GUSTAVO FRANCISCO FRIZZO

ALGORITMO PARALELO PARA MORFISMO DE IMAGEM EM ARQUITETURA MULTIPROCESSADA

JOINVILLE – SC

2011

UNIVERSIDADE DO ESTADO DE SANTA CATARINA UDESC

CENTRO DE CIÊNCIAS TECNOLÓGICAS CCT

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

GUSTAVO FRANCISCO FRIZZO

ALGORITMO PARALELO PARA MORFISMO DE IMAGEM EM ARQUITETURA MULTIPROCESSADA

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Gilmário Barbosa dos Santos

JOINVILLE – SC

2011

GUSTAVO FRANCISCO FRIZZO

ALGORITMO PARALELO PARA MORFISMO DE IMAGEM EM ARQUITETURA MULTIPROCESSADA

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação Integral do CCT/UDESC.

Banca Examinadora:

Orientador**:** _____

MSc, Gilmário Barbosa dos Santos
Departamento de Ciência da Computação – UDESC

Co-orientador**:** _____

Bel, Marlon Fernandes de Alcantara
Departamento de Ciência da Computação – UDESC

Membro**:** _____

Dr, Alexandre Gonçalves Silva
Departamento de Ciência da Computação – UDESC

Membro**:** _____

Dr, Marcelo da Silva Hounsell
Departamento de Ciência da Computação – UDESC

Joinville, 21 de novembro de 2011

AGRADECIMENTOS

Aos meus pais, pelo apoio e motivação para seguir em frente, mesmo estando tão distantes.

Aos meus amigos, pelos momentos de descontração e alegria, que foram extremamente importantes nesta jornada.

A todos os professores da UDESC que contribuíram para minha formação.

Ao professor Gilmário Barbosa dos Santos pela orientação.

“Talvez não tenhamos conseguido fazer o melhor, mas lutamos para que o melhor fosse feito.”

Martin Luther King

RESUMO

A operação de morfismo consiste basicamente em um mapeamento entre duas imagens distintas. O morfismo apresenta diversas aplicações das quais a mais comercial é aquela encontrada no setor de entretenimento (especialmente no cinema, televisão e jogos de computador). Devido às suas características, uma aplicação do morfismo de imagens demanda intenso processamento computacional, o que serve de motivação para estudos visando a paralelização dos algoritmos utilizados. Por outro lado, atualmente dispõem-se cada vez mais de arquiteturas paralelas (multicore - múltiplos núcleos) e de custo acessível. Nesse trabalho, o potencial de paralelismo de um algoritmo de morfismo foi analisado, e uma estratégia paralela foi implementada utilizando uma arquitetura multiprocessada de custo acessível (disponível em versão desktop). Foram feitos experimentos, e os resultados obtidos com os testes de *Speedup*, eficiência e Correlação de Pearson demonstram que a técnica apresenta ganhos de desempenho em arquiteturas multiprocessadas.

Palavras-chave: *processamento de imagens, geometria computacional, morfismo no decorrer do tempo, deformação, processamento paralelo.*

ABSTRACT

The morphing operation is basically a mapping between two different images. The morphing has many applications the most commercial of which is that found in the entertainment industry (especially in cinema, television and computer games). Due to its characteristics, application of a morphing of images demand intense computational processing which serves as motivation for studies aimed at the parallelization of algorithms. On the other hand, currently have to increasingly parallel architectures (multi-core) and affordable. In this work we analyzed the potential parallelism of an algorithm morphing and implemented a parallel strategy using a multi-core architecture affordable (available in desktop version). Experiments were made, and the results obtained from the tests Speedup, efficiency and Pearson Correlation show that the technique has performance gain in multiprocessor architectures.

Keywords: *image processing, computational geometry, morphing over the time, deformation, parallel processing.*

LISTA DE FIGURAS

LISTA DE SIGLAS E ABREVIATURAS

MFFD - *Multilevel Free-Form Deformation*

MIMD - *Multiple Instruction Multiple Data*

MISD - *Multiple Instruction Single Data*

SIMD - *Single Instruction Multiple Data*

SISD - *Single Instruction Single Data*

CPU - *Central Processing Unit*

GPU - *Graphics Processing Unit*

SUMÁRIO

1 INTRODUÇÃO

Em geral, o termo “morfismo” é definido como um mapeamento (relação) entre dois objetos pertencentes a uma mesma categoria ou conjunto (WEISSTEIN, 2011). Seguindo a mesma linha de pensamento, o morfismo de imagens é uma transformação completa de uma imagem com a modificação da relação espacial e do valor dos seus *pixels* (SCHROEDER, 2007). É encontrado em uma larga gama de aplicações que vão desde a área gráfica (SCHROEDER, 2007), passando pela zoologia (HODGES *et al*, 2003) e pela medicina legal (RITZ-TIMME *et al*, 1999). Porém uma das áreas onde mais se detecta a aplicação do morfismo é na área de entretenimento (efeitos especiais no cinema e televisão) e jogos.

Aqui são discutidas duas estratégias importantes de morfismo as quais são baseadas em *crossfading* e em deformação. *No Crossfading,* a técnica mais simples, ocorre uma interpolação linear entre as cores de cada *pixel* da imagem inicial com as cores de cada *pixel* da imagem final, sem levar em consideração a alteração da posição entre eles.

No que se refere ao morfismo baseado em deformações, este pode ser subdividido em vários ramos dos quais se destacam: *Multilevel free-form deformation* (MFFD) e Malha Deformável:

- MFFD utiliza feições modeladas com contornos deformáveis (*Snakes*) (KASS; WITKIN; TERZOPOULOS, 1987). Tal modelo baseia-se em uma *B-Spline* que evolui de acordo com um processo de minimização de energias. Apesar de apresentar propriedades importantes, o sucesso desse método baseia-se na eficiência da estratégia de otimização aplicada no processo de minimização da energia da *snake* (BRESSON *et al*, 2005).
- Quanto à Malha Deformável, ocorre uma marcação de pontos que estão associados às principais características da imagem, como por exemplo, na face humana, onde as principais características seriam os olhos, o nariz, a boca, etc. Esses pontos darão origem a malha que por sua vez separa a imagem em regiões. É nessas regiões que ocorre o morfismo.

A técnica de geração de imagens abordada neste trabalho baseia-se na Malha Deformável. Esta técnica é baseada na interpolação de pontos pré-determinados manualmente entre duas imagens (inicial e final).

Na Figura 1 (direita) são exibidas três imagens representadas por quadriláteros (planos). A imagem inicial é representada no instante *t = 0* enquanto a imagem final encontra-se no instante *t = 1*. A imagem no instante *t* tem seus pontos determinados a partir da interpolação dos seus respectivos pontos nos Instantes *0* e *1*. Na Figura 1 (esquerda) observa-se a deformação de um polígono (e seus pontos constituintes) no instante *t* como resultado da interpolação entre o polígono inicial (*t = 0*) e o final *(t = 1*). Esse processo se repete para todos os polígonos da malha determinando a imagem/morfismo no instante *t*. A interpolação resulta de uma relação entre a imagem inicial e a imagem final determinando o morfismo desejado, ou seja, a sequência de deformações da imagem inicial até ser alcançada a imagem final, em um processo suave e sob as restrições impostas pela relação entre a malha inicial e a malha final.

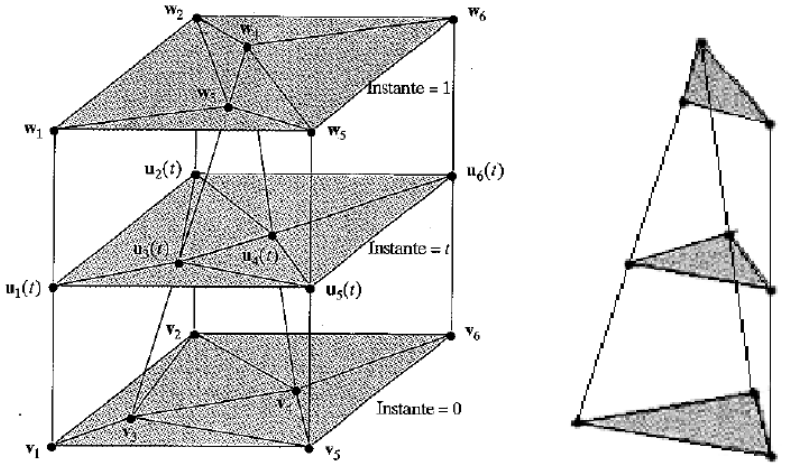


Figura 1 – Interpolação entre duas imagens a inicial (t = 0) e a imagem final (t = 1).

Na figura á esquerda é exibida a relação entre a malha inicial em t = 0 e a malha final em t = 1. Na figura à direita é exibida a deformação para um único polígono da malha.

O objetivo deste trabalho é estudar o ganho de desempenho ao executar o morfismo baseado em Malha Deformável de forma paralela, visto que existe uma necessidade de resultados cada vez mais rápidos no ramo de processamento de imagens (BARBOSA, 2000).

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

O objetivo geral deste trabalho é paralelizar e avaliar o desempenho do algoritmo clássico de morfismo baseado em Malha Deformável.

1.1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos são:

- Implementar e avaliar o morfismo por Malha Deformável;
- Avaliar o impacto da paralelização em multiprocessadores bem como a qualidade do morfismo obtido;

1.2 JUSTIFICATIVA E METODOLOGIA

Processamento digital de imagens é uma área da Ciência da Computação que demanda muito poder de processamento (BARBOSA, 2000). A necessidade de poder computacional decorre do fato das imagens digitais geralmente serem grandes matrizes, e as operações matemáticas dos algoritmos devem ser realizadas sobre todos os elementos (*pixels*) dessas matrizes (PENHA *et al*, 2002). Além disso, os resultados estão cada vez mais exigentes em termos de qualidade (resolução, profundidade de cores). Por isso, essas operações geram um alto tempo de espera (de alguns minutos à até mesmo várias horas) pelo resultado, sendo que se executado em paralelo, o tempo de espera pode ser muito menor. Por outro lado, os computadores multiprocessados têm oferecido um maior poder computacional a um custo bastante acessível. A motivação deste trabalho decorre da popularização desta arquitetura frente à demanda de processamento dos algoritmos tais como o algoritmo de morfismo abordado neste trabalho.

Inicialmente foi feita uma pesquisa bibliográfica para levantar e esclarecer os principais conceitos e técnicas sobre morfismo em imagens e paralelização de

algoritmos, assim como o levantamento de trabalhos que estão sendo desenvolvidos atualmente no campo do morfismo e processamento paralelo de imagens. Então, na pesquisa exploratória, o algoritmo descrito de forma textual em Anton e Rorres (2005) é analisado e uma estratégia de paralelização é proposta, aspecto que não está exposto na descrição do mesmo. Um banco de imagens foi criado, visto que não foi encontrado algo que possa ser usado para comparação de resultados.

Após a criação do banco de imagens, o algoritmo foi implementado e depois paralelizado. Os experimentos consistiram em escolher pares de figuras do banco de imagens, e a partir delas, aplicar o algoritmo de morfismo. Em relação às métricas de avaliação, elas estão descritas no Capítulo 4. O modo com que estas métricas foram utilizadas também está descrito neste capítulo.

1.3 ESTRUTURA DO TRABALHO

Este trabalho está estruturado da seguinte forma:

- **Capítulo 2**: Fundamentação teórica, contendo os principais conceitos sobre arquiteturas paralelas e o levantamento das principais técnicas de morfismo. Também descreve alguns conceitos importantes para a implementação do trabalho;
- **Capítulo 3**: Expõe Trabalhos Correlatos que assemelham-se aos objetivos propostos anteriormente.
- **Capítulo 4**: Descrição e análise do algoritmo de morfismo implementado;
- **Capítulo 5:** Projeto do algoritmo paralelo baseado na técnica de morfismo utilizando Malha Deformável. Também descreve as métricas que serão usadas para avaliar o algoritmo e algumas funções adicionais que foram implementadas, bem como os motivos que determinaram a escolha da linguagem de programação.
- **Capítulo 6:** Apresenta a avaliação dos resultados obtidos nos testes.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados conceitos importantes ao trabalho como técnicas de morfismo, arquiteturas e processamento paralelo.

2.1 MORFISMO

O morfismo de imagens é uma ferramenta de geração de efeitos especiais na televisão e no cinema, cujo objetivo principal é gerar transições suaves entre imagens (WOLBERG, 1998). A priori, o morfismo é gerado a partir de duas imagens, onde uma imagem é transformada na outra por meio de deformações e interpolação de cores. Também é amplamente usado em aplicações científicas, como por exemplo, no estudo da evolução dos seres vivos e na análise do seu crescimento e desenvolvimento (HODGES *et al*, 2003), assim como na assistência a cirurgias plásticas e de reconstrução facial (ANTON e RORRES, 2005).

Como exemplo da aplicação do morfismo, temos uma sequência de transições no clipe musical de Michael Jackson "*Black or White*", que pode ser visto na Figura 2 (BEIER e NEELY, 1992), e a *Disney*, que utiliza o morfismo temporal para gerar imagens intermediárias entre dois *frames* (SOBCHACK, 2000), acelerando o processo de produção das animações.

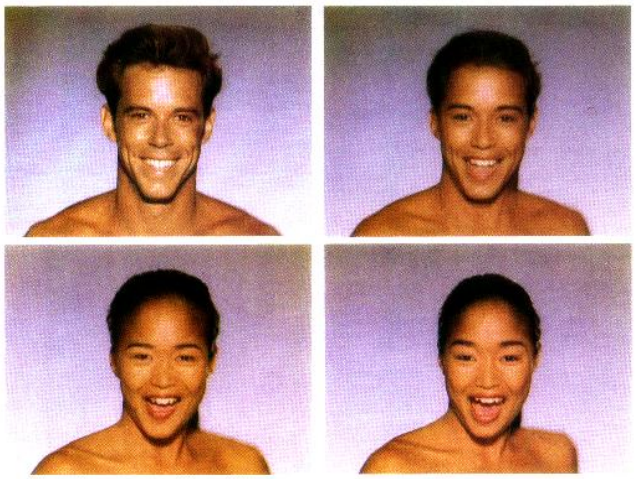


Figura 2 - Exemplo de utilização do morfismo

Basicamente, existem duas estratégias principais de morfismo as quais são baseadas em *crossfading* ou em deformação. Dentro do morfismo por deformação é

dado destaque ao *Multilevel free-form deformation* (MFFD) (baseada em *snakes*) e à Malha Deformável (*mesh warping*).

2.1.1 *Crossfading* (*Mixing*)

Crossfading é uma técnica de mistura de imagens, onde duas imagens são sobrepostas de forma ponderada e normalizada através de interpolação linear. Em outras palavras, duas imagens são colocadas juntas onde cada uma contribuirá com uma porcentagem pré-definida na imagem de saída.

A equação usada para gerar o *Crossfading* é a seguinte:

$$CF = (MV * A) + [(1 - MV) * B] \quad (2.1)$$

Onde *A* e *B* são duas imagens qualquer e *MV* refere-se ao valor da mistura (porcentagem de visualização da primeira imagem). Como é possível notar, a soma entre as porcentagens de *A* e *B* sempre resulta em 1. Um exemplo pode ser visto na Figura 3 (BRINKMANN, 1999).

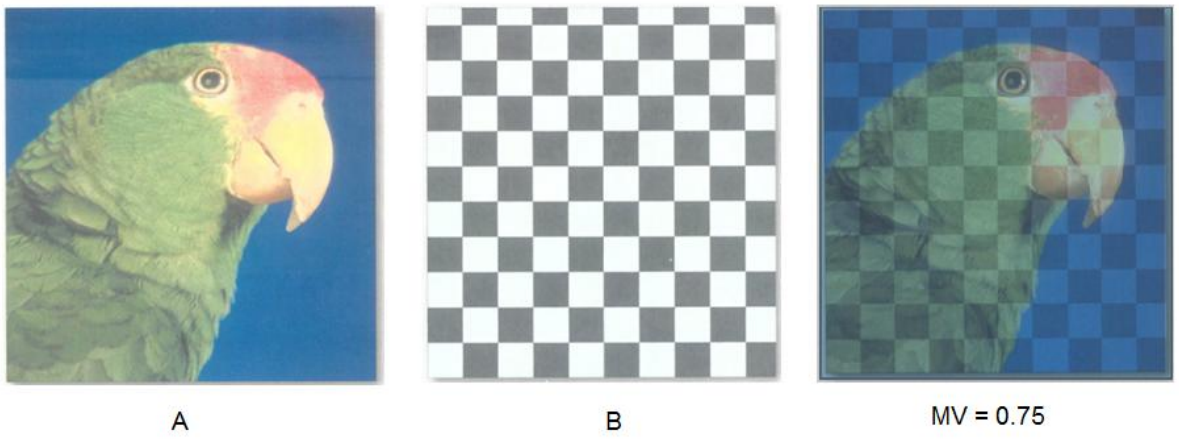


Figura 3 - *Crossfading* entre as imagens *A* e *B* com *MV = 0.75*

Para gerar este efeito durante um determinado período de tempo, é necessário somente iniciar o valor de *MV* em 1, para que a imagem *A* seja exibida completamente, e ir decrementando-o até que ele chegue em 0, onde a imagem *B*

será exibida por completo (BRINKMANN, 1999). Um exemplo pode ser observado na Figura 4:

Figura 4 - Exemplo de *Crossfading* no tempo (WOLBERG, 1998)

Esta técnica evoluiu muito pouco desde sua criação. Originalmente era usada por sua eficiência computacional, já que a base do algoritmo é interpolação linear, e ainda hoje é usada para gerar misturas entre imagens, por ser uma técnica extremamente simples. Mas toda essa simplicidade possui desvantagens claras. Sempre que as imagens são combinadas por interpolação linear, ocorre uma perda de contraste, brilho e nitidez. O resultado do uso da técnica é uma imagem desbotada e sem vida onde muitos detalhes são perdidos (GRUNDLAND *et al*, 2006).

