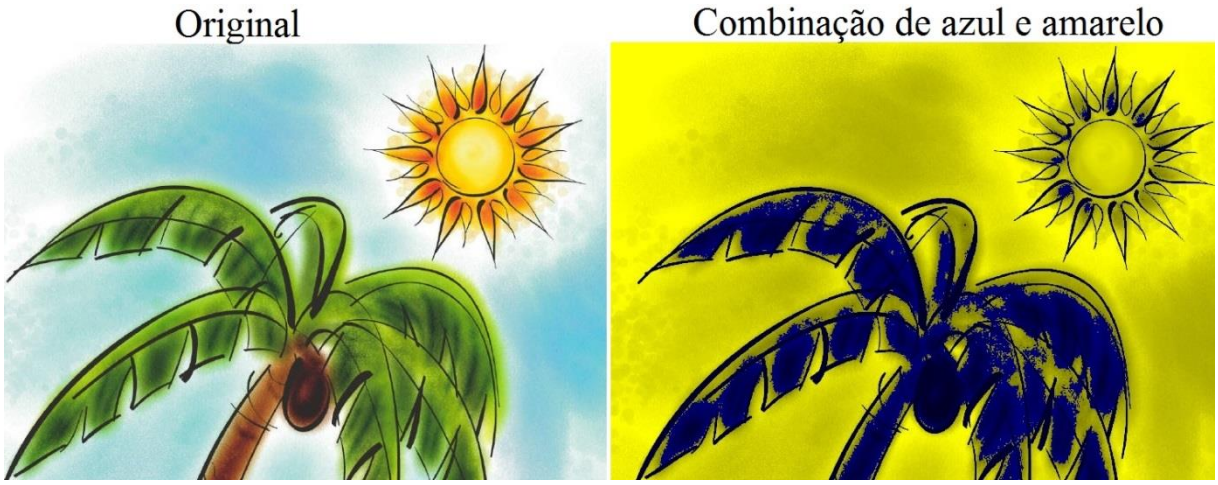


Figura 2.1: Exemplo de combinação de cores

2.4 Análise da imagem gerada pelo xLupa

Primeiramente foram levantadas as deficiências encontradas no xLupa, o que nos indicou as pesquisas por algoritmos de processamento de imagens digitais e computação gráfica [7], que nos levaram às soluções para a correção dos problemas evidenciados.

Depois da etapa de pesquisa aplicamos essas técnicas, e a seguir comparamos os resultados obtidos por estas técnicas com a versão original. A avaliação foi baseada na análise subjetiva da imagem gerada. A avaliação dos resultados obtidos foi feita apenas pela equipe envolvida no projeto, deixando os testes com deficientes visuais para trabalhos futuros.

Esta análise subjetiva leva em consideração os detalhes da imagem original que são perdidos ao se aplicar o tratamento de cores, dando preferência ao método onde as perdas são menores em relação à imagem original.

Ao iniciar o xLupa o usuário seleciona um fator de ampliação e uma combinação de cores (cor de fundo e cor de fonte), e então a imagem na tela será exibida com o fator de ampliação selecionado, exibindo somente as cores selecionadas pelo usuário, sendo que a cor de fundo corresponde aos pixel de maior luminosidade, e cor de fonte aos pixels de menor luminosidade, baseados em um limiar.

A figura 2.1 representa o desktop do Ubuntu [8], sem alterações provenientes do xLupa ou qualquer outro software.

Figura 2.2: Desktop do Ubuntu

Agora, utilizamos esta mesma imagem, mas aplicando a combinação de cores verde para fundo e vermelho para fonte, com o algoritmo de tratamento de cores original do xLupa, e o resultado é mostrado na figura 2.2

Figura 2.3: Desktop do Ubuntu com aplicação de cores

Pode-se observar na figura 2.2 que vários detalhes da imagem são perdidos. Neste caso todos os pixels com intensidade luminosa abaixo de 127 assumem a cor verde, e os restantes assumem a cor vermelha, a perda de detalhes é causada por esta escolha binária de cores feita pelo algoritmo de tratamento de imagem utilizado no xLupa.

Neste caso os pixels com intensidade luminosa acima de 127 receberam a cor 1, e os pixels com intensidade luminosa de 127 abaixo receberam a cor 2, sendo estas as duas cores selecionados pelo usuário, e a aplicação delas é feita através do código em C++ [9] apresentado no algoritmo 1.

```
1        if(k > 127){
2            if(cor_rgb1!=NULL){
3               rgb[0]=cor_rgb1[0];
4               rgb[1]=cor_rgb1[1];
5               rgb[2]=cor_rgb1[2];
6            }
7         }else {
8            if(cor_rgb2!=NULL){
9               cont++;
10               rgb[0] = cor_rgb2[0];
11               rgb[1] = cor_rgb2[1];
12               rgb[2] = cor_rgb2[2];
13             }
14          }
15         PUT_PIXEL24 (subimage_data, lx, ly, source_rowstride, pixel);
```

Algoritmo 1: Código original do xLupa para mudança de cores

Podemos observar que as duas cores tem seus canais RGB fixos, e portanto, não há variação de tons.

2.5 Quantização do espaço de cores

A quantização de cores consiste em converter uma escala de cores para outra, de acordo com os valores associados à cada escala. Por exemplo, uma cor representada em um espaço de cores de 16 bits terá seu valor equivalente em um espaço de cores de 8 bits. É uma conversão de escala, embora nem sempre seja uma transformação linear.

Para tratar e corrigir os problemas citados, estudamos e implementamos novos algoritmos de tratamento de imagens, as quais foram agregadas ao xLupa. Inicialmente o estudo se concentra na melhoria da quantização do espaço de cores para que as combinações selecionadas pelo usuário não gerem perda de informação visual.

Uma das soluções trabalhada foi a quantização do espaço de cores [4], tendo como resultado a divisão das cores em tons, de acordo com a intensidade das cores originais, sendo esse resultado ilustrado na figura 2.3.

Figura 2.4: Desktop do Ubuntu com a solução proposta

Observando a figura 2.3 nota-se que mesmo utilizando somente o verde e o vermelho, com os valores [0, 255, 0] e [255, 0, 0] respectivamente, a imagem conserva mais dos seus detalhes se comparada à figura 2.2, que utilizava o algoritmo original do xLupa. Isto ocorre porque cada pixel da tela assumirá o valor [0, i, 0] ou [j, 0, 0], com i variando de 0 a 127 e j variando de 128 a 255, de acordo com o valor da luminosidade do pixel.

Este resultado foi obtido através da alteração do método do tratamento de cores original do xLupa, substituindo a aplicação da cor selecionada pelo usuário por um tom da mesma cor, mas com tonalidade proporcional à luminosidade do pixel.

Desta forma as cores selecionadas pelo usuário serão quantizadas para um espaço de cores composto pelas diferentes intensidades ou tons da mesma cor, o que permite que regiões

de intensidade luminosa intermediária não sejam ocultas pela escolha binária feita pelo algoritmo original do xLupa.

