Estudo de desempenho de processamento de imagens utilizando multithreading

Lucas Frank Hollmann   
Graduação em Ciência da Computação  
UNIOESTE - Universidade Estadual do Oeste do ParanáCascavel, Brasil  
lucasfrank\_@hotmail.com

Pedro Henrique de Moraes Xavier  
Graduação em Ciência da Computação  
UNIOESTE - Universidade Estadual do Oeste do ParanáCascavel, Brasil  
pedro.xavier1@unioeste.br

Willian Cavaller Faino  
Graduação em Ciência da Computação  
UNIOESTE - Universidade Estadual do Oeste do ParanáCascavel, Brasil  
willianfaino@hotmail.com

# Introdução

Na programação de softwares de computador, normalmente fazemos uso de uma única linha de execução, processando um comando após o outro, no entanto, a maioria dos processadores possuem mais de uma unidade de processamento, possibilitando a divisão das tarefas de um software em múltiplos threads, ou seja, múltiplas linhas de execução paralelas.

O objetivo deste trabalho é aplicar e testar a eficiência de um algoritmo para remover uma cor de uma imagem no formato TIF sendo executado em um thread e comparar com a execução em múltiplos threads.

# Descrição do Problema

O algoritmo proposto neste trabalho é baseado na técnica de efeito visual *chroma key,* que consiste em eliminar uma cor, e cores que se aproximam da cor que se quer eliminar, de um vídeo, essa eliminação é feita a partir do aumento da transparência de pixels que possuem as cores alvo. Essa técnica é, maioritariamente, usada em vídeos em que existe um fundo com uma cor destacada, para isolar algum ou alguns elementos do vídeo, depois, os elementos isolados podem ser inseridos em outros vídeos.

O algoritmo aqui proposto difere, principalmente, da *chroma key,* por só poder ser aplicado em imagens com um formato específico (TIF ou TIFF), porém tem o mesmo objetivo de remover uma cor específica de uma imagem, deixando transparente as regiões onde a cor foi removida.

# Descrição das Implementações

A implementação do algoritmo foi feita na linguagem de programação C++. Inicialmente, abre-se o arquivo da imagem, deste arquivo, é possível adquirir o tamanho da imagem (em bytes), desconsiderando os bytes de informações adicionais, ou seja, apenas a quantidade de bytes usadas para armazenar os pixels. Logo após obter a quantidade de bytes que devem ser analisados pelo algoritmo, copiasse esses mesmos bytes para um vetor. Após isso, o usuário deverá informar a cor (pelo valor dos bytes no formato RGB) que se quer eliminar, uma tolerância, que é usada para decidir o quão próximo da cor alvo um pixel deve estar para ser alterado, além do valor da transparência, que pode variar para cada pixel, dependendo da tolerância e da proximidade entre a cor do pixel e a cor alvo. Por fim o usuário deve informar a quantidade de threads que deseja utilizar para a execução do algoritmo.

O vetor com os bytes da imagem é, então, percorrido de quatro em quatro bytes, ou seja, pixel por pixel (um pixel é composto por quatro bytes, um byte para cada um dos componentes vermelho, verde e azul e um ultimo byte para a transparência) e, a cada pixel, calcula a proximidade entre a cor do pixel e a cor alvo, baseando nessa proximidade, decide se o pixel será alterado ou não, caso seja alterado, o byte de transparência do pixel é alterado com base nessa proximidade e o contador de pixels alterados é incrementado, esse incremento deve ser envolto por um semáforo para impedir o incremento simultâneo, que causaria erros na soma. Essa é a parte do algoritmo que pode ser dividido entre múltiplos threads, caso sejam utilizados mais de um thread, cada thread irá percorrer um segmento do vetor que é controlado através da *ID* do thread.

Por fim, o vetor é escrito em outro arquivo, que gera uma nova imagem com as devidas alterações.

# Análise dos Resultados

As imagens a seguir mostram o funcionamento do algoritmo.



Figura 1: imagem de entrada do software

A cor escolhida para ser removida, em RGB, tem o componente vermelho com o valor de 97, o verde com valor 162, e o azul, 190. Essa se assemelha à cor do céu da imagem. Foi escolhida uma tolerância de valor 15.