Um problema semelhante ocorre quando se usa *Crossfading* para gerar transições entre sequências nos filmes. Para se conseguir um resultado mediano nas transições que durem mais que uma fração de segundo, é preciso usar vários filtros para regular o contraste e o brilho das imagens (GRUNDLAND *et al*, 2006).

2.1.2 *Multilevel Free-Form Deformation* (MFFD)

Esta técnica de morfismo é interessante, pois utiliza o modelo de contorno ativo (*snakes*) (KASS, 1987), para segmentar as feições da imagem. Este modelo é físico-inspirado, e baseia-se na minimização de uma série de parâmetros (energias) para deformar as *snakes*, e deixá-las na posição desejada.

Este modelo básico de *snake* é uma *spline* (curva) controlada continuamente, sob a influência das forças da imagem, forças de restrição externa e as forças internas. As forças internas servem para impor uma restrição de suavidade à *snake*. As forças da imagem empurram a *snake* em direção aos pontos mais salientes da

imagem como linhas, arestas e contornos ou então a pontos pré-definidos. As forças de restrição externa são responsáveis por colocar a *snake* no mínimo local desejado. Estas forças podem vir, por exemplo, da interface do usuário ou até mesmo de mecanismos automáticos de detecção (KASS, 1987).

Uma *snake* pode ser representada parametricamente através da expressão *v(s) = (x(s), y(s))*, e a atuação das energias sobre ela é dada pela equação 2.2:

$$\begin{aligned} E^*_{\text{snake}} &= \int_0^1 E_{\text{snake}}(\mathbf{v}(s))\, ds \\ &= \int_0^1 E_{\text{int}}(\mathbf{v}(s)) + E_{\text{image}}(\mathbf{v}(s)) \\ &\quad + E_{\text{con}}(\mathbf{v}(s))\, ds \end{aligned} \tag{2.2}$$

onde E_{int} representa a energia interna, E_{image} representa as forças da imagem e E_{con} representa as energias de restrição externa (KASS, 1987).

Então, dadas duas imagens, uma inicial e outra final, o processo de morfismo inicia com a segmentação das duas imagens com as *snakes*. Com isso é criada uma relação entre as *snakes* da imagem inicial e final. Por fim, são aplicadas forças nas snakes da imagem inicial até que elas cheguem à forma da *snake* correspondente na imagem final.

Os resultados obtidos com a técnica MFFD podem ser observados na imagem abaixo (Figura 5):

Figura 5 - *Multilevel free-form deformation* (MFFD) (WOLBERG, 1998)

Apesar de possuir propriedades importantes, o sucesso deste método depende de um eficiente e complexo método de minimização de energia. Outro problema é que as *snakes* não podem se mover para lugares muito distantes da sua origem, e nem para concavidades de fronteira (como a parte superior da letra ‘U’), além de ser um método computacionalmente caro.

2.1.3 Malha Deformável (*Mesh Warping*)

A técnica de Malha de Pontos Deformável é semelhante à técnica MFFD, mas em vez de utilizar linhas para separar as principais regiões da imagem, ela utiliza pontos, que por sua vez dão origem a uma malha que separa as regiões características da imagem em uma série de polígonos independentes. Um exemplo pode ser observado na Figura 6, onde no *Instante* 0 temos a imagem inicial e no *Instante* 1 temos a imagem final. Como é possível notar, cada ponto da imagem inicial tem um correspondente na imagem final, característica que gera também um polígono de correspondência entre as imagens. Para originar a imagem intermediária, é preciso interpolar os pontos correspondentes da imagem inicial e final. A partir disso, a malha se deformará, originando a imagem intermediária. Um exemplo de morfismo utilizando malha de pontos pode ser observado na Figura 7. Essa técnica será melhor explicada no Capítulo 4.

A técnica de geração de imagens abordada neste trabalho baseia-se na Malha Deformável e foi escolhida porque divide a imagem em vários pedaços isolados, característica imprescindível em algoritmos paralelos.

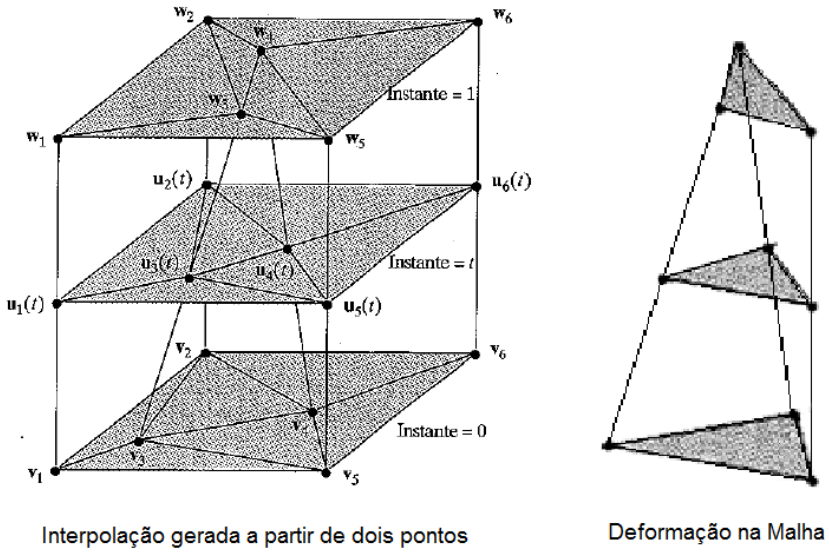


Figura 6 - Exemplo de Interpolação entre imagens

Figura 7 - Exemplo de utilização da Malha Deformável (WOLBERG, 1998)

2.2 MORFISMO PARALELO

Nesta seção será descrito um caso de implementação de morfismo em arquitetura paralela de uma GPU (*Graphics Processing Unit*).

O trabalho de Zamith *et al* (2009) propõe um algoritmo de morfismo paralelo executado em tempo real na GPU, com o objetivo de animar a aparência de um personagem 3D por meio da manipulação da sua textura. A técnica de morfismo utilizada é *warping* e o código foi escrito em CUDA, uma linguagem específica para programação em GPUs da Nvidia. A estratégia usada por Zamith *et al* (2009) busca processar todos os *pixels* da imagem ao mesmo tempo, visto que este é o caso ideal de processamento em GPUs.

Os dados de entrada são representados por cinco matrizes onde duas representam as imagens inicial e final, duas representam as linhas que guiam os *pixels* e a última matriz é utilizada para armazenar os valores da interpolação que permite a reconstrução da deformação da imagem inicial para a final, assim como da imagem final para a inicial.

Para contornar a restrição de banda que existe entre a CPU e a GPU, os autores propuseram a alocação dinâmica de recursos na memória da GPU no início da aplicação evitando a transferência de informações desnecessárias à medida que cada nova imagem é gerada.

O algoritmo segue basicamente quatro etapas:

- Alocação de memória na GPU;
- Transferência de dados da CPU para a GPU;
- Execução do *kernel*;
- Libertação da memória utilizada.

Os testes foram realizados em uma imagem RGBA de 256x256 *pixels* (Figura 8) com o objetivo de simular o envelhecimento facial de um personagem.

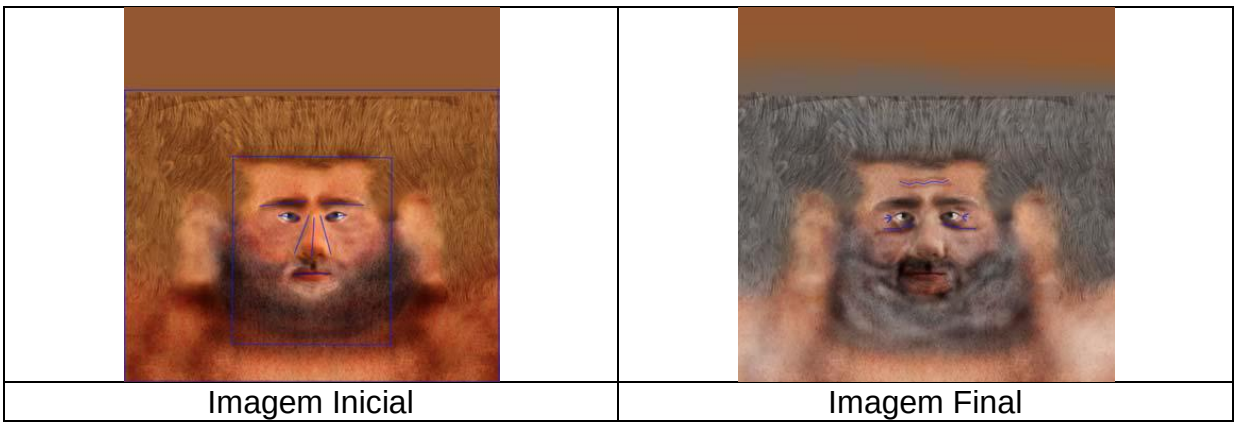


Figura 8 – Texturas utilizadas nos testes

Os resultados são apresentados na Tabela 2, que compara o tempo de processamento em milissegundos do algoritmo sequencial (CPU) e paralelo (GPU). A primeira coluna indica o número de pontos marcados na imagem; a segunda e terceira coluna mostram os tempos de interpolação e de morfismo da CPU e GPU, respectivamente, enquanto a última coluna apresenta o tempo total.

Tabela 2 – Tempos de Processamento na CPU e GPU

	Interpolação		**Morfismo**		**Total**	
#Seg	CPU	GPU	CPU	GPU	CPU	GPU
4	0.000001	0.000114	0.238294	0.010499	0.238295	0.01613
8	0.000001	0.000143	0.625339	0.020555	0.625340	0.020698
16	0.000002	0.000116	1.219.303	0.040778	1.219.305	0.040894
32	0.000003	0.000119	2.641.893	0.082539	2,641896	0,082658

Os resultados mostram que a CPU é mais rápida nos cálculos de interpolação, porem a GPU é muito mais rápida na hora do morfismo, sendo que é possível concluir que a GPU é a mais indicada para solucionar problemas computacionais embaraçosamente paralelos.

2.3 MÁQUINAS PARALELAS

2.3.1 Classificação de Arquiteturas Paralelas

Existem vários fatores que podem ser usados para classificar máquinas paralelas, entre eles, o tipo de processador, o acesso a memória e a sequência e o fluxo de instruções. A classificação mais famosa é a de Flynn (1972), que baseia-se nas diferentes formas de execução de um fluxo de instruções sobre um fluxo de dados. Segundo Flynn (1972), esses fluxos podem ser exclusivos ou múltiplos, e através da combinação entre eles, foram propostas quatro classes: SISD, MISD, SIMD e MIMD (ver Tabela 1).

Tabela 1 - Arquiteturas segundo Flynn (1972)

	SD (*Single Data*)	**MD (*Multiple Data*)**
SI (*Single Instruction*)	**SISD** – Máquinas Von Neumann convencionais.	**SIMD** – Máquinas *Array.*
MI (*Multiple Instruction*)	**MISD** – Sem representantes.	**MIMD** – Multiprocessadores e Multicomputadores.

2.3.2 SISD

A classe SISD (*Single Instruction Single Data*) tem como representante as Máquinas de Von Neumann convencionais, que possuem um único fluxo de instruções atuando sobre um único fluxo de dados. Na Figura 9, é possível observar o fluxograma desta arquitetura, onde a unidade de controle *C* repassa as instruções para a unidade de processamento *P*, que por sua vez processa os dados da memória (*M*) (ROSE e NAVAUX, 2003).

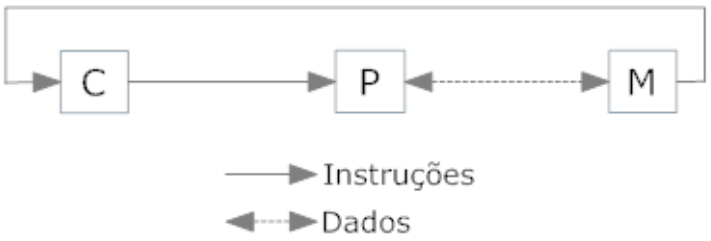


Figura 9 - Fluxograma da Arquitetura SISD (ROSE e NAVAUX, 2003)

2.3.3 MISD

MISD (*Multiple Instructions Single Data*) são máquinas que possuem diferentes fluxos de instruções operando sobre o mesmo fluxo de dados. O fluxograma desta arquitetura pode ser visualizado na Figura 10, onde cada unidade de controle *C* recebe um fluxo de instruções diferente, que atuam sobre o mesmo fluxo de dados, através das unidades de processamento *P*. Segundo Rose (2003), a classe MISD não faz nenhum sentido, além de ser impraticável.

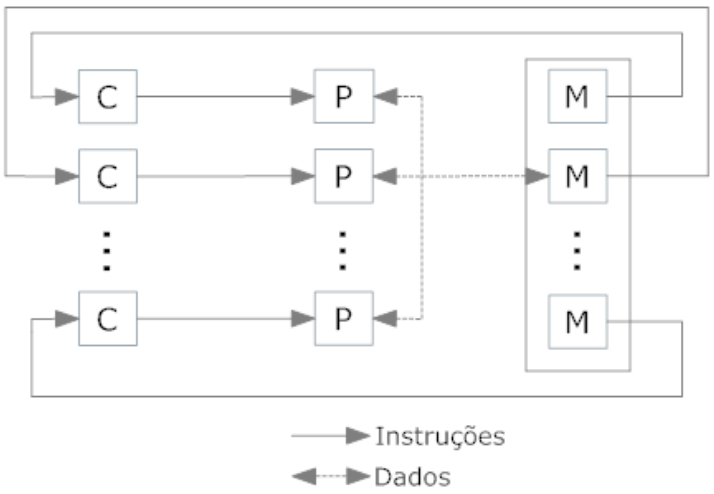


Figura 10 - Fluxograma da Arquitetura MISD (ROSE e NAVAUX, 2003)

2.3.4 SIMD

Na arquitetura SIMD (*Single Instruction Multiple Data*), existe apenas uma unidade de controle *C*, que repassa as instruções para *n* processadores *P*. Cada processador executará suas instruções sobre um determinado fluxo de dados de forma síncrona (ver Figura 11). Tecnicamente, pode-se perceber que o mesmo programa está sendo executado sobre diferentes dados, que faz com que o princípio

de execução da arquitetura SIMD assemelhe-se bastante ao paradigma da execução sequencial.

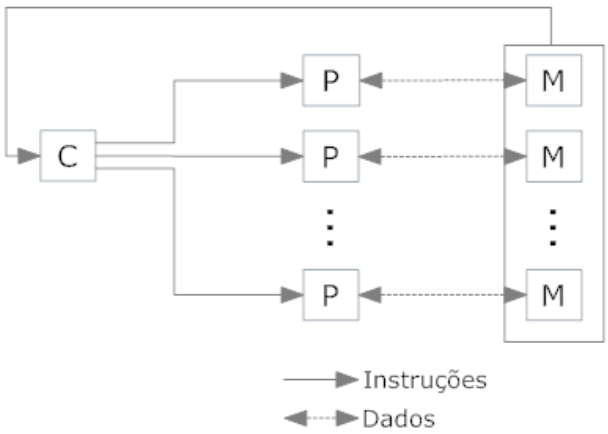


Figura 11 - Fluxograma da Arquitetura SIMD (ROSE e NAVAUX, 2003)

2.3.5 MIMD

Na arquitetura MIMD (*Multiple Instruction Multiple Data*), diversos processadores trabalham em paralelo, processando suas tarefas concorrentemente de forma assíncrona para, num intervalo de tempo, concluírem a tarefa. Esses processadores geralmente são gerenciados pelo mesmo sistema operacional, além de serem idênticos. Resumindo, “um multiprocessador é um computador que possui vários processadores que se comunicam e cooperam para resolver uma dada tarefa” (ROSE e NAVAUX, 2003).

Em uma máquina MIMD, cada unidade de controle *C* repassa para seu processador *P* um fluxo de instruções, para que seja executado sobre um fluxo de dados próprio (ver Figura 12). Dessa forma, cada processador executa o seu próprio programa sobre seus próprios dados de forma assíncrona. Basicamente, qualquer computador que possua dois ou mais núcleos de processamento se enquadra nesta arquitetura.

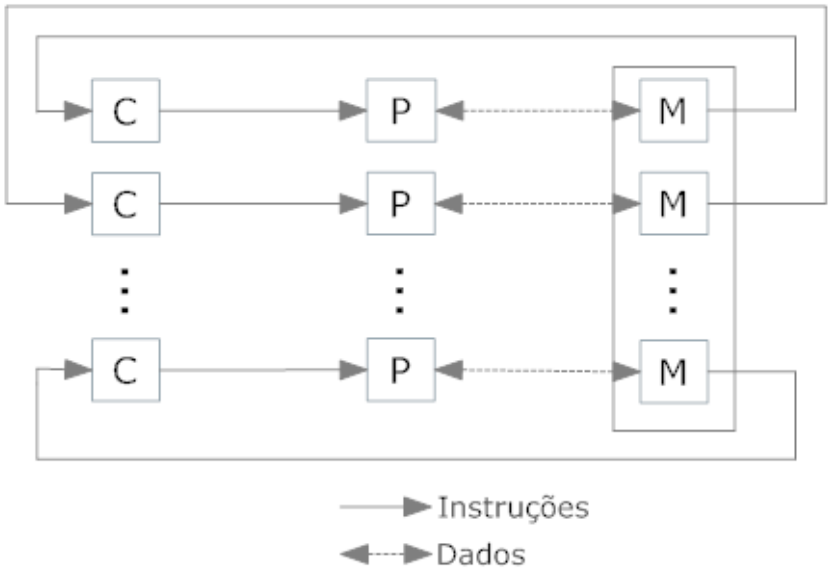


Figura 12 - Fluxograma da Arquitetura MIMD (ROSE e NAVAUX, 2003)

A classe de máquinas MIMD pode ainda, ser subdividida em duas subclasses de acordo com o acesso a memória. Máquinas com memória compartilhada são conhecidas como multiprocessadores, enquanto máquinas que possuam memória distribuída e privada são chamadas de multicomputadores.