A escala será limitada pela intensidade luminosa da cor selecionada pelo usuário, sendo este o limite superior, ou seja, a quantização vai gerar tons das cores selecionadas com intensidades luminosas variando entre 0 e a intensidade luminosa da cor selecionada.

Inicialmente calculamos a intensidade do pixel pela equação 2.1 (pág 6). Depois é feito um cálculo onde se aplica o valor de "k" para converter as cores selecionadas pelo usuário para um tom da mesma cor, com tonalidade proporcional à intensidade de cada pixel, conforme apresentado no algoritmo 2.

```
1   int k = (0.299*rgb[0] + 0.587*rgb[1] + 0.114*rgb[2]);
2   if(k > 127){
3       if(cor_rgb1!=NULL){
4          rgb[0]=(cor_rgb1[0]*k)/255;
5          rgb[1]=(cor_rgb1[1]*k)/255;
6          rgb[2]=(cor_rgb1[2]*k)/255;
7       }
8   }else {
9       if(cor_rgb2!=NULL){
10         cont++;
11         rgb[0] = (cor_rgb2[0]*k)/255;
12         rgb[1] = (cor_rgb2[1]*k)/255;
13         rgb[2] = (cor_rgb2[2]*k)/255;
14        }
15    }
16   PUT_PIXEL24 (subimage_data, lx, ly, source_rowstride, pixel);
```

Algoritmo 2: Versão modificada do tratamento de cores

O trecho de código do algoritmo 2 é utilizado para tratar cada pixel da matriz de pixels capturada pelo método de tratamento de imagem do xLupa, e é executado constantemente durante a utilização do xLupa, ou seja, a imagem sempre estará sendo atualizada enquanto o usuário utiliza o computador

2.6 Problemas remanescentes

Embora a solução adotada para o problema das cores resolva em boa parte a ocultação de detalhes da imagem, ela ainda não é perfeita e em alguns cenários o resultado pode ser aproximado do resultado gerado pela versão original do xLupa.

Estes problemas são observados quando se escolhe a cor preta ou outra muito escura, próxima do preto para fundo e/ou fonte pois, como foi explicado anteriormente a cor selecionada pelo usuário define o tom mais claro que a cor terá após a quantização, ou seja, cores muito escuras não terão uma variação de tons suficientemente grande para garantir uma boa visualização de detalhes da imagem.

A figura 2.4 mostra a imagem exibida na tela ao selecionar a cor verde para fundo e preto para fonte. Pode-se observar que a ocultação de detalhes ocorre nas áreas de cor preta do mesmo modo que ocorria no algoritmo original do xLupa, embora agora este problema seja limitado à cores muito escuras. A figura 2.5 mostra o zoom de uma região da tela onde esta perda de detalhes é mais evidente.

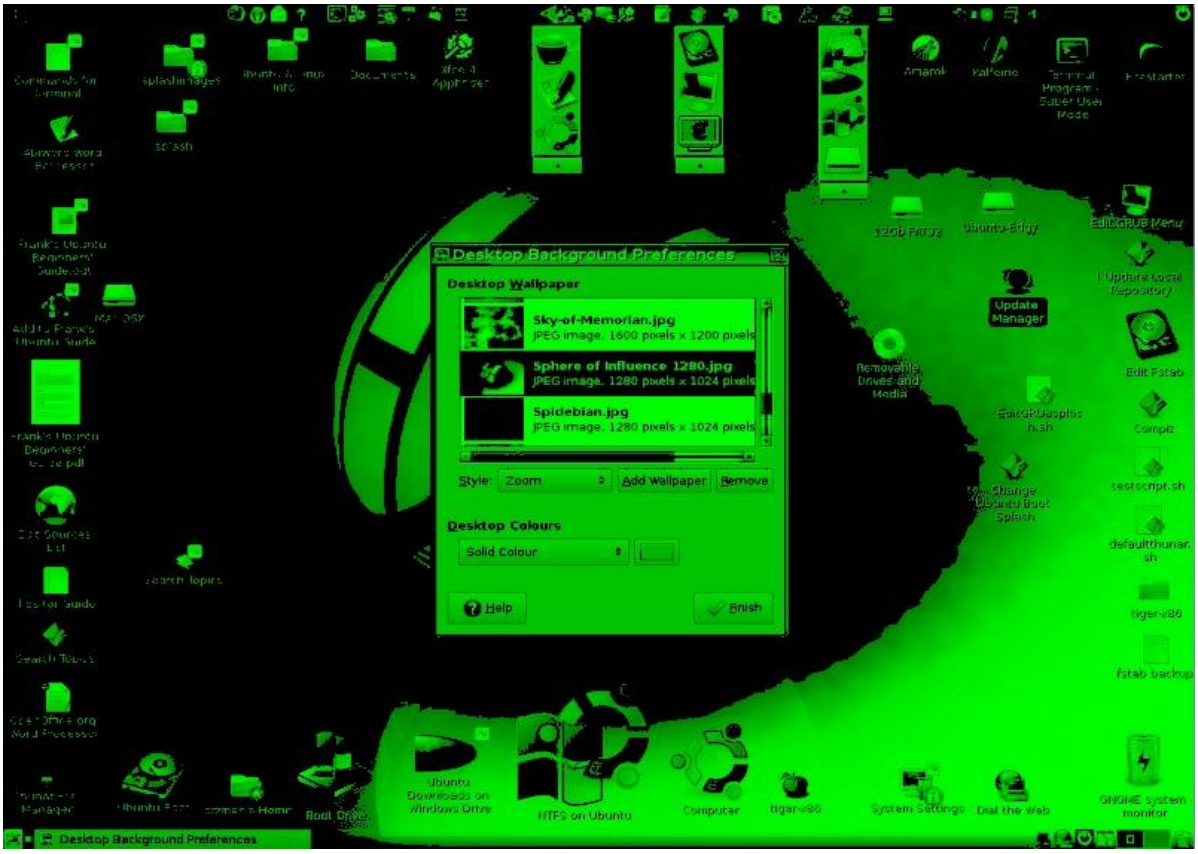


Figura 2.5: Desktop do Ubuntu com os problemas da solução proposta

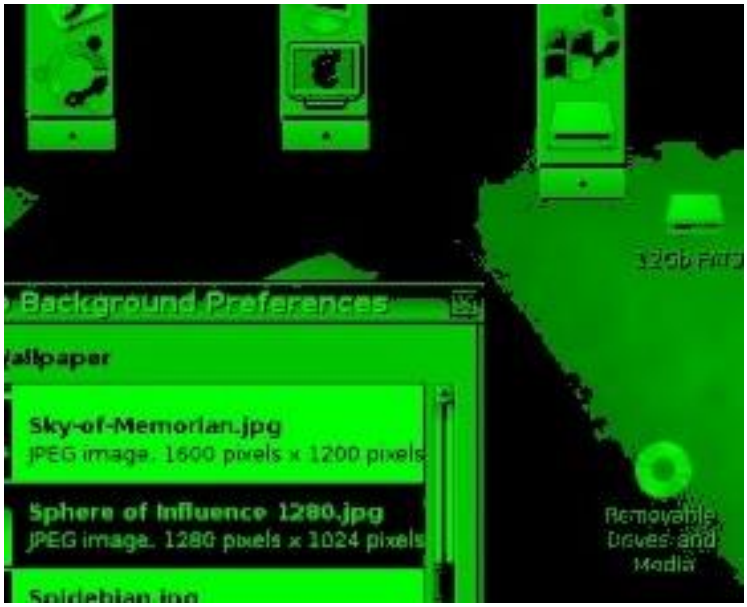


Figura 2.6: Região ampliada da tela

A solução para os problemas gerados nesta situação ainda está sendo estudada, e uma possibilidade seria uma verificação extra para definir se a cor selecionada pelo usuário é

muito escura e se sim, aumentar o valor de modo à permitir uma variação maior de tons, o que gera outros problemas.