Figura 2: imagem de saída do software (áreas quadriculadas representam a transparência)

Na imagem resultante, podemos ver que uma parte do céu foi removido, porém ainda faltaram algumas partes, também vemos que outras regiões da imagem foram removidas. A próxima figura mostra uma tentativa com uma imagem mais simples, com um fundo de uma cor sólida, o objetivo é remover o fundo.

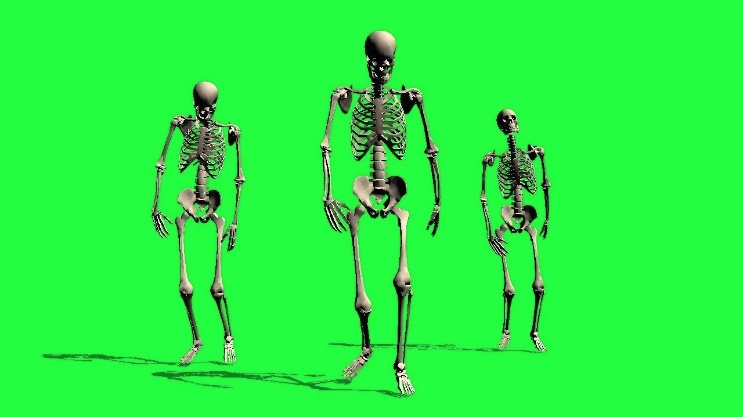


Figura 3: Outra entrada do algoritmo

Desta vez, a cor escolhida possui o componente vermelho com valor 0, verde com valor 255 e azul, 0 (verde puro). Foi usada uma tolerância de valor 40. A imagem a seguir mostra os resultados.