2.3.5.1 Multiprocessadores

Máquinas multiprocessadas possuem um único espaço de endereçamento, que é compartilhado por todos os processadores. Como exemplo tem-se os computadores domésticos equipados com processadores *Intel Core Duo* ou *AMD Phenom II X2*, que se enquadram na classificação de arquiteturas MIMD Assíncronas de memória compartilhada.

2.3.5.2 Multicomputadores

Em multicomputadores, cada máquina tem sua própria memória, e os processadores se comunicam por meio da troca de mensagens. Um exemplo dessa arquitetura seria dois ou mais computadores interligados por um barramento *Ethernet*.

2.3.6 Tipos de Concorrência

As principais formas de concorrência em arquiteturas são a temporal e a de recursos. Na concorrência temporal, ocorre uma sobreposição de execuções no tempo, gerando um ganho no desempenho final de processamento. Por outro lado, na concorrência de recursos, também chamada de espacial, a arquitetura possui vários processadores ou elementos de processamento que trabalham em paralelo na execução das tarefas que compõe o processamento (ROSE e NAVAUX, 2003). Esse tipo de concorrência pode ser síncrona, que resulta em arquiteturas SIMD, e pode ser também assíncrona, que resulta em arquiteturas MIMD.

2.4 GRANULARIDADE

Granularidade é a relação entre a quantidade de instruções executadas e o número de sincronizações entre as *threads* ou processos durante a execução. processos que necessitam de muita sincronização possuem granularidade fina, enquanto processos que possuem pouca sincronização possuem granularidade grossa (MORETTI, 1997). A granularidade pode ser vista também como a quantidade de pedaços que a informação foi cortada para ser processada. Por exemplo, supondo que uma imagem de baixa resolução precise passar por um determinado algoritmo de processamento, provavelmente ela será processada mais rapidamente em um núcleo, do que se fosse cortada e enviada a 32 núcleos de processamento, devido a alta granularidade.

A granularidade é um ponto importante na paralelização de algoritmos, pois processos que possuem granularidade fina, consequentemente possuem alta taxa de sincronização, e por isso sofrem perda de desempenho. Para se atingir o máximo de desempenho em aplicações paralelas, a sincronização deve ocorrer com a menor frequência possível.

2.5 SINCRONIZAÇÃO

Sincronização é a troca de informações entre *threads* ou processos. Geralmente afeta negativamente o desempenho das aplicações, pois muitas vezes algumas *threads* ou processos podem ficar ociosos, esperando as informações dos outros. Por isso, é necessário ter um bom sistema de balanceamento de cargas, para evitar que eles fiquem ociosos, e evitar ao máximo a sincronização.

2.6 PROCESSOS

Processo é a abstração de um programa em execução e possui informações sobre registradores, variáveis e valores do contador do programa (TANENBAUM e WOODHULL, 2000). Basicamente, um processo é uma entidade independente e possui um programa, entrada, saída e um estado. É importante salientar que um programa pode estar ligado a vários processos, assim como um processo pode estar ligado a várias threads. O responsável por administrar os vários processos na máquina é o sistema operacional, que alterna a CPU entre eles através de um algoritmo de escalonamento.

No geral, a criação e destruição, e também a comunicação entre processos é mais lenta que a criação, destruição e comunicação de *threads*, visto que *threads* utilizam o mesmo espaço de memória, além de muitas vezes o sistema operacional nem tomar conhecimento das mesmas, enquanto a maioria das atividades de um processo realiza chamadas de sistema.

A vantagem de utilizar processos em vez de *threads* está no fato de que se ocorrer algum erro em algum dos processos de um programa, somente aquele precisa ser reiniciado, ao invés de todo o programa. Na Figura 13 é possível visualizar um programa que utiliza vários processos em vez de várias *threads.* No *Google Chrome*, cada aba do navegador é um processo. Se por acaso alguma aba parar de responder, apenas ela será afetada enquanto as outras continuaram trabalhando.

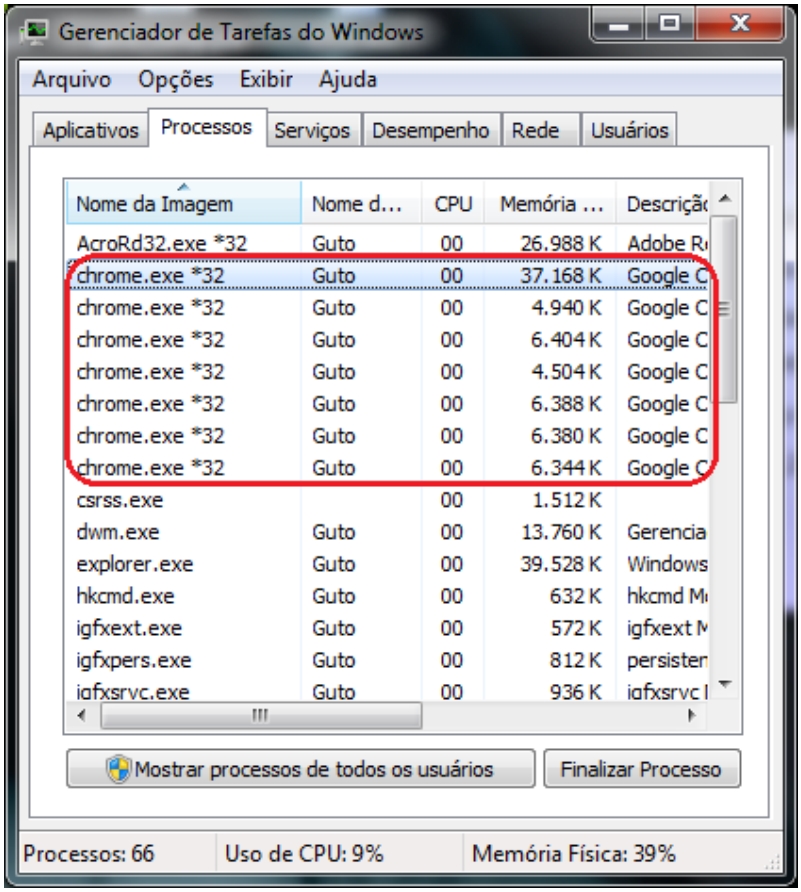


Figura 13 - Cada nova aba no *Google Chrome* é um novo processo

2.7 *THREADS*

Threads, ou processos leves, são fluxos de execução dentro de um processo. Os sistemas operacionais modernos permitem que um processo tenha associado a ele vários fluxos de execução, fato denominado de *multithreading*. Todas as threads de um processo compartilham dos mesmos recursos do processo, assim como o mesmo espaço de endereçamento (TANENBAUM e WOODHULL, 2000) (ver Figura 14).

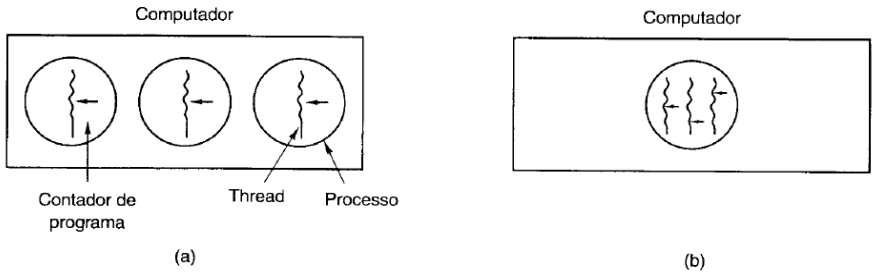


Figura 14 - (a) Três processos, cada um com uma *thread*. (b) Um processo com três *threads*.

2.8 CONSIDERAÇÕES PARCIAIS

Este trabalho se aterá apenas à paralelização do algoritmo em máquinas MIMD Assíncronas de memória compartilhada, ou seja, multiprocessadas, pois é uma arquitetura de baixo custo e bem difundida.

É através da concorrência espacial assíncrona que este trabalho visa melhorar o desempenho do algoritmo de morfismo, visto que, existe uma relação entre a quantidade de núcleos de processamento que trabalham em conjunto e o desempenho, sendo que cada núcleo poderá executar suas tarefas concorrente e sem necessidade de sincronização.

4 ALGORITMO DE MORFISMO IMPLEMENTADO

Aqui será apresentado o algoritmo de morfismo baseado na descrição de Anton e Rorres (2005), que expõe os passos a serem seguidos de forma clara e objetiva.

Primeiramente é preciso escolher os elementos chave da imagem que pretende-se deformar. Essa marcação de pontos pode seguir outros critérios também, de acordo com o efeito desejado. Para isso, deve-se marcar *n* pontos v_1, v_2, v_3, ..., v_n na imagem, que serão chamados de pontos de vértice (ver Figura 15).

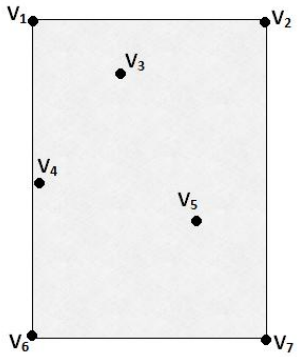


Figura 15 - Marcação de Pontos na Figura

Após escolher os pontos, é preciso executar uma triangulação da imagem (ver Figura 16), que consiste em traçar retas entre os pontos de vértice respeitando as seguintes regras:

1. As retas devem formar os lados de uma coleção de triângulos;
2. As retas não podem se cruzar;
3. Cada ponto de vértice é o vértice de pelo menos um triângulo;
4. A união dos triângulos é o retângulo. Essa regra obriga que os quatro cantos do retângulo sejam pontos de vértice;
5. A coleção de triângulos é máxima, ou seja, não restam vértices para conectar.

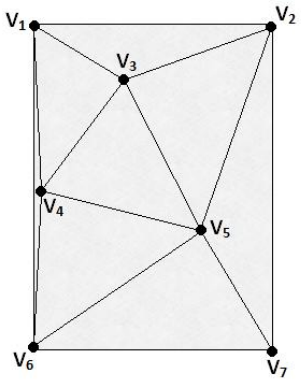


Figura 16 - Exemplo de triangulação correta

A quantidade de triângulos formados pelos *n* vértices é dada pela equação

$$m = 2n - 2 - k \qquad (4.1)$$

onde *k* é o número de vértices que se encontram nas fronteiras da imagem, incluindo os 4 pontos presentes nos cantos.

A deformação na imagem é dada pelo movimento dos *n* vértices para as novas posições $w_1, w_2, w_3, ..., w_n$ (ver Figura 17), de acordo com as mudanças que deseja-se efetuar (ANTON e RORRES, 2005).

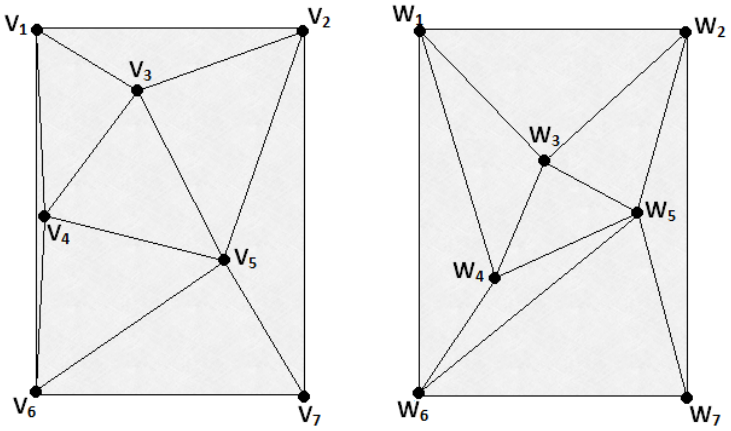


Figura 17 - Deformação entre imagens

Existem duas restrições que envolvem a movimentação dos pontos. São elas:

1. Os quatro pontos que definem o retângulo devem permanecer fixos, da mesma forma que todos os pontos situados nas fronteiras da imagem também devem permanecer, ou podem no máximo mover-se para outro lugar no mesmo lado do retângulo. Todos os demais pontos devem permanecer dentro do retângulo.

2. Os triângulos formados inicialmente não podem ficar sobrepostos depois de executada a movimentação dos vértices, ou seja, os pontos não podem se movimentar de uma forma que gere cruzamento das linhas dos polígonos, como mostra a Figura 18.

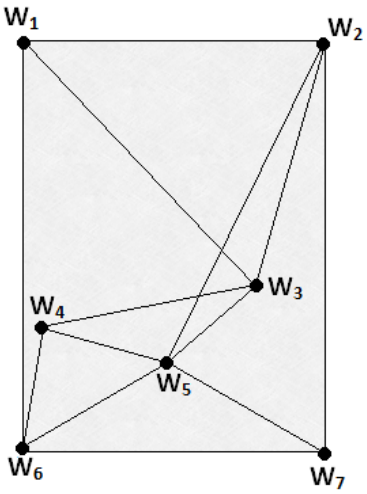


Figura 18 - Exemplo de movimentação não permitida (Cruzamento de linhas)

A primeira restrição garante que a imagem continuará com a forma retangular inicial. A segunda restrição garante que a imagem resultante não tenha um resultado artificial (ANTON e RORRES, 2005).

A Figura 19 mostra duas fotografias da mesma mulher, sendo que a *Imagem inicial* foi tirada 20 anos antes da *Imagem Final*.

Figura 19 - Duas fotografias com intervalo de 20 anos

A partir delas, foram geradas duas triangulações, uma baseada na *Imagem Inicial* e outra baseada na *Imagem Final* (ver Figura 20). Foram usados 94 vértices, que geraram 179 triângulos. Note que os pontos foram cuidadosamente escolhidos ao longo das características essenciais da imagem, como olhos, nariz, boca e cabelo (ANTON e RORRES, 2005).

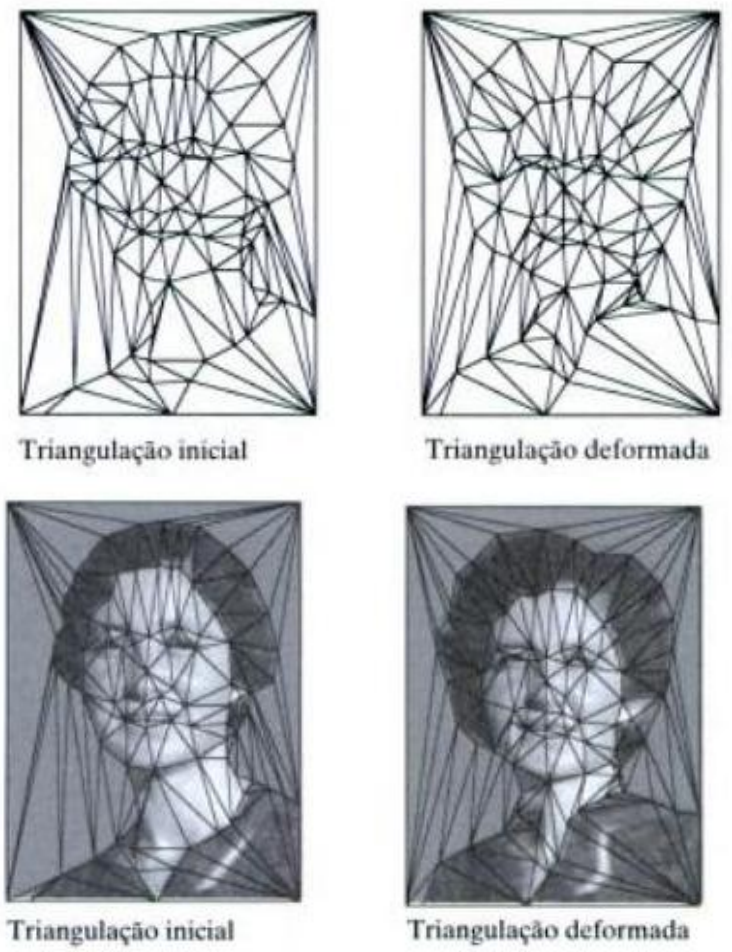


Figura 20 - Triangulação nas Imagens

A partir dessas duas triangulações, a imagem inicial foi deformada, e o resultado pode ser visto na Figura 21. É importante ressaltar que ainda não ocorreu o morfismo em si, mas sim apenas uma deformação da *Imagem Inicial*, que tomou a forma e as feições da *Imagem Final,* pois a *Imagem deformada* representa a mulher no seu formato mais velho, mas usando os níveis de cinza de quando era mais jovem (ANTON e RORRES, 2005).

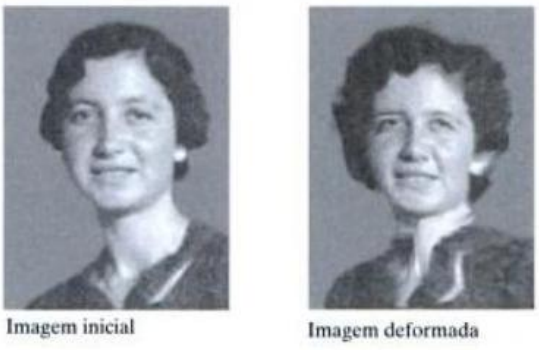


Figura 21 - Resultado da Deformação

Deformações dependentes do tempo são geradas quando os vértices de uma imagem inicial são movidos continuamente durante um período de tempo da sua posição original até posições finais especificadas. Isso gera uma animação na qual uma imagem inicial é deformada continuamente até chegar em um estado final. A unidade de tempo escolhida varia de 0 a 1, onde $t = 0$ corresponde a imagem inicial e $t = 1$ corresponde a imagem final. Para mover um vértice do instante de tempo 0 para o instante de tempo 1, usam-se retas geradas por uma interpolação linear ligando as posições iniciais com as posições finais (ANTON e RORRES, 2005).

A Figura 22 mostra uma deformação dependente do tempo com 5 valores entre 0 e 1.