Ao reajustar a cor selecionada pelo usuário, algumas cores ficariam descaracterizadas, como o preto, que passaria à ser cinza, e também a cor selecionada pelo usuário poderia não aparecer no tom mostrado durante a seleção em uma imagem muito clara, por exemplo.

A figura 2.6 mostra o resultado da solução proposta para o problema das cores muito escuras, e pode-se observar que embora os detalhes não sejam ocultos como no caso anterior, a cor preta fica descaracterizada, transformada em cinza. Utilizamos para o exemplo a mesma combinação de cores, verde para fundo e preto para fonte. A figura 2.7 mostra a mesma região ampliada da figura 2.5, com a solução proposta e a figura 2.8 compara as duas.

Figura 2.7: Desktop do Ubuntu com a correção da solução

Figura 2.8: Região ampliada da tela com a solução proposta

Figura 2.9: Comparação entre os dois resultados

A solução adotada é a modificação das cores encontradas na seleção inicial de cores do xLupa, com cores de fonte um pouco mais claras, e sendo assim cores muito escuras só podem ser selecionadas manualmente durante a utilização do xLupa, permitindo que os usuários mais básicos não tenham problemas com cores muito escuras.

Capítulo 3

Otimização do desempenho

Além do problema das cores descrito na seção anterior, o xLupa demanda um alto poder de processamento com a ampliação de tela. Como o processo de tratamento de cores é executado durante a utilização do xLupa, este deve ser otimizado para não inviabilizar o desempenho do xLupa.

Avaliamos o desempenho utilizando o terceiro fator de ampliação com cor de fundo verde e cor de fonte vermelha. Diferentes fatores de ampliação ou cores de fundo e fonte não apresentam qualquer impacto no desempenho, portanto esta comparação é suficiente.

A solução adotada para o problema das cores adiciona operações ao código, cálculos extras, e isso resultou em uma considerável redução no desempenho, representando uma diferença considerável na relação à carga total da CPU em um core i5 3300.

Sem habilitar o tratamento de cores o xLupa ocupa 43% da carga total da CPU. Na versão original a carga da CPU ficava em 57% e com o novo tratamento de cores esta carga subiu para 68%.

Este impacto na performance deve-se à utilização de divisões em ponto flutuante, que são custosas em termos de processamento. A figura 3.1 mostra a comparação da carga utilizada pela CPU entre a versão anterior do xLupa e a versão do xLupa com o novo tratamento de cores, porém sem a otimização.

Neste capítulo serão abordadas versões do xLupa, que são elas: 1 – A versão original, sem melhorias na qualidade da imagem em duas cores. 2 – A versão com qualidade visual melhorada e não-otimizada. 3 – A versão otimizada da versão 2. 4 – A versão otimizada com instruções SIMD. 5 – A versão final.

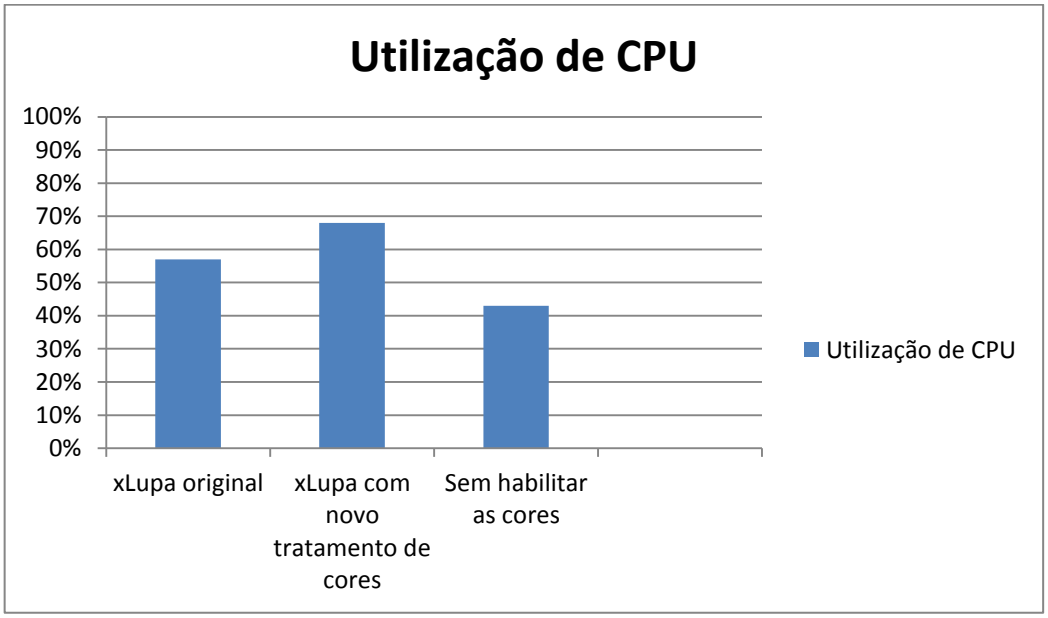


Figura 3.1: Comparação da utilização da CPU entre a versão original e a versão com o novo tratamento de cores

Como vemos na figura 3.1, a versão anterior do tratamento de cores do xLupa utiliza 57% da carga da CPU, enquanto a versão com o tratamento de cores utiliza 68%, o que é uma diferença significativa, representando 11% mais CPU do que a versão anterior.

3.1 Otimização do código

Inicialmente, removemos uma variável presente na versão original que não é mais utilizada, que é o cálculo da média entre os canais RGB do pixel. Esta média por sua vez era utilizada no lugar da intensidade do pixel, somente para comparar com o limiar, e seu valor era calculado da seguinte forma:

```
1        int media = (rgb[0]+rgb[1]+rgb[2])/3;
```

Após este processo passamos a trabalhar com valores inteiros apenas, com exceção do cálculo da luminosidade do pixel que é calculada em *float* e armazenada em inteiro (truncado), pois a qualidade gráfica não foi comprometida pela redução na precisão e a utilização de números em ponto flutuante sempre será mais custosa ao processador do que a utilização de inteiros.