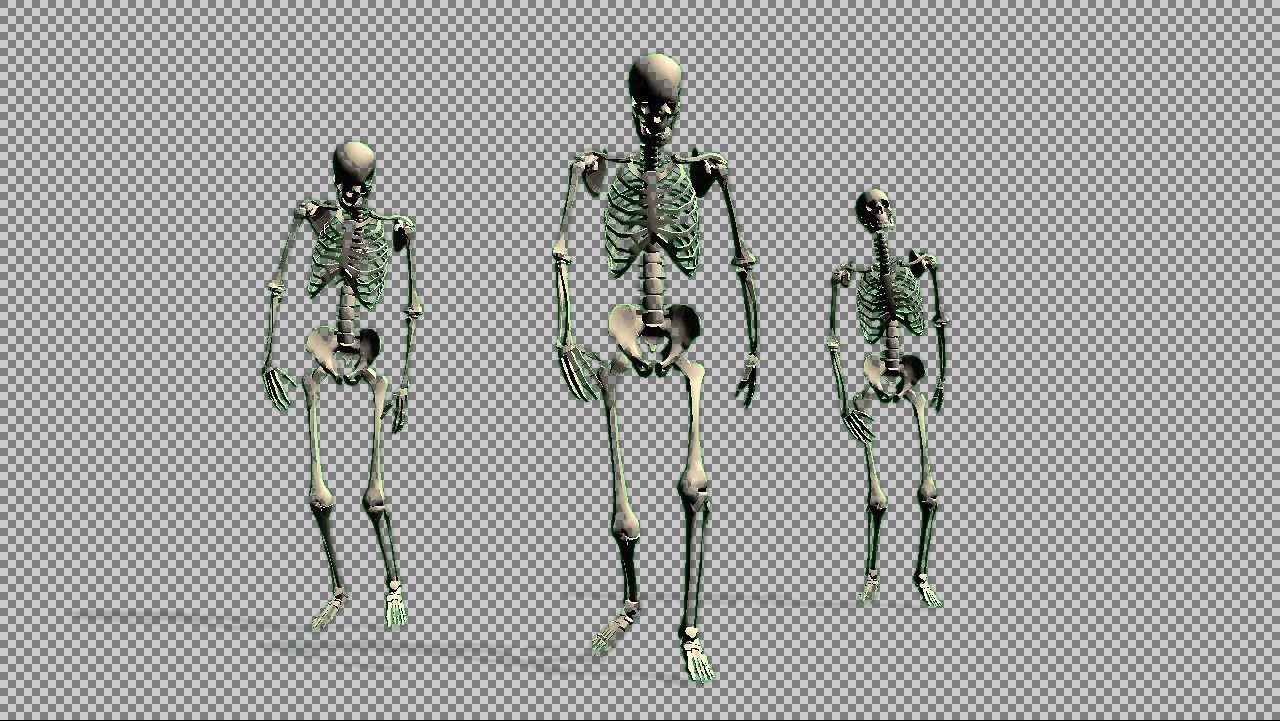


Figura 4: Saída do algoritmo para a segunda imagem

Dessa vez tivemos um resultado mais satisfatório, apesar das visíveis falhas.

Para os testes de desempenho, foram escolhidas imagens com apenas uma cor, com objetivo de remover todos os pixels, ou não remover nenhum, assim, testando os casos extremos que podem ser executados pelo algoritmo.

A tabela a seguir apresenta os resultados obtidos ao remover todos os pixels da imagem.

|  |  |  |
| --- | --- | --- |
| Tamanho da imagem (em pixels) | Tempo de execução (em segundos) | |
| um thread | quatro threads |
| 1280 x 720  (3.686.400 bytes) | 0,056219 | 0,161662 |
| 1920 x 1080  (8.294.400 bytes) | 0,077615 | 0,297510 |
| 3840 x 2160  (33.177.600 bytes) | 0,192956 | 1,085390 |
| 7680 x 4320  (132.710.400 bytes) | 0,722857 | 4,379740 |

Tabela 1: resultados ao remover todos os pixels

A próxima tabela mostra os resultados ao não remover nenhum pixel.

|  |  |  |
| --- | --- | --- |
| Tamanho da imagem (em pixels) | Tempo de execução (em segundos) | |
| um thread | quatro threads |
| 1280 x 720  (3.686.400 bytes) | 0,023485 | 0,014429 |
| 1920 x 1080  (8.294.400 bytes) | 0,038641 | 0,024023 |
| 3840 x 2160  (33.177.600 bytes) | 0,057515 | 0,031137 |
| 7680 x 4320  (132.710.400 bytes) | 0,157863 | 0,103838 |

Tabela 1: resultados ao não remover nenhum pixel

Podemos observar que, ao remover todos os pixels, o resultado com quatro threads se torna muito mais lento em relação aos resultados com um único thread. Isso acontece devido ao alto gasto de desempenho com os semáforos, que são usados para realizar a sincronização entre os threads. Já os resultados sem alteração em nenhum pixel apresentam uma melhora significativa ao utilizar quatro threads, isso acontece porque o semáforo só é executado se o pixel for alterado, como não há alterações, os threads podem executar sem interrupções.

# Conclusões

Em teoria, o uso de múltiplos threads oferece uma grande melhoria de desempenho nos softwares, porém nem sempre é assim, muitas vezes o algoritmo pode vir a perder desempenho ao invés de se tornar mais rápido. Algoritmos com uma necessidade muito grande de sincronização tendem a perder desempenho pois os threads são constantemente interrompidos pelos comandos de sincronização, fazendo com que a execução com vários threads se aproxime de uma execução linear, além disso, o tempo de execução dos comandos de sincronização fazem com que o algoritmo tome ainda mais tempo. Porém em algoritmos com pouca, ou nenhuma, necessidade de sincronização, a execução em múltiplos threads, de fato, aumenta significativamente a velocidade de execução.

O algoritmo apresentado neste trabalho conseguiu chegar à uma melhoria de desempenho ao processar imagens onde nenhum pixel é alterado, porém, perdeu muito desempenho no caso contrário. Mas, na maioria dos casos, as imagens que precisam desse tipo de processamento possuem uma grande quantidade de pixels a serem removidos (como a figura 3), portanto podemos dizer que não vale a pena o uso de vários threads para esse tipo de processamento de imagem.