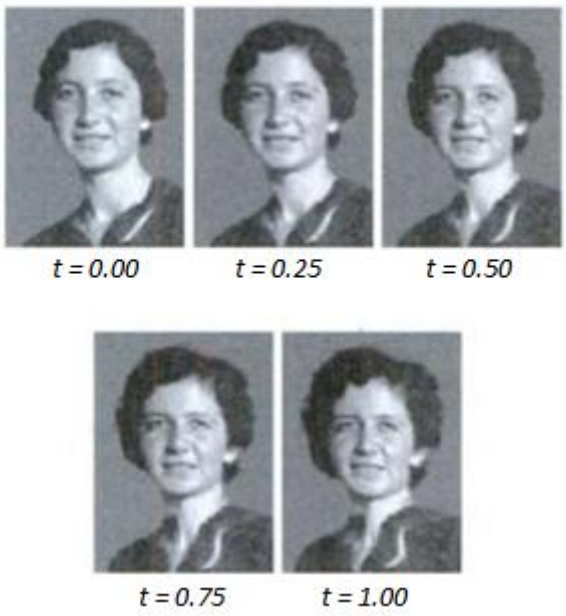


Figura 22 - Exemplo de Deformação no tempo

O morfismo pode ser definido como a combinação de duas deformações de duas imagens diferentes, que associam características correspondentes entre elas. Então escolhemos uma delas para ser chamada de imagem inicial, fazendo com que a outra seja chamada de imagem final. Depois disso, gera-se uma deformação de $t = 0$ a $t = 1$ onde a imagem inicial será deformada para a forma da imagem final. Após isso, gera-se uma deformação de $t = 0$ a $t = 1$ na qual a imagem final será deformada para a forma da imagem inicial. Por fim, para gerar o morfismo propriamente dito, em cada instante t entre 0 e 1 aplica-se uma média ponderada dos níveis de cinza das duas imagens. Essa média ponderada possui o mesmo conceito do *mixing*, só que em vez de aplicar o *mixing* a toda a imagem, ele é

aplicado apenas dentro de cada região deformada. Resumidamente, pode-se afirmar que o morfismo é a junção de duas técnicas, a técnica de deformação e a técnica de *mixing*. A Figura 23 mostra a versão com morfismo da Figura 22.

Figura 23 - Exemplo de morfismo no tempo

Os passos para produzir o morfismo são apresentados a seguir:

1. Dados uma imagem inicial p_0 e uma imagem final p_1, posicionam-se n pontos de vértice $v_1, v_2, ..., v_n$ nas principais características da imagem inicial;
2. Posicionam-se n pontos de vértice correspondentes a $w_1, w_2, ..., w_n$ na imagem final nas principais características;
3. Desenha-se a malha, traçando retas entre os pontos de vértice;
4. Para cada instante de tempo t entre 0 e 1, encontra-se a posição dos vértices $u_1(t), u_2(t), ..., u_n(t)$ por meio de uma interpolação linear.
5. Triangula-se o morfismo da imagem do instante t de maneira similar as triangulações das imagens inicial e final;

6. Determina-se o mapeamento (morfismo) entre a imagem inicial e imagem no instante *t* por meio da técnica de mapeamento (GONZALEZ e WOODS, 2000) descrita na próxima sessão.
7. Finalmente determina-se a coloração da imagem $p_I\ (u)$ no ponto *u* do morfismo da imagem usando a equação 4.2.

$$p_I(u) = (1 - t)p_0(v) + tp_1(w) \qquad (4.2)$$

A Figura 24 exemplifica os passos descritos acima.

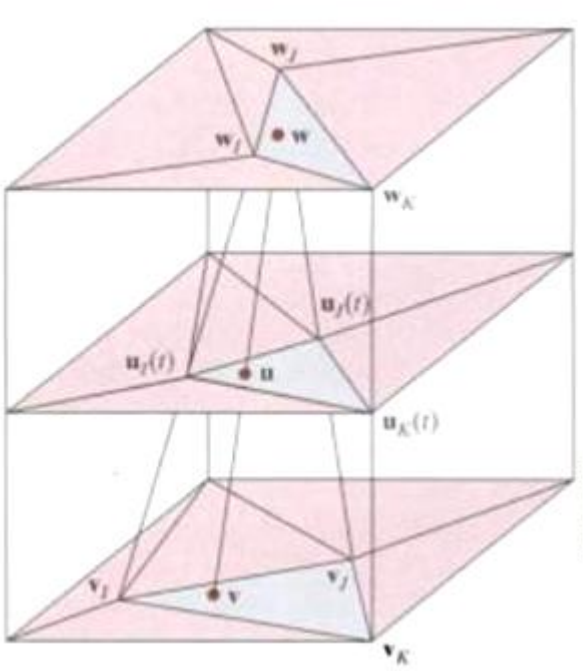


Figura 24 - Exemplificação do algoritmo de morfismo

O último passo é o que difere deformações de morfismos. O peso das cores depende do fragmento de distância que os vértices se encontram das suas origens. Por exemplo, se o vértice se movimentou um quarto do caminho até seu destino, ou seja, *t* = 0,25, então deve-se usar três quartos do nível de cinza da imagem inicial e apenas um quarto de cinza da imagem final. Assim, conforme o tempo avança, além da forma da imagem inicial mudar gradativamente para a forma da imagem final, as cores também vão mudando gradativamente, até chegarem nas cores da imagem final (ANTON e RORRES, 2005).

4.1 TRANSFORMAÇÃO ESPACIAL DE MAPEAMENTO

As transformações geométricas são operações aplicadas sobre imagens e geralmente modificam a relação espacial entre os *pixels* (GONZALEZ e WOODS, 2000), e são usadas para os mais diversos fins como restauração, distorção, ampliação, redução e rotação. Neste trabalho é importante entender o conceito de mapeamento.

Transformação espacial de mapeamento é um tipo de transformação geométrica, e pode ser entendida como uma função que gera uma relação entre todos os *pixels* de uma imagem de entrada e a sua saída. Por meio de uma determinada função, os *pixels* são deslocados, gerando uma distorção na imagem (Figura 25).

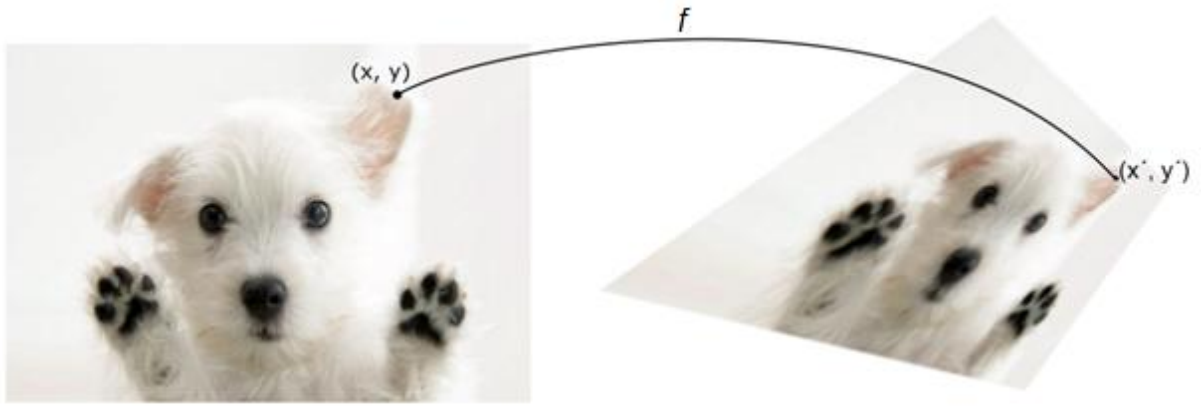


Figura 25 - Exemplo de Mapeamento

Supondo que uma imagem *f*, com coordenadas de *pixel (x, y)*, sofra uma distorção geométrica *g* com coordenadas *(x', y')*. Essa transformação pode ser expressa como

$$x' = r(x, y)$$
$$y' = s(x, y) \quad (4.3)$$

em que *r(x, y)* e *s(x, y)* representam as transformações espaciais que geram a imagem de saída *g(x', y')*. Para gerar essa função de realocação de *pixels*, é necessário usar pontos de amarração, que são um subconjunto de *pixels* cuja

posição é conhecida na imagem de entrada e na de saída, que no caso podem ser os vértices de um polígono.

Ao utilizar os vértices dos dois quadriláteros, temos 8 pontos de amarração, duas equações bilineares e 8 coeficientes. Esses 8 coeficientes precisam ser definidos, para que possam ser aplicados a todos os *pixels* da imagem. A equação da transformação fica da seguinte forma:

$$r(x, y) = c_1x + c_2y + c_3xy + c_4$$
$$s(x, y) = c_5x + c_6y + c_7xy + c_8 \quad (4.4)$$

Logo,

$$x' = c_1x + c_2y + c_3xy + c_4$$
$$y' = c_5x + c_6y + c_7xy + c_8. \quad (4.5)$$

Os coeficientes c_i, onde $i = 1, 2, ..., 8$ podem ser encontrados através da resolução de um sistema linear. Esses coeficientes constituem a combinação linear que é utilizada para gerar a transformação espacial (GONZALEZ e WOODS, 2000).

No caso de utilização de triângulos em invés de quadriláteros, a equação usada é

$$x' = c_1x + c_2y + c_3$$
$$y' = c_4x + c_5y + c_6. \quad (4.6)$$

Pois existem apenas 6 pontos de amarração e 6 coeficientes à descobrir.

O cálculo para descoberta dos coeficientes é efetuado em dois passos: um para os pontos que representam as linhas (x) e outro para as colunas (y).

Por exemplo, temos os seguintes conjuntos de vértices:

$$A = [x_1, y_1], [x_2, y_2], [x_3, y_3]$$

$$B = [x_4, y_4], [x_5, y_5], [x_6, y_6]$$

onde o conjunto A representam os vértices de um triângulo em uma imagem qualquer e B representam os vértices do mesmo triângulo, só que em outra imagem e com valores diferentes.

Então os sistemas lineares são montados da seguinte forma:

$$\begin{aligned} A[x_1]*c_1 + A[y_1]*c_2 + 1*c_3 &= B[x_4] \\ A[x_2]*c_1 + A[y_2]*c_2 + 1*c_3 &= B[x_5] \\ A[x_3]*c_1 + A[y_3]*c_2 + 1*c_3 &= B[x_6] \end{aligned} \tag{4.7}$$

$$\begin{aligned} A[x_1]*c_4 + A[y_1]*c_5 + 1*c_6 &= B[y_4] \\ A[x_2]*c_4 + A[y_2]*c_5 + 1*c_6 &= B[y_5] \\ A[x_3]*c_4 + A[y_3]*c_5 + 1*c_6 &= B[y_6] \end{aligned} \tag{4.8}$$

onde c_1, c_2 e c_3 são usados para realizar o mapeamento das coordenadas que representam as linhas e c_4, c_5 e c_6 são usados para realizar o mapeamento das coordenadas que representam as colunas. Após resolver os dois sistemas lineares separadamente, descobrem-se os coeficientes, e então deve-se realizar o caminho inverso, ou seja, pegar um ponto dentro do triângulo A que possui a posição desconhecida em B (x', y'), e aplicar a equação 4.9 para descobrir a sua posição:

$$\begin{aligned} x' &= c_1x + c_2y + c_3 \\ y' &= c_4x + c_5y + c_6. \end{aligned} \tag{4.9}$$

Após isso, o *pixel* com as coordenadas [x', y'] pode receber a cor do *pixel* representado por [x, y].

Para resolução destes sistemas lineares, foi utilizada o Método de Eliminação de Gauss.

5 PROJETO DO SISTEMA

O algoritmo receberá duas imagens em tons de cinza, sendo que uma é definida como imagem inicial e a outra como imagem final. Após isso, é necessário realizar a marcação de pontos nas imagens, buscando apontar as principais características. Essa marcação deverá seguir um padrão, visto que precisa existir uma correspondência entre os pontos marcados nas imagens inicial e final. Esses pontos são adicionados a um vetor que faz a referida correspondência por meio de seu índice. O padrão de entradas de pontos adota a seguinte sequência:

1. Marcar um ponto na imagem inicial;
2. Marcar um ponto na imagem final;
3. Repetir os passos 1 – 2 até que todos os pontos desejados tenham sido marcados.

A malha, formada por triângulos, é gerada automaticamente por meio do algoritmo de Triangulação de Delaunay, que é provido pela biblioteca "*matplotlib.tri"* (HUNTER; DALE; DROETTBOOM, 2010), visto que cada imagem precisa de um número elevado de pontos. Por exemplo, se existirem 30 pontos na imagem, serão gerados 54 triângulos. Caso fosse feito manualmente, a chance de ocorrer erros na criação dos triângulos seria muito elevada, além de tornar-se uma tarefa demorada. Em seguida, é possível definir o tamanho da sequência de morfismo, ou seja, quantas imagens intermediárias devem ser criadas. Finalmente, o algoritmo é executado e as imagens são geradas e, em seguida, gravadas no computador.

5.1 ENTRADA DO SISTEMA

O sistema possui como entrada duas imagens, que posteriormente são segmentadas em regiões pela malha de pontos. Os parâmetros como a quantidade de processos e também a quantidade imagens que o morfismo deve gerar são inseridos em um *script python*, junto com os dados gerados na triangulação. Um exemplo da *interface* do sistema pode ser observado na Figura 26, que mostra duas imagens segmentadas pela malha de pontos.

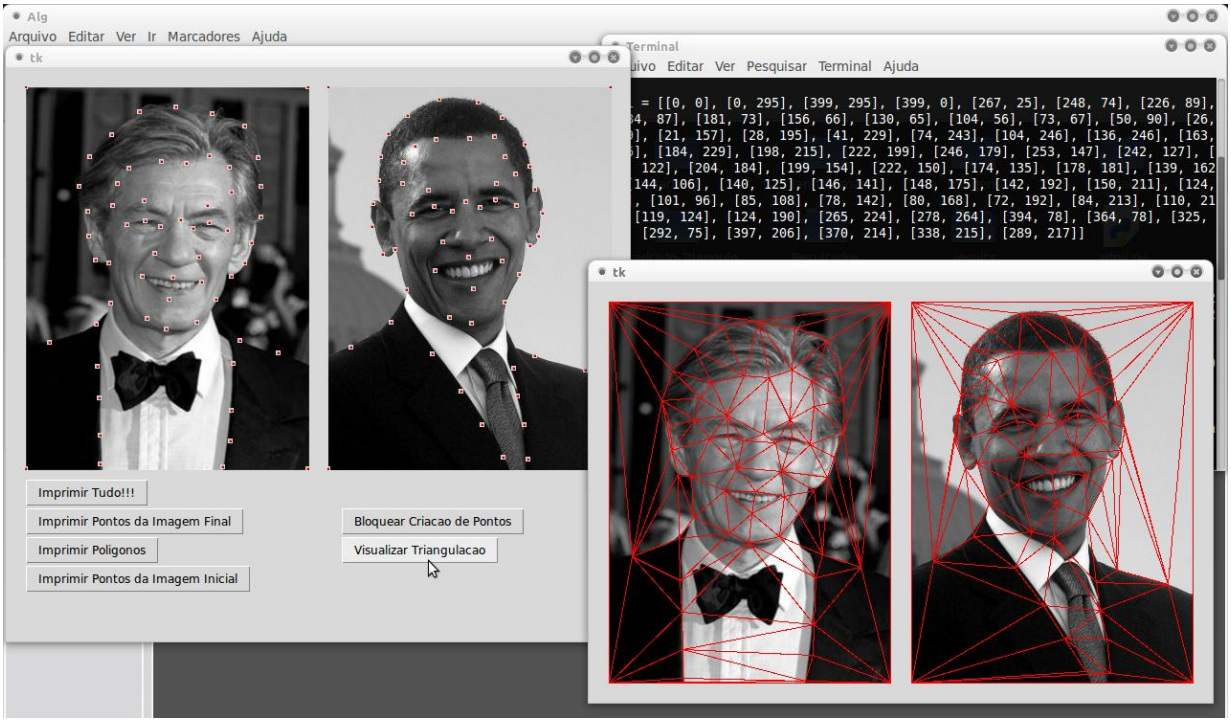


Figura 26 - Interface do programa

Existem quatro pontos essenciais criados automaticamente, localizados nos cantos da imagem e que são responsáveis por manter as imagens com seu formato retangular original.

5.2 LINGUAGEM

Nesta seção são descritas as características da linguagem escolhida e algumas bibliotecas importantes na implementação do algoritmo, que levaram à escolha da mesma.

5.2.1 Python

Python é uma linguagem interpretada, de alto nível com tipagem dinâmica, multiparadigma e está sob licença *open source* (PYTHON, 2010). As principais características e motivações que levam a usar esta linguagem são: portabilidade, bibliotecas de suporte e integração de componentes (LUTZ; ASCHER, 2007).

Python tem uma vasta biblioteca padrão, com um grande conjunto de funcionalidades pré-compiladas, que vão deste processamento assíncrono até o suporte a arquivos *ZIP* (PYTHON, 2010). Além disso, existem muitas bibliotecas independentes e livres sendo desenvolvidas, como a *ia636* (SILVA, 2003), *matplotlib* (HUNTER; DALE; DROETTBOOM, 2010) e a *numpy* (NUMPY, 2010) que são utilizadas no trabalho.

O interpretador do Python não dá suporte real a programação paralela por meio de *threads*. Para dar suporte as *threads,* foi criado um bloqueio global do interpretador (*Global Interpreter Lock*), chamado de GIL, que deve ser feito pela *thread* para poder acessar com segurança os objetos do Python, mas que causa perda de desempenho. Sem esse bloqueio, as mais simples operações poderiam causar inconsistência na execução, como por exemplo, no caso de duas *threads* incrementarem uma variável qualquer, mas essa variável receber o incremento de apenas uma das threads (PYTHON, 2010).

Por esse motivo, são utilizados processos ao invés de *threads*. Eles são criados por meio da biblioteca padrão *multiprocessing*.