A solução adotada para otimizar as divisões foi a utilização da instrução "*shift*" da linguagem Assembly [10], e esta instrução consiste no deslocamento dos bits, neste caso usamos "*shift right*", que é o deslocamento dos bits para a direita, que dá o mesmo resultado das divisões por potências de 2, de acordo com o tamanho do deslocamento.

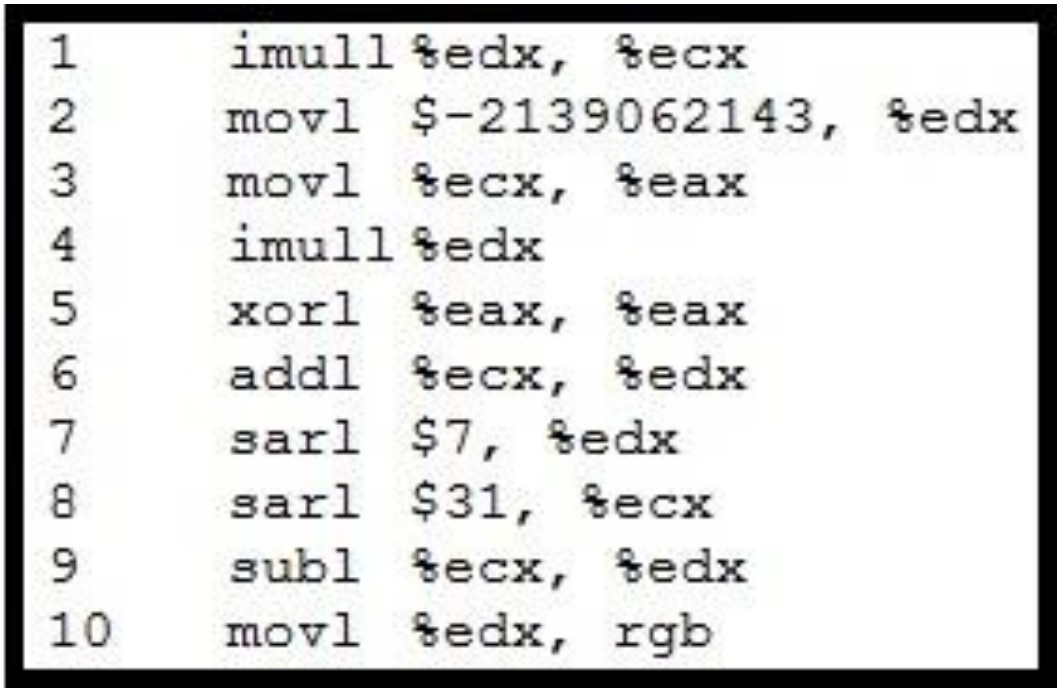
Um deslocamento para a direita representa uma divisão por 2, dois deslocamentos, por 4, três deslocamentos, por 8, e assim sucessivamente. Utilizamos 8 deslocamentos, ou seja, uma divisão por 256, que retorna um resultado um pouco diferente do anterior, que utilizava divisão por 255, porém a diferença na qualidade visual foi irrelevante (não percebida), enquanto o ganho em desempenho foi satisfatório, ficando próximo à versão original.

Na linguagem C++ as divisões por potências de 2 são automaticamente compiladas como instruções de shift direita, portanto não foi necessária a utilização de Assembler *inline* ou funções ASM, simplesmente substituímos as divisões por 255 por divisões por 256, ou seja dividindo por 2^8, conforme apresentado no algoritmo 3.

```
1   int k = (0.299*rgb[0] + 0.587*rgb[1] + 0.114*rgb[2]);
2   if(k > 127){
3       if(cor_rgb1!=NULL){
4           rgb[0]=(cor_rgb1[0]*k)/256;
5           rgb[1]=(cor_rgb1[1]*k)/256;
6           rgb[2]=(cor_rgb1[2]*k)/256;
7       }
8   }else {
9       if(cor_rgb2!=NULL){
10          cont++;
11          rgb[0] = (cor_rgb2[0]*k)/256;
12          rgb[1] = (cor_rgb2[1]*k)/256;
13          rgb[2] = (cor_rgb2[2]*k)/256;
14      }
15  }
16  PUT_PIXEL24 (subimage_data, lx, ly, source_rowstride, pixel);
```

Algoritmo 3: Versão otimizada do tratamento de cores

O algoritmo 4 mostra o código assembler gerado pelo método não otimizado, referente ao trecho de código que calcula os valores para o vetor rgb.



```
1     imull %edx, %ecx
2     movl  $-2139062143, %edx
3     movl  %ecx, %eax
4     imull %edx
5     xorl  %eax, %eax
6     addl  %ecx, %edx
7     sarl  $7, %edx
8     sarl  $31, %ecx
9     subl  %ecx, %edx
10    movl  %edx, rgb
```

Algoritmo 4: Código assembly do tratamento de cores não-otimizado

Na linha 1 a instrução imull multiplica o valor do canal R por k, que está armazenado em edx, com resultado sendo armazenado em ecx. Na linha 2 a instrução movl atribui à edx o valor -2139062143, que corresponde a -11111110111111101111111101111111 em binário, que será usado como uma máscara de bits. Na linha 3 a instrução movl copia o valor de ecx, que contém o valor da multiplicação de k pelo canal R, para eax.

Na linha 4 a instrução imull multiplica eax pelo valor de edx, que contém a máscara de bits, sendo o resultado armazenado em edx. Na linha 5 a instrução xorl zera o valor de eax. Na linha 6 a instrução addl soma o valor de ecx, que contém ainda o valor da multiplicação de k pelo canal R, à edx, que contém a multiplicação de k pelo canal R pela máscara de bits, armazenando o resultado em edx. Na linha 7 a instrução sarl desloca em 7 posições os bits de edx, o que equivale à uma divisão por 128.

Na linha 8 a instrução sarl desloca em 31 posições o valor de ecx, o que é equivalente à uma divisão por 2.147.483.648. Na linha 9 a instrução subl subtrai o valor de ecx de edx, que contém o resultado parcial da divisão por 255, resultando no valor final do canal R, e

armazena esse resultado em edx. Na linha 10 a instrução movl guarda o valor do canal R na primeira posição do vetor rgb.

Simplificando, a divisão de um número X qualquer por um número Y qualquer que não seja uma potência de 2, com n bits pode ser representada como na equação 3.1, onde SHR representa o deslocamento de bits para a direita, H representa o número de bits com o qual o divisor pode ser representado e MB representa a máscara de bits, que varia de acordo com o divisor e o número de bits utilizados.

$$X/Y = SHR\ H\ (X * MB + X)\ \ SHR\ N - 1(X) \qquad \text{(Equação 3.1)}$$

O algoritmo 5 mostra o código em assembler gerado pela divisão pelo mesmo trecho abordado no algoritmo 4.

```
1     imull vec, %edx
2     testl %edx, %edx
3     leal    (%edx), %eax
4     cmovs %eax, %edx
5     xorl  %eax, %eax
6     sarl  $8, %edx
7     movl  %edx, rgb
```

Algoritmo 5: Código assembly do tratamento de cores otimizado

Na linha 1 a instrução imull multiplica o valor de cor_rgb[1] por k, que está armazenado em edx, e o resultado é armazenado em edx. Na linha 2 a instrução testl verifica se o valor de edx é zero, como uma operação AND sobre os bits de edx. Na linha 3 a instrução leal atribui ao registrador eax o valor de edx.