5.2.1.1 *Ia636*

O módulo *ia636* (SILVA, 2003) é uma caixa de ferramentas para processamento de imagens. O módulo inclui funções para criação e manipulação de imagens, leitura de arquivos de imagens, manipulação de contraste, processamento de cor, manipulações geométricas, transformações, filtragem, técnicas de *thresholding* automático, aproximações de meio-tom entre outras.

5.2.1.2 *Matplotlib*

Matplotlib (HUNTER; DALE; DROETTBOOM, 2010) é uma biblioteca usada para gerar gráficos, histogramas, espectros de potência, gráficos de barras entre outros. Um módulo interessante desta biblioteca é o *matplotlib.nxutils*, que possui funções de propósito geral, como por exemplo de geometria computacional, que não se encontram na biblioteca *numpy*.

5.2.1.3 *Numpy*

Numpy (NUMPY, 2010) é um biblioteca voltada para computação científica, que possui funcionalidades como suporte a *arrays* multidimensionais, ferramentas para integração com C/C++, funções de álgebra linear e transformada de Fourier.

5.2.2 Considerações Parciais

Pelos motivos apresentados até aqui, o algoritmo de morfismo paralelo proposto neste trabalho será desenvolvido em Python, com a ajuda das bibliotecas citadas anteriormente, como a *Matplotlib, Ia636, Numpy*, etc. A paralelização será feita com a ajuda da biblioteca padrão *multiprocessing,* e a plataforma base de desenvolvimento será Linux.

5.3 ALGORITMO

O fluxograma do algoritmo pode ser visualizado na Figura 27.

O algoritmo de morfismo receberá como entrada duas imagens já definidas pela malha de pontos. Durante a marcação, todos os pontos serão indexados e rotulados por meio da entrada padrão já explicada, para que seja possível fazer a correspondência entre eles.

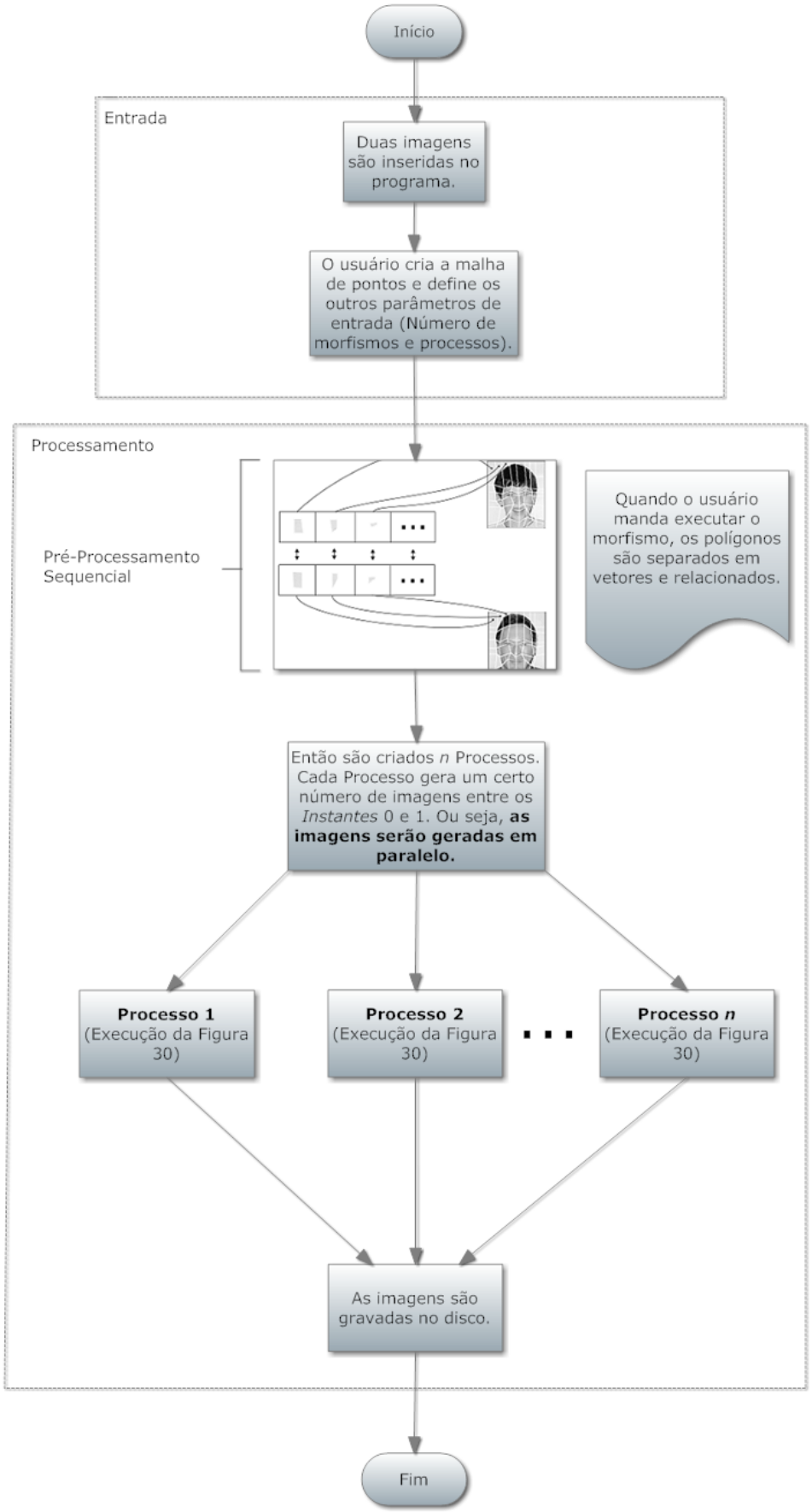


Figura 27 - Fluxograma de execução do algoritmo

Como o algoritmo possui as informações de quais pontos estão ligados entre si, por meio da posição dos vetores, ele pode separar as regiões, e colocá-las em dois vetores, onde um vetor será para as regiões da imagem inicial e o outro vetor será para as regiões da imagem final. A posição de cada elemento no vetor de regiões da imagem inicial irá corresponder exatamente a mesma posição e região do vetor de elementos da imagem final, como pode ser visto na Figura 28.

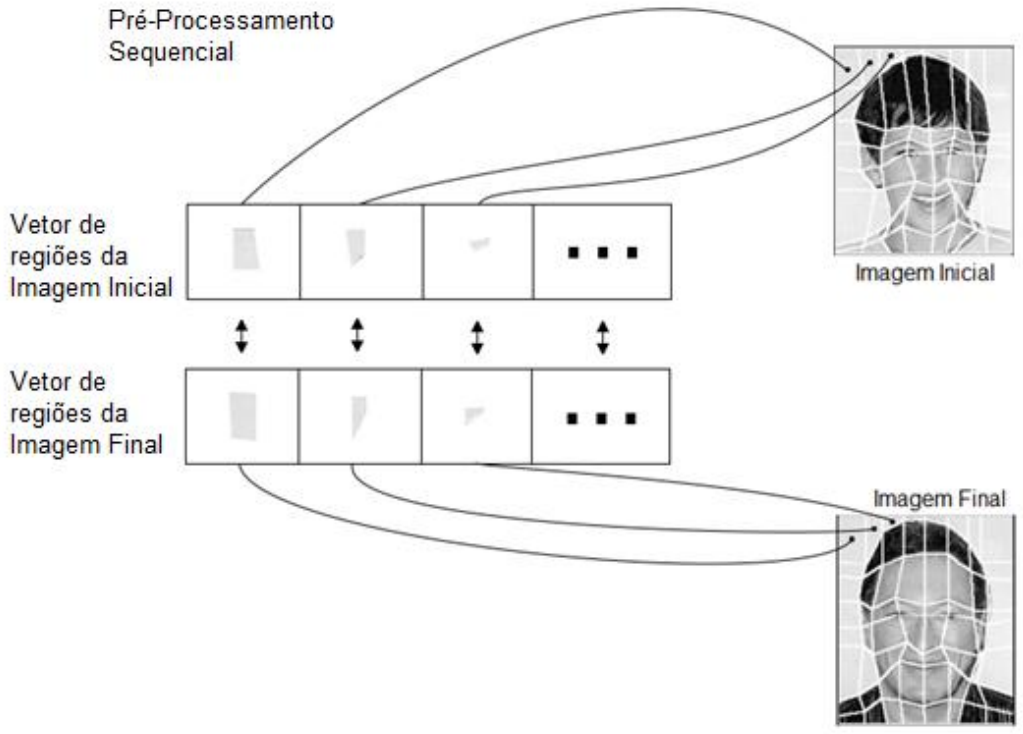


Figura 28 - Relação entre as imagens inicial e final

Por exemplo, analisando a Figura 29 temos seis regiões distintas. As regiões no *Instante* 0 serão colocadas no vetor v_i, enquanto as regiões no *Instante* 1 serão colocadas no vetor w_i. O vetor u_i será a saída do algoritmo. Então se, por exemplo, o algoritmo for analisar a região na posição 0 do vetor v_i, e essa posição corresponder a região escura da imagem inicial mostrada na Figura 29, consequentemente a posição 0 no vetor w_i também corresponderá a região escura da imagem final.

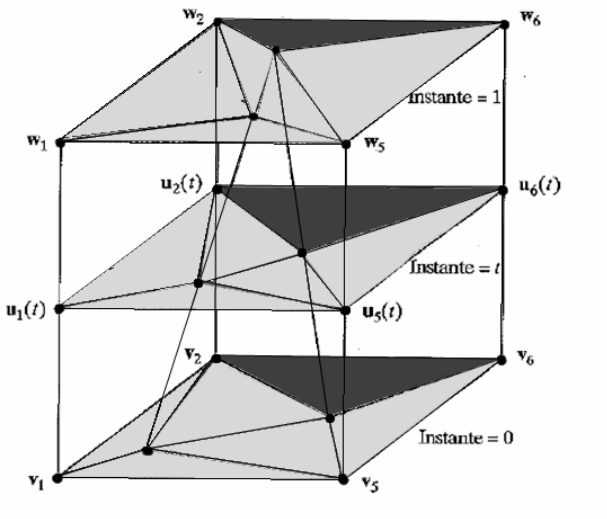


Figura 29 - Regiões independentes dentro da imagem

Após a criação dos vetores, o algoritmo criará uma quantidade definida de processos que atuarão em paralelo. Cada processo será incumbido de gerar *n* imagens no *Instante u(t)* (Figura 27). A posição no tempo-espaço do *Instante u(t)* será definida com uma simples interpolação linear dos pontos da imagem inicial e final (*Instante* = *t* da Figura 29). Então, será executada a função de mapeamento e posteriormente a função de interpolação de cores, para preencher possíveis buracos. O retorno desses processos será um vetor com as regiões segmentadas da imagem no *Instante u(t)* (ver Figura 30). Após todos os processos terminarem, o algoritmo grava as imagens no disco, terminando assim o morfismo.

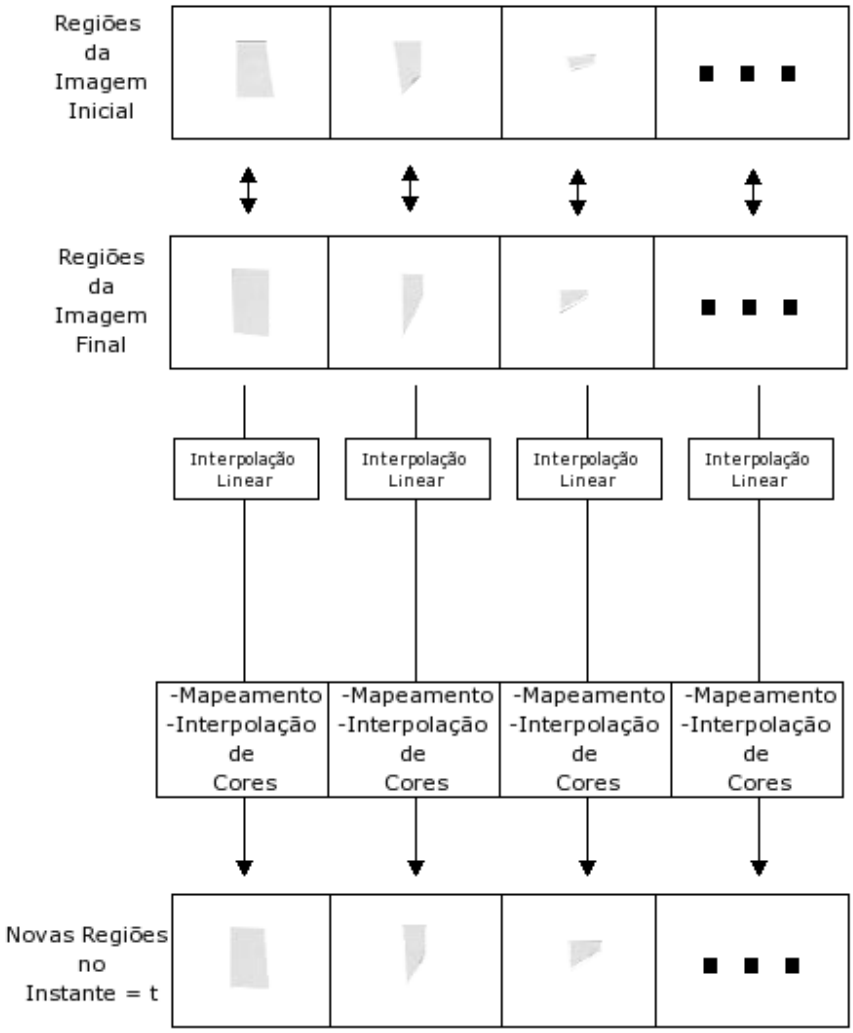


Figura 30 - Execução de um dos processos do algoritmo

Como comentado no Capítulo 4, o método de Eliminação de Gauss foi usado para determinação das constantes utilizadas na função de mapeamento.

5.4 MÉTRICAS DE AVALIAÇÃO

Nesta seção estão descritas as métricas utilizadas neste trabalho.

5.4.1 *Speedup*

Speedup é usado como parâmetro de avaliação do desempenho de algoritmos paralelos em relação aos sequenciais. Basicamente, ele mede o ganho de velocidade do algoritmo, comparando o tempo de execução do algoritmo em um único núcleo de processamento com o tempo de execução em *n* núcleos de processamento (MORETTI, 1997). A fórmula para calcular o *Speedup* é a seguinte:

$$speedup_{(n)} = \frac{tempoExecução_{(serial)}}{tempoExecução_{(paralelo)}} \quad (5.1)$$

O *Speedup* ideal possui a forma do gráfico da Figura 31, que mostra a relação entre o ganho de desempenho e o número de processadores.

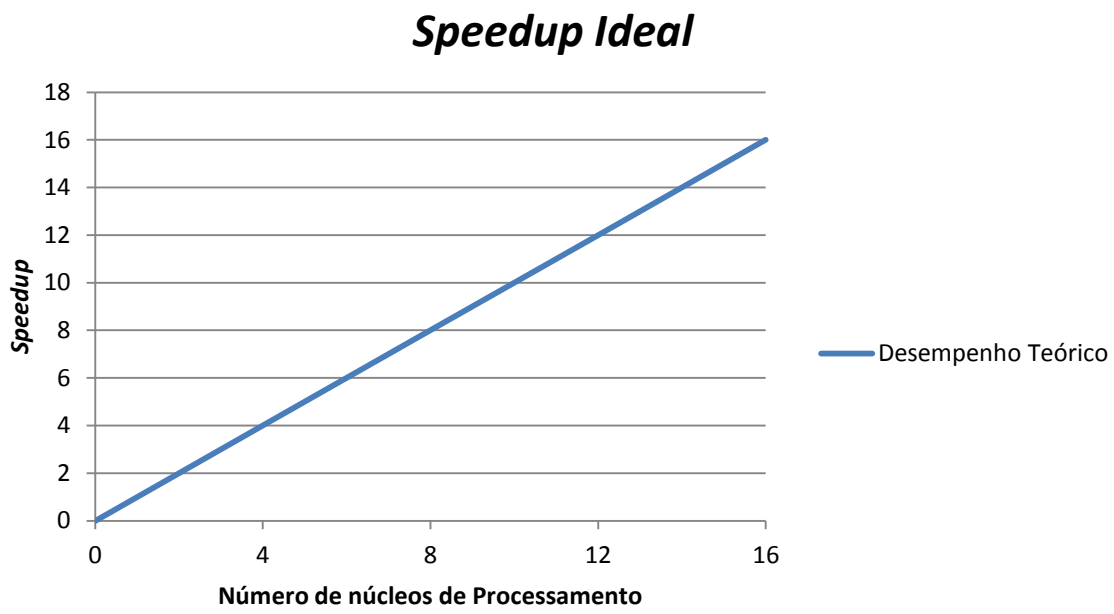


Figura 31 - Gráfico de *Speedup* Ideal

5.4.2 Eficiência

Eficiência é a medida que indica se um elemento de processamento foi utilizado de forma proveitosa. A eficiência é dada pela razão entre *Speedup* e o número de núcleos de processamento *n* (JUNIOR, 2006):

$$eficiência_{(n)} = \frac{speedup_{(n)}}{n} \quad (5.2)$$

Quando a eficiência é igual a 1, significa que o elemento de processamento foi usado 100% do tempo de execução do algoritmo, atingindo o desempenho máximo.

5.4.3 Coeficiente de Correlação de Pearson

Neste trabalho, o Coeficiente de Pearson será utilizado para medir a diferença entre as imagens iniciais e finais, e também entre as imagens intermediárias geradas pelo algoritmo, buscando assim, provar que o morfismo é realmente suave, ou seja, sem alterações bruscas entre uma imagem e outra.

O Coeficiente de Correlação de Pearson é uma medida capaz de indicar o grau de relacionamento linear entre duas variáveis quaisquer (BENESTY *et al*, 2008). Esta medida é especialmente útil em processamento de imagens, onde é possível realizar comparações entre imagens e o reconhecimento de padrões (NETO *et al*, 2007).

O coeficiente de Pearson é definido por *r* pela equação 5.3:

$$r = \frac{\sum_i (x_i - \bar{x})\,(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2}\,\sqrt{\sum_i (y_i - \bar{y})^2}} \quad (5.3)$$

Onde, x_i é a intensidade do *i-ésimo pixel* de uma imagem qualquer e $\dot{x}$ é a média de seus pixels. Já y_i é a intensidade do *i-ésimo pixel* de uma outra imagem qualquer e $\bar{y}$ também é a média de seus *pixels*.

O coeficiente pode variar entre -1 e 1 onde:

- *r* = -1 significa que uma imagem é exatamente o oposto da outra. Por exemplo, se a primeira imagem é formada apenas por *pixels* brancos, a segunda imagem será formada apenas por *pixels* pretos;
- *r = 0* significa que as imagens são independentes, ou seja, não possuem nenhum tipo de correlação linear.

- $r = 1$ significa que as imagens são exatamente iguais;

Ou seja, quando mais próximo de 1 forem os resultados, melhores eles serão.

5.5 APLICAÇÃO DAS MÉTRICAS DE AVALIAÇÃO

Em relação ao desempenho do algoritmo, foram definidos como métricas os resultados do *Speedup*, que é a razão entre o tempo de execução sequencial e o tempo de execução paralelo, e a eficiência, que é a razão entre o *Speedup* e o número de núcleos de processamento utilizados.

Segundo Wolberg (1998), o objetivo principal do morfismo é gerar transições suaves entre imagens. Então, a métrica de qualidade escolhida para a geração de imagens será o Coeficiente de Correlação de Pearson.

Para realizar essa medição, foi gerada uma sequência de 10 morfismos. Então, os coeficientes de Pearson Absoluto e Relativo foram calculados.

- Absoluto: Analisa a evolução da correlação das imagens em relação a imagem final, ou seja, o coeficiente será calculado entre todas as imagens e a imagem final.
- Relativo: Verifica o grau de correlação entre uma imagem qualquer e sua sucessora, buscando assim mostrar o quão semelhante elas são.

5.5 RESULTADOS ESPERADOS

Espera-se que ao final deste trabalho o algoritmo de morfismo esteja produzindo uma sequência de imagens aceitável, vistas pelo quesito do Coeficiente de Correlação de Pearson, ou seja, valores muito próximos a 1. Em relação ao desempenho, espera-se que, com o aumento de núcleos de processamento, o desempenho também aumente, chegando próximo do *Speedup* ideal. Este alto desempenho é esperado porque os processos são basicamente independentes, livres de sincronização.

6 RESULTADOS

Os testes foram realizados em um computador equipado com processador AMD Phenom II 2.8 GHz de quatro núcleos, com 4 GB de memória RAM e 500 GB de HD. O sistema operacional usado foi o *Linux Mint 8*. O grupo de imagens usadas nos testes está representado na Figura 32. O primeiro par de imagens foi escolhido para representar duas pessoas com características distintas como o comprimento do cabelo e o tom da pele. O segundo par de imagens foi retirado do trabalho de Wolberg (1998), que faz um apanhado sobre técnicas de morfismo e por fim, a imagem do gato foi escolhida para representar a utilização da técnica no cinema, que utiliza constantemente efeitos de transformação entre seres fisicamente diferentes, como a de um humano em um gato.

Figura 32 - Imagens usadas nos testes

Para garantir que os resultados não fossem afetados por oscilações do sistema, cada teste foi executado 5 vezes, e o resultado final é representado pela média simples de todas as execuções.

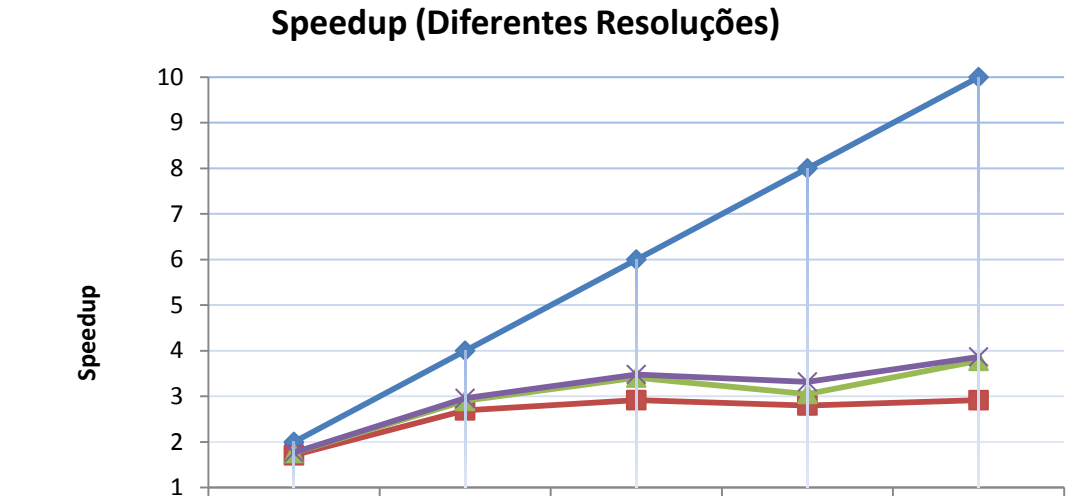
6.1 RELAÇÃO ENTRE A RESOLUÇÃO DAS IMAGENS E O DESEMPENHO DO ALGORITMO

No teste de *Speedup* foram selecionados três grupos de imagens de diferentes resoluções, para poder analisar como o algoritmo se comporta processamento imagens de diferentes resoluções:

- Resolução A: 135x100 *pixels;*
- Resolução B: 400x296 *pixels;*

- Resolução C: 480x640 *pixels.*

Para cada resolução, o algoritmo foi executado com o seguinte número de processos: [1, 2, 4, 6, 8, 10], gerando 10 imagens no morfismo. Os resultados podem ser analisados no Gráfico 1. O resultado do teste com 1 processo foi omitido, visto que ele só foi usado para calcular os *Speedups* dos processos subsequentes.



	2 Processos	4 Processos	6 Processos	8 Processos	10 Processos
Ideal	2	4	6	8	10
Resolução A	1,7073	2,6923	2,9167	2,8	2,9157
Resolução B	1,7576	2,9	3,4118	3,0526	3,7778
Resolução C	1,7662	2,9565	3,48	3,3171	3,8667

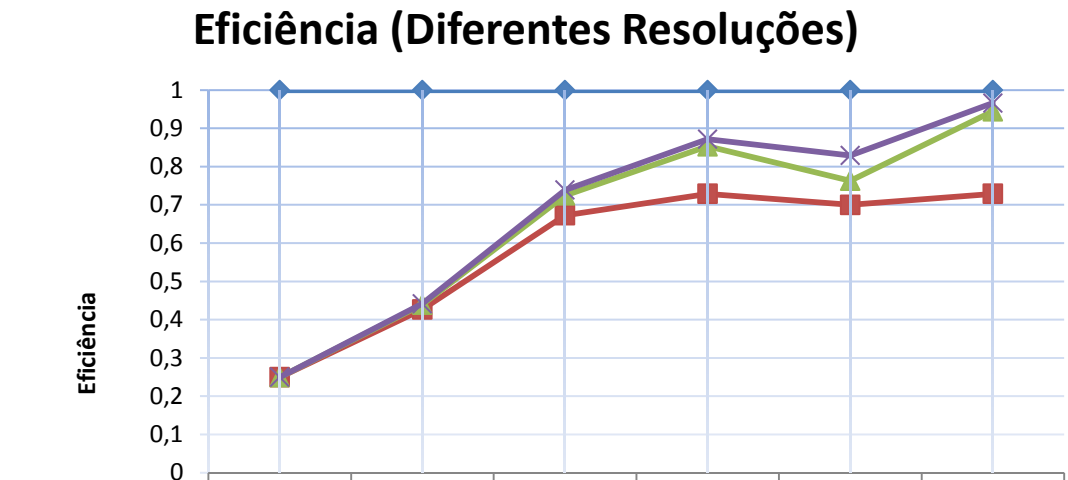
Gráfico 1 - *Speedup* (Diferentes Resoluções)

É possível notar que, até seis processos, o algoritmo obteve um Speedup crescente. A partir de seis processos praticamente não há aceleração da velocidade de processamento (s*peed up*), devido a distribuição de imagens entre os processos. É o que pode ser visualizado no Gráfico 1 onde se observa dez imagens a serem geradas e nem sempre elas serão divididas igualmente entre todos os processos. Por exemplo, quando o algoritmo é executado com oito processos, seis deles receberão uma imagem para tratar, enquanto os outros dois receberão duas imagens. Isso faz com que a CPU fique ociosa em um determinado ponto da execução, pois seis processos já estarão finalizados, enquanto os outros dois ainda estarão executando, pois possuem uma imagem a mais. Isso pode ser contornado utilizando-se o mesmo número de Imagens e de processos, visto que, os testes com dez imagens e dez processos apresentaram os melhores resultados.

É importante resaltar que, como o processador utilizado possui apenas 4 núcleos físicos, mesmo com o aumento de processos, o *Speedup* se manteve abaixo de 4. Isso é causado por que, quando o número de processos ultrapassa o número de núcleos físicos da máquina, inicia-se uma concorrência pela utilização da CPU, fazendo com que apenas 4 processos executem no mesmo instante de tempo.

Os *Speedups* para Resolução A foram os que apresentaram os piores resultados. Isso se deve ao fato de que o tempo de processamento de cada imagem é muito baixo em relação ao tempo de criação e sincronização dos processos, mostrando que a paralelização é mais vantajosa quando se necessita executar uma maior quantidade de cálculos. Outra informação que é possível extrair do Gráfico 1 é que o aumento de processos, em relação aos quatro núcleos físicos, não piora o desempenho do algoritmo mesmo quando esse número foi igual a 2,5 vezes o número de núcleos físicos do processador. Isso mostra que o tempo que o sistema operacional demora em criar e sincronizar novos processos é desprezível quando há uma elevada quantidade de cálculos para realizar.

O Gráfico 2 apresenta os testes sobre a eficiência do algoritmo. Eles seguem o mesmo roteiro utilizado nos testes de *Speedup*. Essa métrica mostra se a unidade de processamento foi utilizada de forma proveitosa ou não.



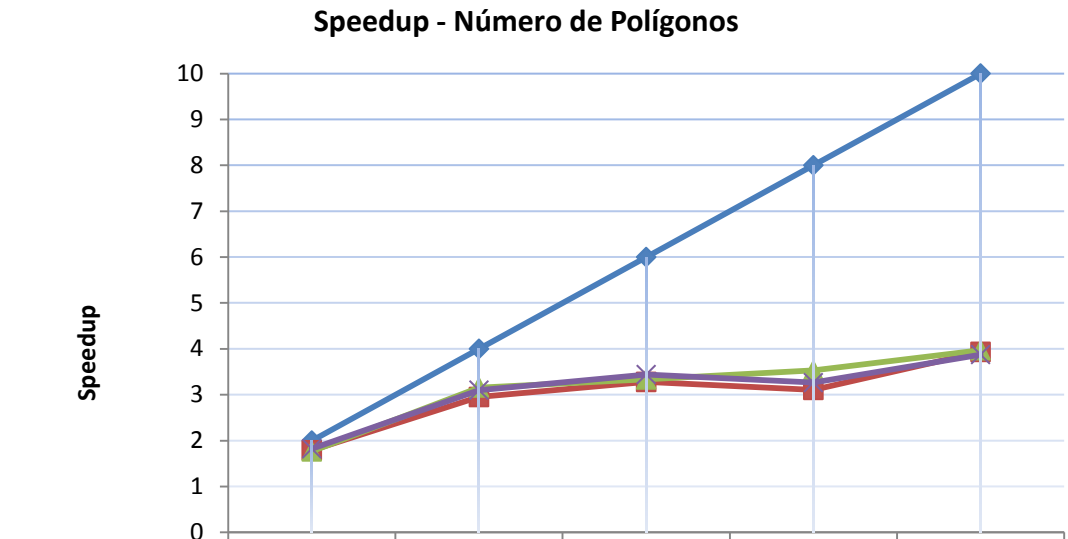
	1 Processo	2 Processos	4 Processos	6 Processos	8 Processos	10 Processos
Ideal	1	1	1	1	1	1
Resolução A	0,25	0,4268	0,6731	0,7292	0,7	0,7292
Resolução B	0,25	0,4394	0,725	0,853	0,7632	0,9445
Resolução C	0,25	0,4416	0,7391	0,8718	0,8293	0,966

Gráfico 2 - Eficiência (Resolução Diferentes Resoluções)

Quando o algoritmo é executado apenas com um processo, a eficiência fica em apenas 25%, mostrando que apenas um núcleo de processamento foi utilizado, e os outros ficaram ociosos. A eficiência melhorou conforme o aumento de processos, sendo que com dez processos, ela ficou muito próxima do ideal para as Resoluções B e C. Já no caso da Resolução A, o ganho não foi tão significativo devido ao tamanho reduzido da imagem.

6.2 RELAÇÃO ENTRE O NÚMERO DE POLÍGONOS E O DESEMPENHO DO ALGORITMO

Este teste foi realizado para avaliar a forma com que a quantidade de polígonos afeta o desempenho do algoritmo. O Gráficos 3 mostra o *Speedup* de execuções no mesmo formado dos gráficos anteriores e em imagens com a mesma resolução, mas com quantidades distintas de polígonos: 24, 94 e 174.

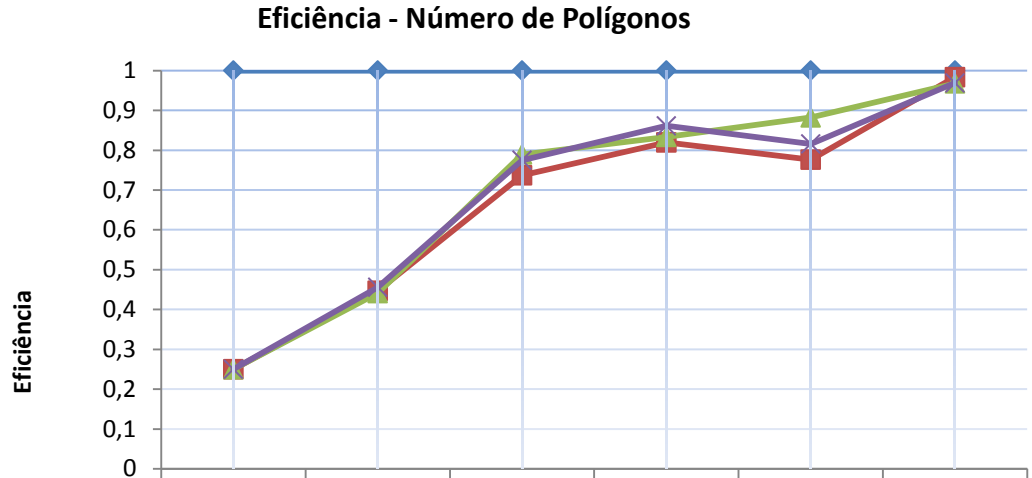


	2 Processos	4 Processos	6 Processos	8 Processos	10 Processos
Ideal	2	4	6	8	10
24 Polígonos	1,7879	2,95	3,278	3,1053	3,933
94 Polígonos	1,7647	3,1579	3,333	3,529	3,97
174 Polígonos	1,8235	3,1	3,44	3,2632	3,875

Gráfico 3 - *Speedup* (Número de Polígonos)

É possível concluir que o algoritmo teve pouca variação de desempenho entre as diferentes quantidades de polígonos. Assim como nos resultados anteriores, o teste com 10 processos obteve as melhores médias, mostrando que, para a arquitetura aqui utilizada, o número elevado de processos não sobrecarrega a CPU.

Em relação a eficiência do algoritmo e o número de polígonos, os resultados, que podem ser analisados no Gráfico 4, possuem o mesmo padrão dos testes com diferentes resoluções, chegando próximo da eficiência máxima quando executado com 10 processos.



	1 Processo	2 Processos	4 Processos	6 Processos	8 Processos	10 Processos
Ideal	1	1	1	1	1	1
24 Polígonos	0,25	0,447	0,737	0,8195	0,7763	0,9833
94 Polígonos	0,25	0,4412	0,7895	0,8333	0,8824	0,9678
174 Polígonos	0,25	0,4559	0,775	0,8611	0,8158	0,9688

Gráfico 4 - Eficiência (Número de Polígonos)

6.3 COEFICIENTE DE CORRELAÇÃO DE PEARSON

O Coeficiente de Correlação de Pearson foi usado com o intuito de analisar o grau de correlação entre as imagens e a evolução do morfismo.

A marcação de pontos nos dois primeiros experimentos busca relacionar as características semelhantes das imagens inicial e final, como por exemplo, olhos com olhos, nariz com nariz, etc. Porém, no experimento 3, por tratar de imagens de raças diferentes, segue outra linha de marcação, que não se baseia nas características principais das imagens, mas sim no efeito que se quer produzir. Isso significa que a marcação depende unicamente do julgamento do marcador. Entretanto, isso não significa que a criação da malha nos experimentos 1 e 2 não possa ser gerada buscando algum efeito específico, como por exemplo, a transformação da boca na imagem inicial em um nariz na imagem final.

A Figura 33 mostra a triangulação do primeiro experimento. O resultado está representado na Figura 34 e os Gráficos 5 e 6 representam seus Coeficientes Absoluto e Relativo, respectivamente.