Na linha 4 a instrução cmovs move o valor de eax para edx caso edx não seja zero, como verificado na linha 2. Na linha 5 a instrução xorl zera o valor do registrador eax.

Na linha 6 a instrução sarl desloca os bits do resultado da multiplicação de k por cor_rgb[1] em 8 posições para a direita, o que equivale à uma divisão por 256. Na linha 7 a instrução movl armazena o novo valor do canal R, que está em edx, na primeira posição do vetor rgb.

A principal diferença entre os algoritmos 4 e 5 é que no algoritmo 4 são necessárias mais operações, um sarl adicional e as operações extras envolvendo a máscara de bits, o que explica a diferença de desempenho entre as duas versões.

A figura 3.2 mostra a diferença de desempenho entre a versão do xLupa com tratamento de cores sem otimização, a versão otimizada e a versão original. Podemos notar que a versão otimizada diminui em 9% a carga total da CPU, ocupando 59%. sendo que a versão original ocupa 57%, ou seja, pouco impacto no desempenho sobre a versão original em troca de um grande aprimoramento na qualidade visual.

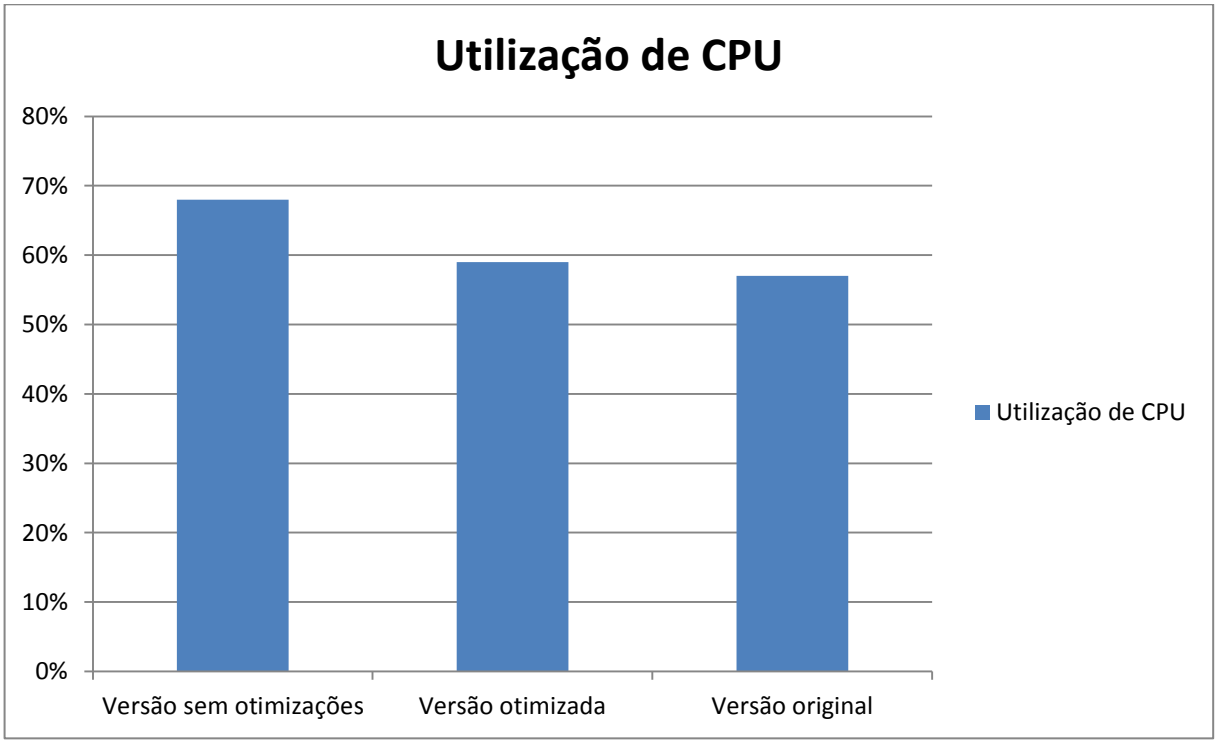


Figura 3.2: Comparação da utilização de CPU entre as versões com novo tratamento de cores não-otimizada, otimizada, e a versão original

Na figura 3.4 temos uma comparação do resultado visual gerado pela versão não-otimizada e pela versão otimizada e podemos notar uma qualidade gráfica semelhante, de uma pequena área da figura 3.3, encontrada logo abaixo.

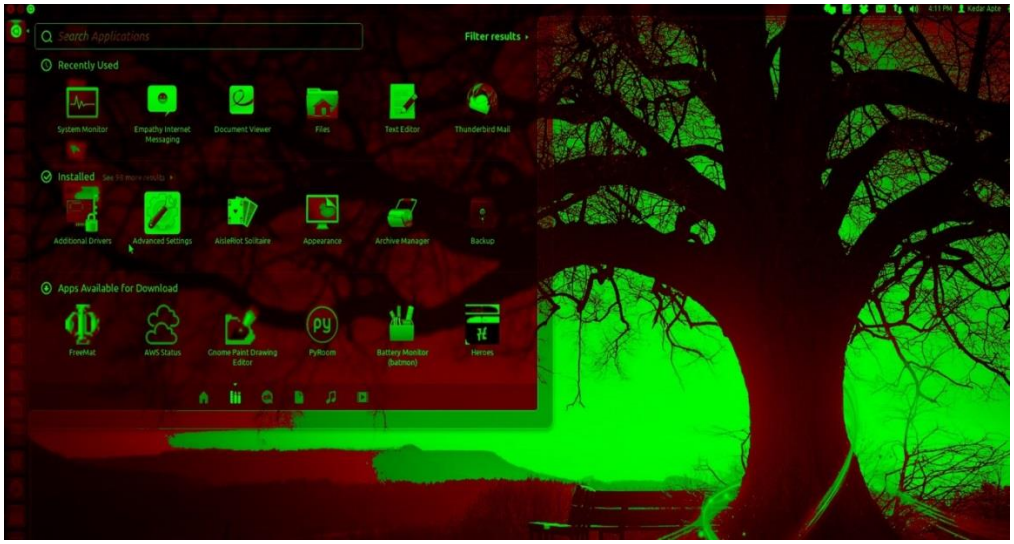


Figura 3.3: Desktop do Ubuntu (2)