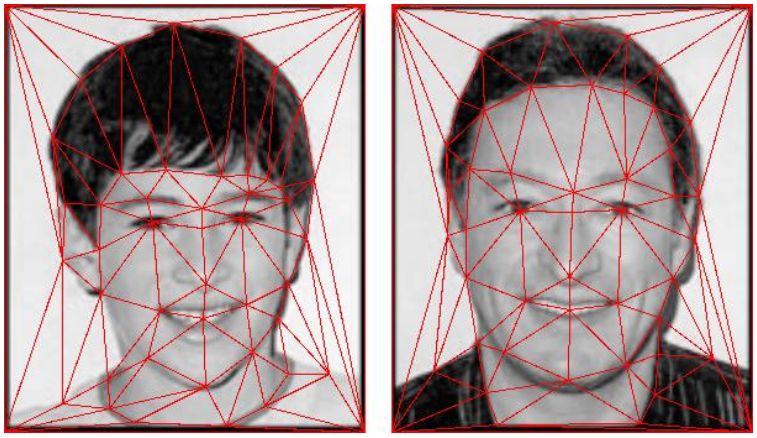


Figura 33 - Triangulação do Experimento 1

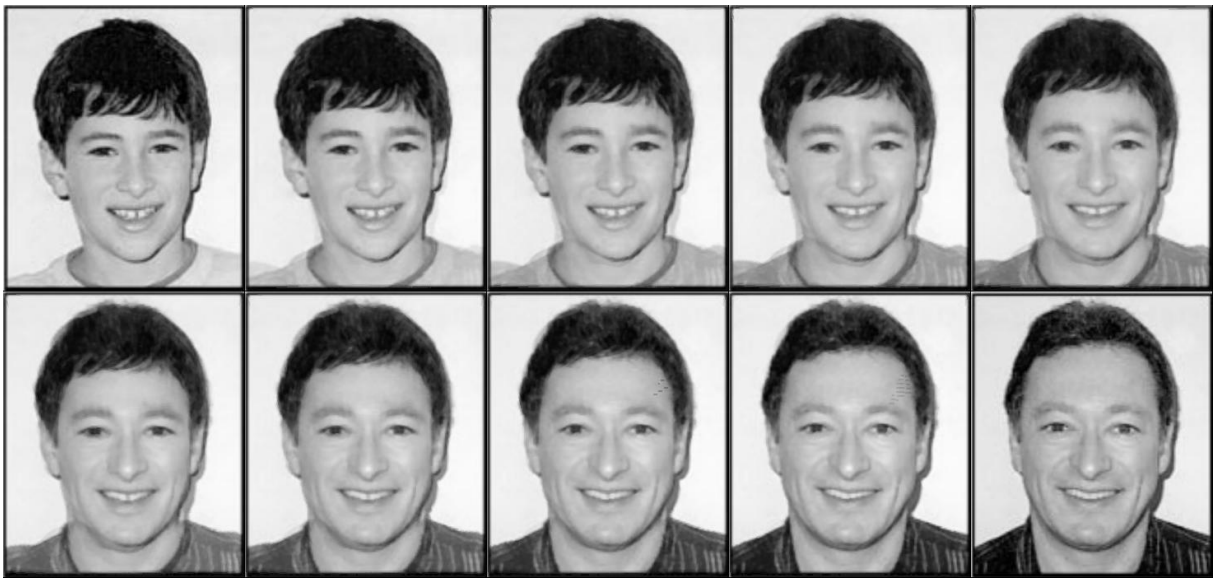


Figura 34 - Resultado do Experimento 1

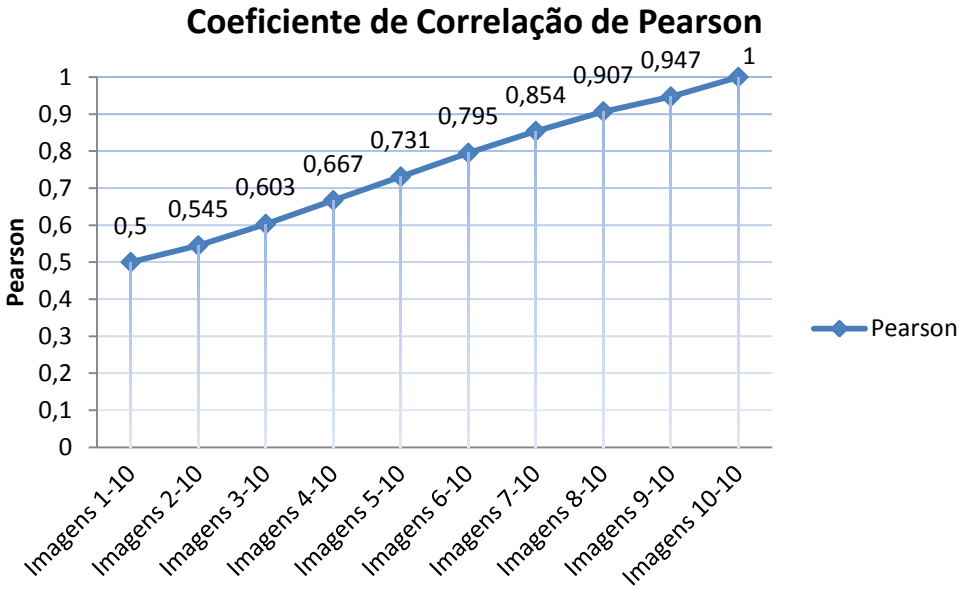


Gráfico 5 - Coeficiente de Correlação de Pearson Absoluto - Experimento 1

Partindo de uma imagem inicial muito diferente da final, as imagens obtidas vão gradualmente se aproximando da imagem final conforme o Gráfico 5. É também possível observar uma variação linear e crescente dos valores de correlação demonstrando que entre as duas imagens bastante diferentes foi obtida uma sequência de morfismos que gradualmente se aproximam da imagem final. Resta saber se a sequência gradual de morfismos apresenta uma evolução a passos suaves. É o que se pretende demonstrar com o Gráfico 6 no qual se avalia o quanto cada Imagem é semelhante à sua subsequente. Nota-se por esse gráfico que os pares de imagens subsequentes são fortemente correlacionadas (similares), isso é percebido na pequena variação nos baixos valores dos índices de correlação que vão de 0,001 a 0,034. O ideal seria uma variação constante, porém, durante o morfismo alguns *pixels* não são coloridos devido aos números fracionados gerados pelo cálculo do sistema linear. Para preenchê-los, é preciso utilizar uma técnica de interpolação de cores, e essa técnica pode causar uma pequena mudança na tonalidade de alguns *pixels*, e que consequentemente afetam o resultado do Coeficiente de Pearson.

Se os Gráficos 1 a 4 demonstram o bom desempenho computacional do método implementado, os Gráficos 5 e 6 demonstram o grau de eficiência estética do método, ou seja, a suavidade do morfismo entre os pares de imagens.

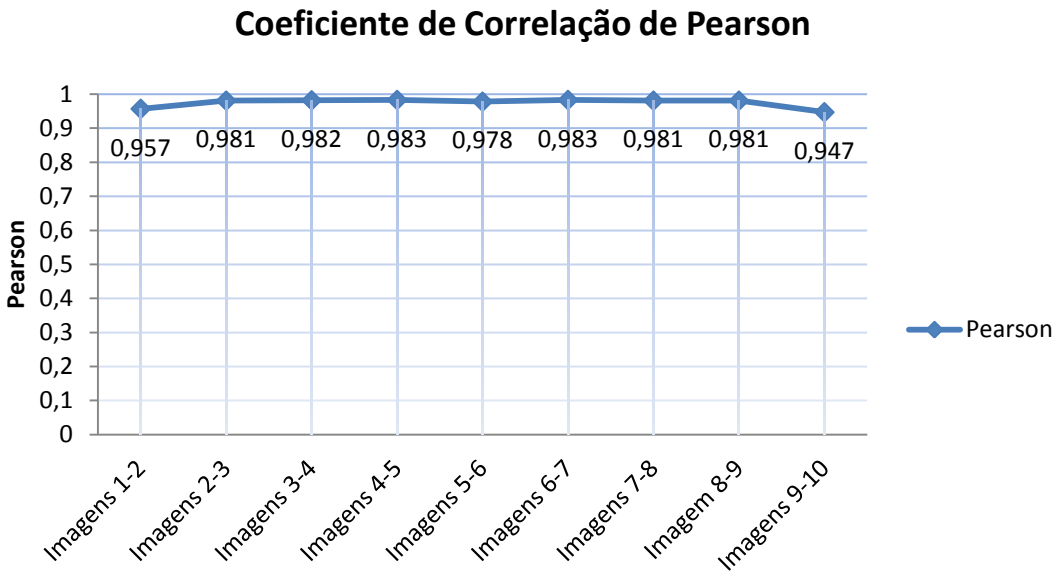


Gráfico 6 - Coeficiente de Correlação de Pearson Relativo - Experimento 1

A Figura 35 representa a triangulação do segundo experimento realizado e a Figura 36 representa o resultado. Os gráficos 7 e 8 mostram o Coeficiente de Pearson Absoluto e Relativo, respectivamente.

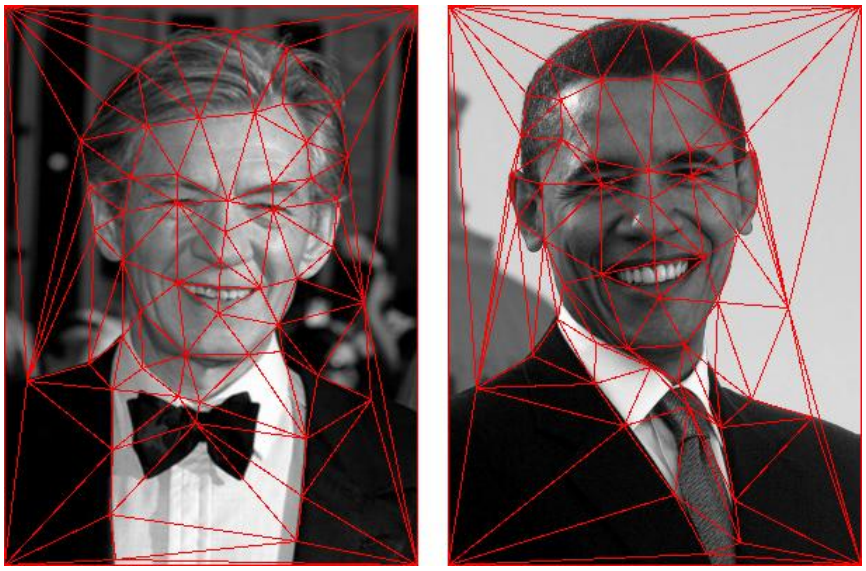


Figura 35 - Triangulação do Experimento 2

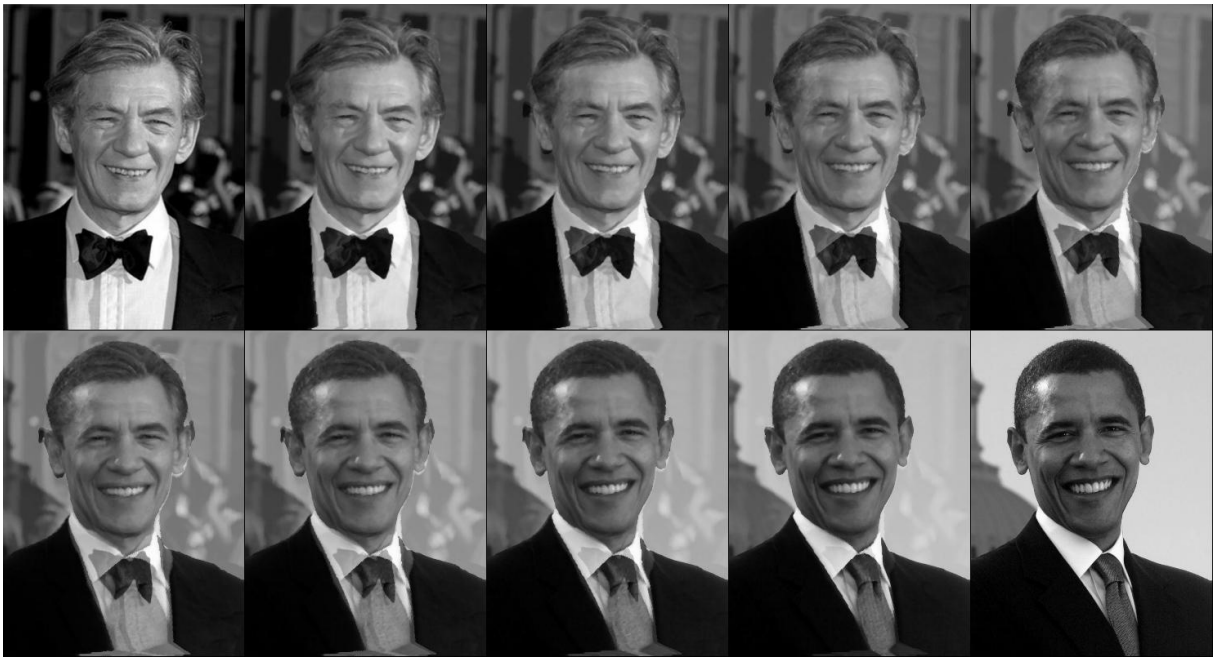


Figura 36 - Resultado do Experimento 2

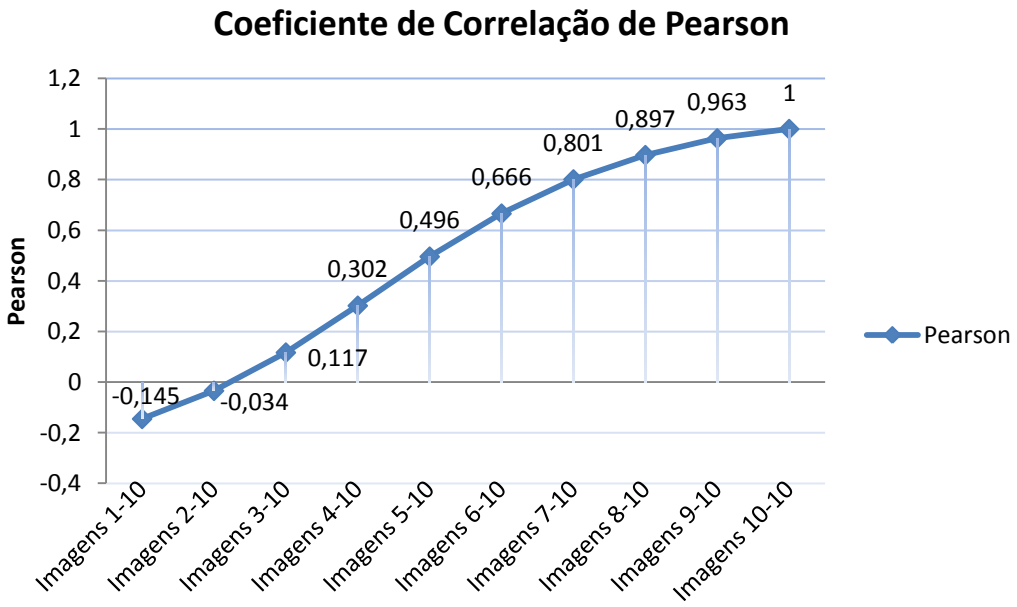


Gráfico 7 - Coeficiente de Correlação de Pearson Absoluto - Experimento 2

Por meio do gráfico acima, é possível notar que existia uma grande diferença entre as imagens segundo o Coeficiente de Pearson, tanto que esse valor é negativo entre as Imagens 1-10 e 2-10. Mas quanto mais próximo da Imagem final, maior é o Coeficiente Absoluto. Além do mais, os valores estão variando de forma uniforme, ou seja, sem alterações bruscas de valores, o que mostra que o morfismo também foi executado de forma uniforme e gradual.

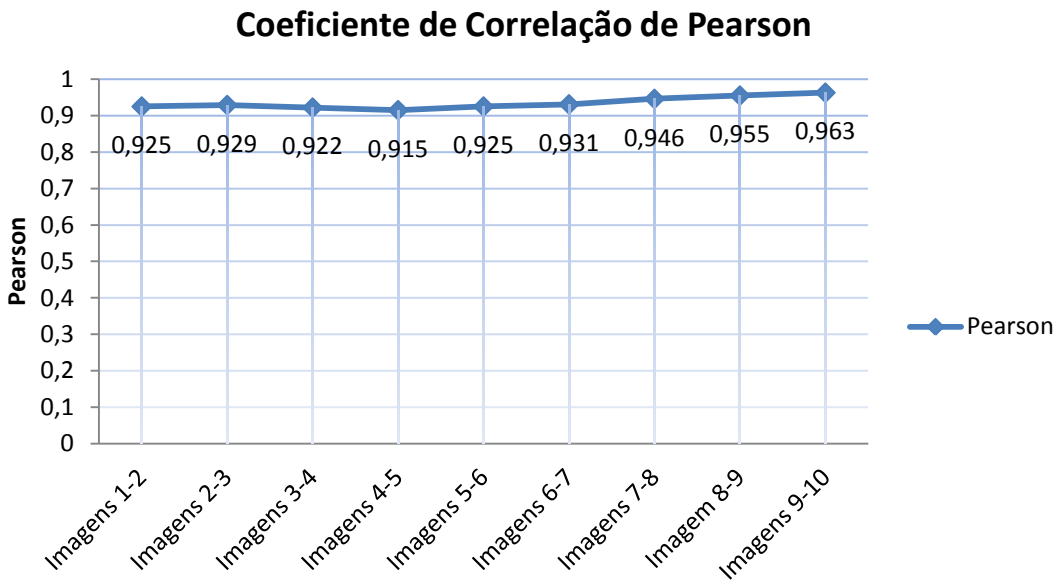


Gráfico 8 – Coeficiente de Correlação de Pearson Relativo - Experimento 2

O gráfico 8 mostra o quão cada Imagem é semelhante a sua subsequente. É possível observar uma pequena variação nas semelhanças que vão de 0,004 à 0,010, devido a questão de interpolação de cores já explicada.