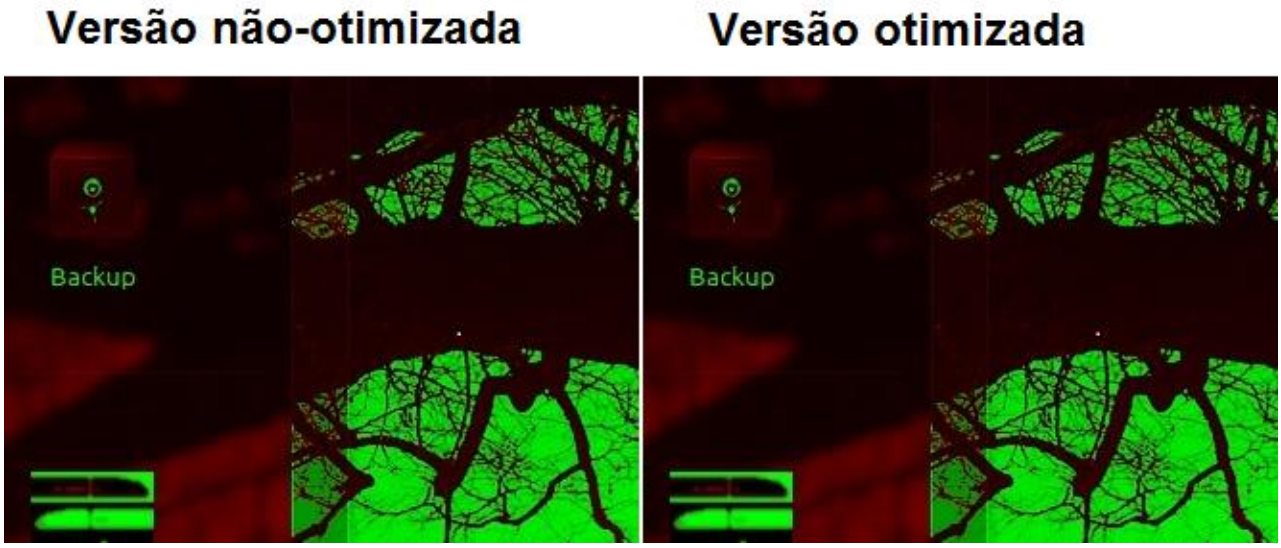


Figura 3.4: Comparação da qualidade visual entre a versão não-otimizada e a versão otimizada

A comparação do desempenho entre as versões original, versão com novo tratamento de cores sem otimização e versão com tratamento de cores otimizada se encontra na figura 3.4, mostrando que o impacto da versão otimizada na performance é bem pequeno em relação à versão original, o que justifica o novo tratamento de cores, tendo em vista uma grande melhora na qualidade visual per um pequeno custo adicional para a CPU.

Na figura 3.6 temos a comparação da qualidade visual entre as 3 versões abordadas anteriormente, considerando a mesma região da imagem do desktop do Ubuntu.

Figura 3.5: Comparação da qualidade visual entre as 3 versões abordadas

Considerando toda a análise feita, avaliando os resultados obtidos por ambas as versões, optamos por ficar com a versão otimizada, levando em consideração também que muitas das instituições que porventura venham a utilizar o xLupa podem ter máquinas com desempenho inferior à utilizada nos testes.

Nas seções 3.2, 3.3 e 3.4 apresentaremos os resultados da utilização de instruções SIMD, e a versão final do método de tratamento de imagem será apresentada na seção 3.5.

3.2 Instruções SSE

O SSE (*Streaming SIMD Extensions*) é uma unidade computacional SIMD (*Single Instruction, Multiple Data*) projetado pela Intel e disponibilizada pela primeira vez nos processadores da série Pentium III em 1999, e algum tempo depois a AMD adicionou o suporte ao SSE aos seus processadores Athlon XP.

Suas instruções possibilitam ao programador declarar e acessar registradores de 128 bits, que podem armazenar por exemplo, 4 floats de 32 bits cada, sendo possível a utilização de 4 cálculos em paralelo, visando melhorar o desempenho com suas instruções de movimentação de dados, operações aritméticas, operações de comparação, lógicas e de mistura, elaboradas de modo a aproveitar os 128 bits dos registradores.

3.3 Aplicação do SSE ao tratamento de cores

Para otimizar os cálculos utilizados no tratamento de cores declaramos variáveis do tipo m128 [11], que são registradores de 128 bits, os quais utilizamos para armazenar os valores das tuplas RGB e efetuar os cálculos necessários de forma paralela, e utilizamos funções SSE intrínsecas [12] da linguagem C. Na seção de declarações incluímos variáveis compatíveis, ficando o novo código como no algoritmo 4.

```
1       __m128 A, B, C;
2       float vec[4] __attribute__ ((aligned (16)));
3       int k = 0;
4       A = _mm_set_ps(rgb[0], rgb[1], rgb[2], 0);
5       B = _mm_set_ps(0.299, 0.587, 0.114, 0);
6       C = _mm_mul_ps(A, B);
7        _mm_store_ps(vec, C);
8        k = (vec[1] + vec[2] + vec[3]);
9                 if (k > 127) {
10                 if (cor_rgb1 != NULL) {
11                 A = _mm_set_ps(cor_rgb1[0], cor_rgb1[1], cor_rgb1[2], 0);
12                 B = _mm_set_ps(k, k, k, 0);
13                 C = _mm_mul_ps(A, B);
14                B = _mm_set_ps(256, 256, 256, 256);
15                 C = _mm_div_ps(C, B);
16                 _mm_store_ps(vec, C);
17                 rgb[0] = vec[3];
18                 rgb[1] = vec[2];
19                 rgb[2] = vec[1];
20                 }
21                    } else {
22                     if (cor_rgb2 != NULL) {
23                      cont++;
24                 A = _mm_set_ps(cor_rgb1[0], cor_rgb1[1], cor_rgb1[2], 0);
25                B = _mm_set_ps(k, k, k, 0);
25                 C = _mm_mul_ps(A, B);
26                B = _mm_set_ps(256, 256, 256, 256);
27                 C = _mm_div_ps(C, B);
28                 _mm_store_ps(vec, C);
29                 rgb[0] = vec[3];
30                 rgb[1] = vec[2];
31                 rgb[2] = vec[1];
32                    }
33             }
34              PUT_PIXEL24(subimage_data, lx, ly, source_rowstride, pixel);
```

Algoritmo 5: Método de tratamento de cores utilizando instruções SSE

Na linha 1 temos os registradores de 128 bits A, B e C, na linha 2 temos um vetor de floats com bits alinhados, que será utilizado para armazenar conteúdo de registradores de 128 bits, na forma de 4 floats de 32 bits, e na linha 3 temos a variável “k”, que é do tipo inteiro e receberá valores truncados. Abaixo temos o cálculo do “k” através de instruções SSE.

Na linha 4 o registrador A recebe a tupla RGB do pixel, na linha 5 o registrador B recebe as constantes para a multiplicação dos valores dos canais RGB do pixel respectivamente, na linha 6 o registrador C recebe o valor da multiplicação de A por B, na linha 7 o vetor “vec” recebe o registrador C e na linha 8 “k” recebe a soma das multiplicações.

A seguir temos o cálculo dos novos valores para os canais RGB do pixel, após a comparação de “k” com o limiar, e o código abaixo mostra o cálculo e atribuições destes valores utilizando instruções SSE. A comparação de k com o valor 127 ocorre na linha 9, e não sofre alterações, por isso omitimos sua descrição assim como a condição para o “else” o incremento no contador e a colocação do novo pixel na tela.

Na linha 11 o registrador A recebe os valores da tupla RGB da cor de fundo selecionada pelo usuário, na linha 12 o registrador B é preenchido com “k” (os zeros na quarta posição dos registradores A e B são necessários para preencher a quantidade de parâmetros), na linha 13 multiplica-se os valores RGB por “k”, na linha 14 preenchemos o registrador B com 256, na linha 15 dividimos C por B, na linha 16 o resultado é armazenado em “vec”, e das linhas 17 a 19 os novos valores RGB são atribuídos ao pixel que será colocado na tela.

A atribuição dos canais RGB da cor de fundo são análogas à cor de fonte, e é mostrada das linhas 24 à 31. Desta forma temos o método de tratamento das cores reescrito com instruções SSE, e o desempenho foi comparado à solução considerada a melhor no capítulo 3.1.

3.4 Comparação de desempenho com SSE

O desempenho da solução eleita a melhor no capítulo 3.1 foi comparado com o desempenho da versão em SSE, e o resultado, que não foi favorável à versão SSE pode ser visto na figura 3.7. A qualidade visual continuou idêntica às versões sem otimização e a versão otimizada.

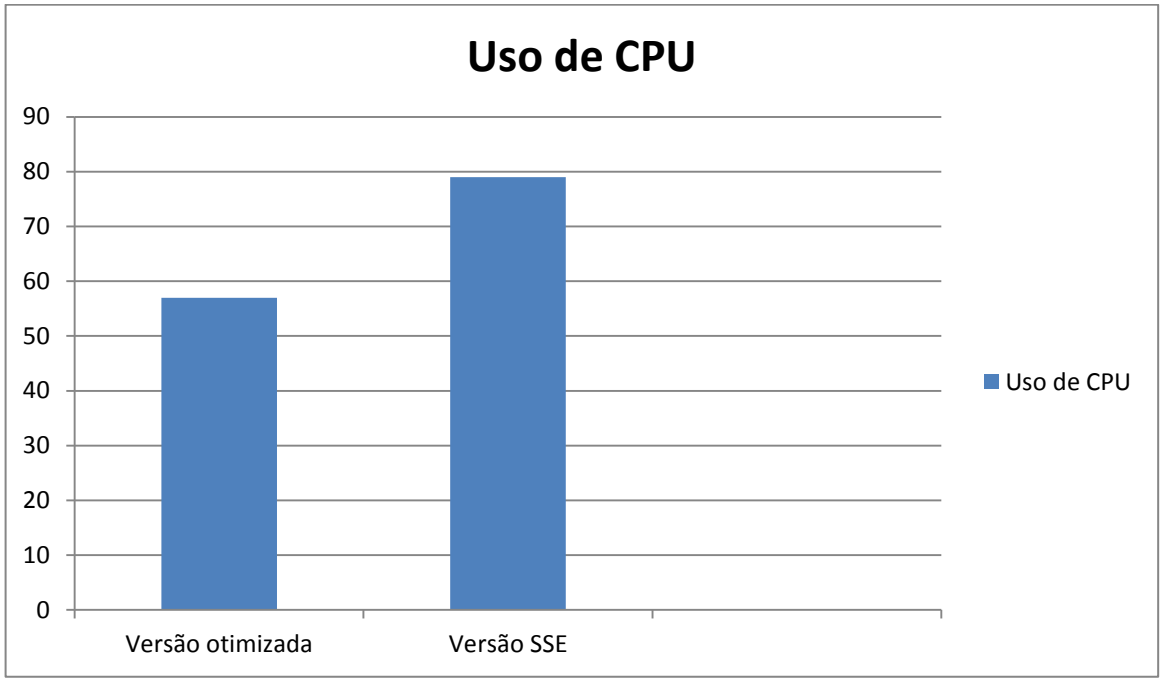


Figura 3.6: Comparação de desempenho com SSE

A versão otimizada ocupa 59% da CPU, enquanto a versão SSE ocupa 79% e tendo em vista que a versão não-otimizada do tratamento de cores ocupava 68% da CPU, a versão SSE mostra-se inviável até se comparada à versão com o pior desempenho entre as 4 abordadas neste capítulo.

Esta perda de desempenho deve-se ao fato do código não apresentar um nível de paralelismo suficiente para beneficiar-se de instruções SIMD. O uso de SSE é mais indicado para trabalhos que envolvem grande precisão nos cálculos em ponto flutuante ou grandes conjuntos de dados para cálculos em paralelo.

3.5 Versão Final do Método de Tratamento de Cores

Nesta seção apresentamos o resultado da substituição do cálculo da intensidade do pixel (equação 2.1) por uma aproximação deste resultado, que é a variável média, onde sacrificamos um pouco da precisão em prol do desempenho, como vemos na equação 3.1.

int media = (rgb[0]+rgb[1]+rgb[2])/3 (equação 3.1)

Assim temos a versão final do método de tratamento de cores, que apresentou o melhor desempenho sem perdas significantes na qualidade visual em relação às 4 outras versões vistas neste capítulo, o código final fica sendo o algoritmo 5.

```
1   int media = (rgb[0]+rgb[1]+rgb[2])/3;
2   if(media > 127){
3       if(cor_rgb1!=NULL){
4           rgb[0]=(cor_rgb1[0]*media)/256;
5           rgb[1]=(cor_rgb1[1]*media)/256;
6           rgb[2]=(cor_rgb1[2]*media)/256;
7       }
8   }else {
9       if(cor_rgb2!=NULL){
10          cont++;
11          rgb[0] = (cor_rgb2[0]*media)/256;
12          rgb[1] = (cor_rgb2[1]*media)/256;
13          rgb[2] = (cor_rgb2[2]*media)/256;
14      }
15  }
16  PUT_PIXEL24 (subimage_data, lx, ly, source_rowstride, pixel);
```

Algoritmo 6: Versão final do método de tratamento de cores

A figura 3.7 mostra a comparação do desempenho entre as 5 versões abordadas neste capítulo, justificando nossa decisão em manter o algoritmo 5 como versão final do método de tratamento de cores.

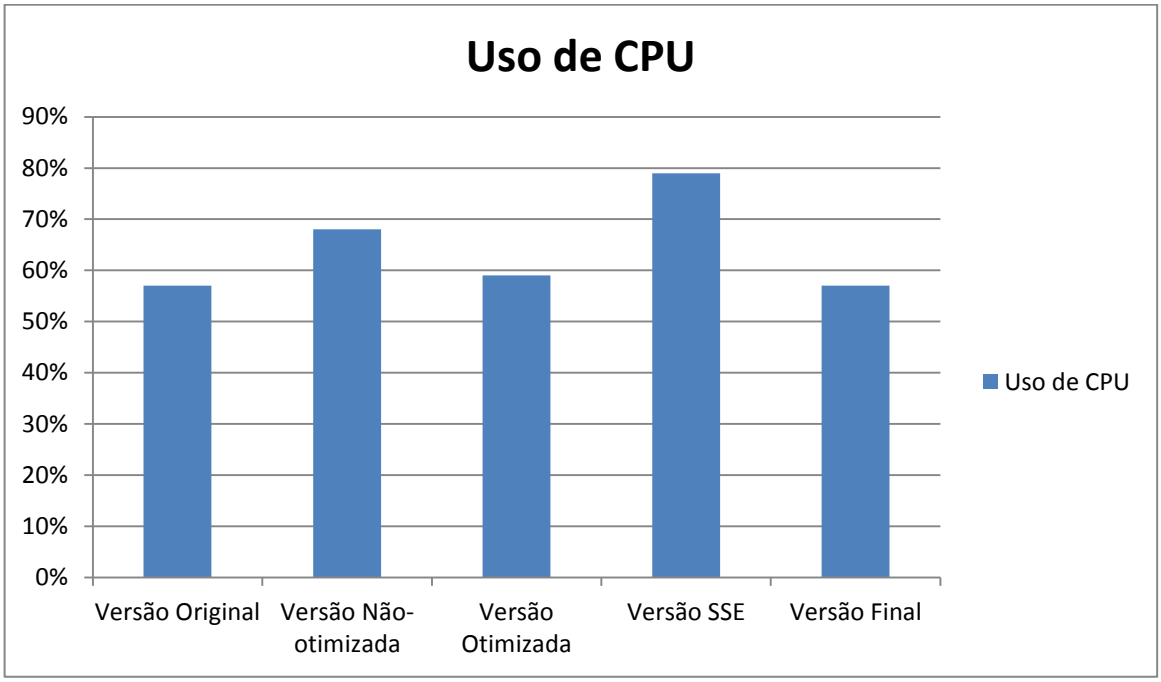


Figura 3.7: Comparação de entre as 5 versões abordadas neste capítulo

Podemos observar que o desempenho da versão final, trocando K pela variável média, é igual ao desempenho da versão original (57%). Deste modo a versão final não causa perda de desempenho em relação à original mesmo apresentando uma qualidade visual superior, com menos ocultação de detalhes.

Capítulo 4

Conclusões

Houve um significante progresso no tratamento de cores utilizado pelo xLupa no que se refere à qualidade da imagem, devido ao estudo, implementação e combinação de técnicas de processamento de imagens digitais, como foi constatado pela análise subjetiva feita pela equipe envolvida no progresso.

O xLupa ainda tem espaço para aprimoramentos, em seu código em C++, no que se refere ao tratamento de cores, é bem acessível, facilitando experimentos com variadas técnicas de tratamento de imagem que porventura venham à ser utilizados em trabalhos futuros.

Os principais problemas encontrados quando as combinações de cores eram ativadas já tem uma solução que se mostra eficiente na conservação dos detalhes entre a imagem original e a imagem resultante do tratamento de cores.

O desempenho do tratamento de cores foi melhorado através da otimização dos cálculos utilizados no código, ficando igual ao desempenho da versão original, porém com a qualidade visual aprimorada.

Destacaremos aqui alguns elementos do xLupa que conhecemos durante o trabalho e cujo aprimoramento poderia fazer parte de trabalhos futuros.

O cursor do mouse é uma imagem JPG [4] fixa, que não sofre alterações resultantes da aplicação de cores e fatores de ampliação, o que poderia ser trabalhado de modo à se adequar às necessidades de usuários que necessitem de tamanhos diferenciados de cursor ou cores com as quais tenham mais facilidade de trabalhar.

A cruz do mouse é um recurso disponível do xLupa para facilitar a visualização do cursor, porém não agradou aos usuários, e a exploração de métodos para a localização do

cursor por pessoas de baixa visão pode resultar em um recurso viável à ser agregado ao xLupa.

Quando um fator de ampliação muito grande é selecionado, temos uma redução na qualidade da imagem gerada pelo aliasing [7], ou seja, serrilhados que se tornam mais evidentes com a ampliação, como se a densidade de resolução [4] fosse reduzida e utilização de GPU para acelerar o processamento poderiam vir à ser explorados em eventuais trabalhos futuros.

Testes com usuários de baixa visão ainda podem ser feitos em trabalhos futuros com a finalidade de avaliar as funcionalidades da versão atual do software e identificar possíveis defeitos não percebidos pela equipe do projeto.

Referências Bibliográficas

[1] ASSIS, LOURO. *Saber: Dados do IBGE sobre deficiência*. Consultado na Internet: http://louroassis.blogspot.com.br/2011/12/saber-dados-do-ibge-sobre-deficiencia.html, em 30/06/2013.

[2] SOUZA, ODAIR M. *xLupa*. Consultado na Internet: http://projetos.unioeste.br/campi/xlupa/, em 09/04/2013.

[3] AGUILAR, LUIS J. *Programação em C++: Algoritmos, Estruturas de Dados e Objetos*. Porto Alegre: MacGraw Hill, 2008.

[4] GONZALES, R.l C & WOODS, R.E. *Processamento de Imagens Digitais*. São Paulo: Blücher Ltda, 2000.

[5] MUELLER, TOBIAS. *Gnome Dev Center*. Consultado na Internet: https://developer.gnome.org/gtkmm/stable/, em 15/06/2013.

[6] SCORTEGAGNA, ALEXANDRE S. *Estudo e Aplicação de Algoritmos para o Tratamento do Contraste no Ampliador de Tela xLupa*. EAIC 2012, Maringá – PR. Consultado na Internet: http://www.eaic.uem.br/eaic2012/anais/artigo.php?cod=1527, em 02/03/2013.

[7] CONCI, A. & AZEVEDO, E. *Computação Gráfica - Teoria e Prática*. Rio de Janeiro: Campus, 2003.

[8] HILLEBRANDT, TIAGO. *Comunidade Ubuntu Brasil*. Consultado na Internet: http://www.ubuntu-br.org/, em 22/07/2013.

[9] DEITEL, PAUL J. *C++ How to Program (6th Edition)*. Rio de Janeiro: Pearson/Prentice Hall, 2007.

[10] MANZANO, JOSÉ AUGUSTO N. G., *Programação Assembly*. São Paulo: Érica, 2008.

[11] MORIMOTO, CARLOS E. *Instruções SSE, Manual completo*. Consultado na Internet: http://www.hardware.com.br/livros/hardware-manual/instrucoes-sse.html, em 13/09/2013.

[12] BANCROFT, JOSHUA. *Intel® Intrinsics Guide*. Consultado na INTERNET: http://software.intel.com/en-us/articles/intel-intrinsics-guide, em 15/09/2013.