O terceiro experimento busca analisar o morfismo de um rosto humano para uma forma diferente, que neste caso é o rosto de um gato, visto que existem diferenças interessantes como a localização das orelhas e o tamanho dos olhos. Neste caso, a marcação de pontos não precisa seguir uma ligação lógica entre as imagens, como por exemplo, um ponto na orelha na imagem inicial representar um ponto na orelha na imagem final. A posição dos pontos depende unicamente do julgamento da pessoa que está gerando os mesmos, de acordo com os efeitos artísticos desejados, que neste caso é transformar a cabeça do ser humano na cabeça do gato. A Figura 37 representa a triangulação das imagens e a Figura 38 o resultado. Os Gráficos 9 e 10 representam os Coeficientes de Pearson deste experimento.

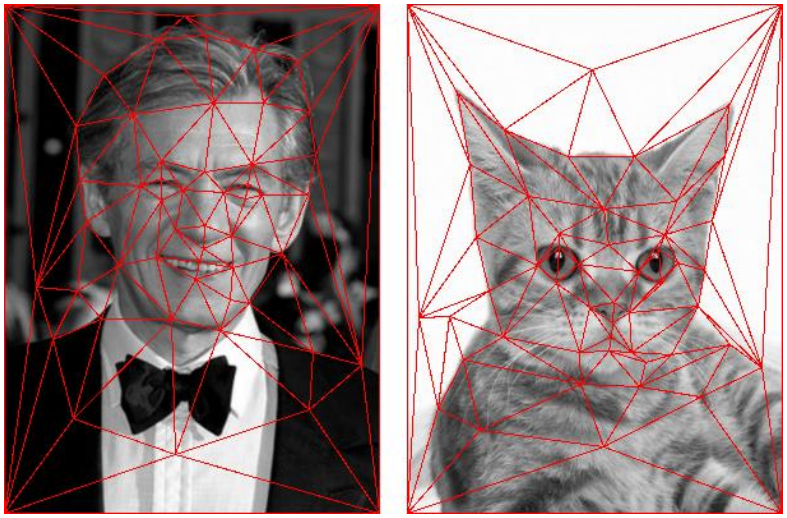


Figura 37 - Triangulação do Experimento 3

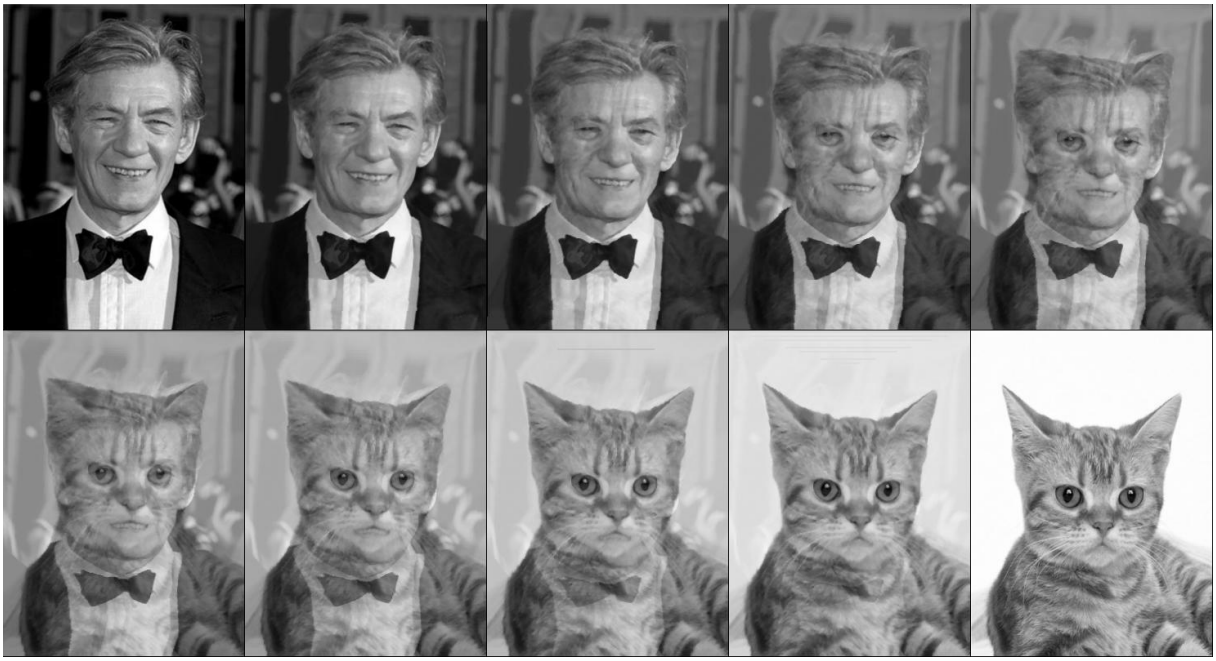


Figura 38 - Resultado do Experimento 3

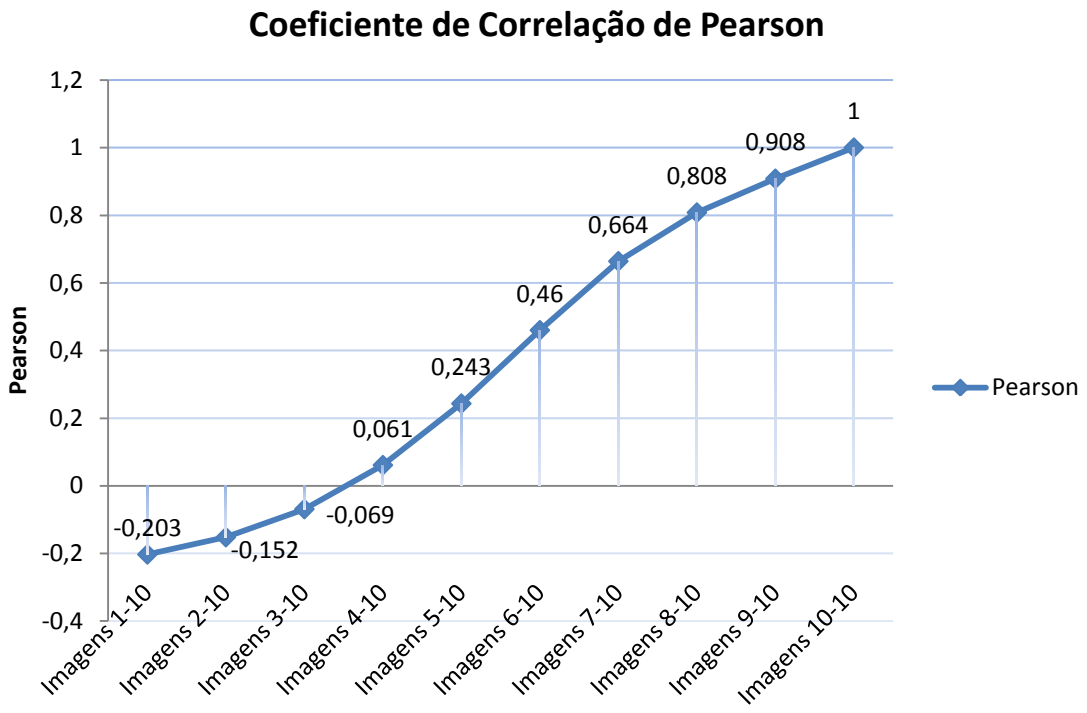


Gráfico 9 - Coeficiente de Correlação de Pearson Absoluto - Experimento 3

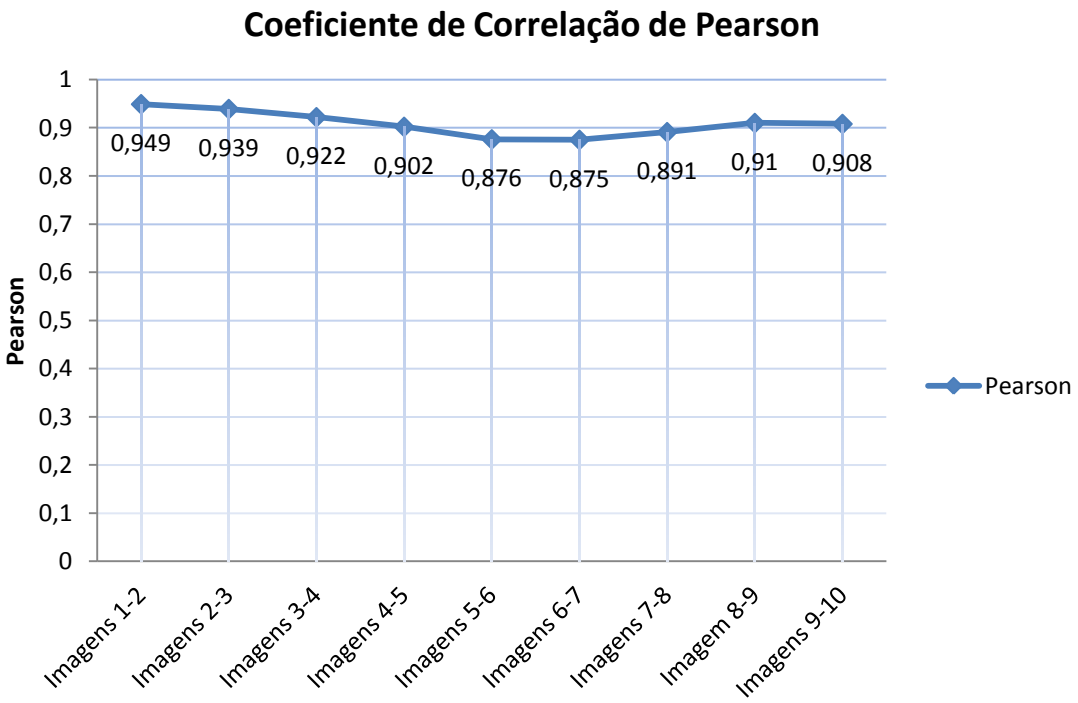


Gráfico 10 - Coeficiente de Correlação de Pearson Relativo - Experimento 3

De acordo com o Gráfico 9, o morfismo continuou suave, mesmo utilizando Imagens de raças diferentes. Já o Gráfico 10 apresentou uma variação elevada entre algumas imagens, comparada aos outros experimentos. Isso pode ter sido causado devido à textura da imagem, visto que o gato é coberto por pelos onde o ser humano possui apenas pele.

CONSIDERAÇÕES FINAIS

O morfismo de imagens é uma ferramenta para geração de efeitos especiais na televisão e no cinema, além de possuir aplicações no ramo de visão computacional e imagens médicas. Também é amplamente usado em aplicações científicas, como por exemplo, no estudo da evolução dos seres vivos e na análise do seu crescimento e desenvolvimento, assim como na assistência a cirurgias plásticas e de reconstrução facial.

Devido a necessidade de resultados cada vez mais exigentes em termos de qualidade (resolução, profundidade de cores), o tempo de processamento vem aumentando consideravelmente, podendo levar até mesmo várias horas. Por outro lado, os computadores multiprocessados têm oferecido um maior poder computacional a um custo bastante acessível. A motivação deste trabalho de paralelizar o algoritmo de morfismo decorre da popularização desta arquitetura frente à demanda de processamento do mesmo.

Basicamente, existem duas técnicas principais de morfismo: *crossfading* e deformação. A técnica de morfismo baseada em deformações ainda pode ser subdividida em dois ramos: a *Multilevel free-form deformation* (MFFD) e a malha de pontos. A técnica de geração de imagens abordada neste trabalho baseia-se na Malha Deformável e foi escolhida porque divide a imagem em vários pedaços isolados, característica imprescindível em algoritmos paralelos.

Para avaliar o algoritmo foram utilizadas as métricas de *Speedup*, eficiência e o Coeficiente de Correlação de Pearson, como forma de validação do trabalho.

O algoritmo foi capaz de atingir índices altos nos testes de *Speedup* e eficiência, devido a sua característica assíncrona. Mas visto que o processador utilizado possui apenas quatro núcleos físicos, mesmo com o aumento de processos, o *Speedup* se manteve abaixo de 4. Isso é causado por que, quando o número de processos ultrapassa o número de núcleos físicos da máquina, inicia-se uma concorrência pela utilização da CPU, fazendo com que apenas quatro processos executem no mesmo instante de tempo.

Os testes apontaram que a utilização de imagens em baixa resolução pode tornar o uso do algoritmo paralelo pouco vantajoso, visto que o tempo de

processamento de cada imagem é muito baixo. Também mostraram que o aumento de processos não afeta, e inclusive melhora o desempenho do algoritmo, mesmo que esse número seja 2,5 vezes maior ao número de núcleos físicos.

Em relação ao Coeficiente de Pearson, foi possível observar que o grau de Correlação variou de forma uniforme em todos os testes, mostrando que o morfismo também ocorreu de forma uniforme e bem distribuída entre toda a sequência de imagens. Também foi possível observar um alto grau de Correlação de cada imagem com sua sucessora, ou seja, o Coeficiente de Pearson sempre resultou em um valor próximo a um.

Por fim, é possível afirmar que algoritmo paralelo de morfismo e sua utilização em máquinas multiprocessadas é viável, visto que todo o poder de processamento estará sendo usado, fazendo com que o tempo de espera pelos resultados seja reduzido proporcionalmente ao número de núcleos físicos disponíveis.

Sugestão para trabalhos futuros é alterar o algoritmo para processar as imagens paralelamente em arquiteturas com núcleos físicos em quantidades maiores que quatro bem como em um cluster de computadores, permitindo assim, aumentar significativamente o número de processadores. Esse estudo entraria na questão de qual é a melhor forma de organizar o processamento, visto que um cluster geralmente é interligado por uma rede *Ethernet* local, e por isso, a banda de comunicação é muito menor do que a de um barramento entre a CPU e memória, por exemplo.

Outra abordagem que poderá ser explorada em trabalhos futuros refere-se à adaptação desse trabalho para o tratamento do envelhecimento de faces baseada em morfismo conforme descrito em Schroeder (2007). Isso é possível definindo-se características que determinam o envelhecimento e transferindo essas características a uma Imagem através de um formato de malha apropriado. Isto é útil no auxílio à busca de pessoas desaparecidas por muitos anos, atualização de fotografias tiradas há muito tempo por empresas e também na busca de fugitivos da polícia.

REFERÊNCIAS

ANTON, H.; RORRES, C. **Álgebra Linear com Aplicações**. 8. ed. Porto Alegre: Bookman. p. 495-499. 2005.

BARBOSA J. M. G. **Paralelismo em Processamento e Análise de Imagens Médicas**. Tese (Doutorado em Engenharia Electrotécnica e de Computadores) - Faculdade de Engenharia da Universidade do Porto, 2000.

BEIER, T.; NEELY, S. **Feature-Based Image Metamorphosis**. SIGGRAPH '92 Proceedings of the 19th Annual Conference On Computer Graphics And Interactive Techniques. New York, NY, USA. p. 35-42. 1992.

BENESTY, J.; CHEN, J. e HUANG, Y. **On the Importance of the Pearson Correlation Coefficient in Noise Reduction.** IEEE Transactions on audio, speech and language processing, v. 16, n. 4. p. 757-765. 2008.

BRESSON, X.; ESEDOGLU, S.; VANDERGHEYNST, P.; THIRAN, J.; OSHER S. **Fast Global Minimization of the Active Contour/Snake Model**. Journal of Mathematical Imaging and Vision. p. 151-167. 2005.

BRINKMANN, R. **The Art and Science of Digital Compositing**. San Francisco: Morgan Kaufmann. ISBN 978-0121339609. 1999.

CHIYOSHI, F.; GALVÃO, R. D.; MORABITO, R. **Modelo hipercubo**: análise e resultados para o caso de servidores não-homogêneos. Pesquisa Operacional, v. 21, n. 2, p. 199-218. 2001.

FLYNN, M. **Some Computer Organizations and Their Effectiveness**. Computers, IEEE Transactions on , Vol. C-21, p. 948-960, 1972.

GONZALEZ, Rafael C.; WOODS, Richard E. **Processamento de Imagens Digitais**. São Paulo: Edgard Blücher, ISBN 8521202644. 2000.

GRUNDLAND, M.; VOHRA, R.; WILLIAMS, G. P.; DODGSON, N. A. **Cross Dissolve Without Cross Fade**: Preserving Contrast, Color and Salience in Image Compositing. Eurographics 2006, Volume 25, Number 3. p. 577-586. 2006.

HODGES, W. L; GARLAND, T.; REYES, R.; ROWE, T. **Visualizing Horn Evolution by Morphing High-resolution X-ray CT Images**. International Conference on Computer Graphics and Interactive Techniques. ACM SIGGRAPH 2003 Sketches & Applications. New York, USA. 2003.

HUNTER, J.; DALE, D.; DROETTBOOM, M. **Intro: Matplotlib**. 2010. Disponível em: <http://matplotlib.sourceforge.net/>. Acessado: 22 Out. 2010.

JUNIOR, F. P. **Seleção de Variáveis e Características como Aplicação Paralela para Cluster MPI**. Dissertação (Mestre em Ciência da Computação) – Universidade Estadual de Maringá, Maringá, 2006.

KANG, S. B. **A Survey of Image-based Rendering Techniques**. Cambridge Research Laboratory. Technical Report Series. August 1997. Disponível em: < http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.4770&rep=rep1&type=pdf >. Acessado: Ago. 2010.

KASS, M.; WITKIN, A.; TERZOPOULOS D. **Snakes**: Active Contour Models. International Journal of Computer Vision. Springer Netherlands. p. 321-331. 1987.

LUTZ, M.; ASCHER D. **Aprendendo Python**. Porto Alegre: Bookman, 2007.

MORETTI C. O. **Análise de Estruturas Utilizando Técnicas de Processamento Paralelo Distribuído**. Dissertação (Mestre em Engenharia Civil) – Escola Politécnica da Universidade de São Paulo, 1997.

NETO, A. M.; RITTNER, L.; LEITE, N.; ZAMPIERI, D.E.; LOTUFO, R.; MENDELECK, A. **Pearson's Correlation Coefficient for Discarding Redundant Information in Real Time Autonomous Navigation System**. 16th IEEE

International Conference on Control Applications Part of IEEE Multi-conference on Systems and Control Singapore. p. 426-431. 2007.

NUMPY. **NumPy**. 2010. Disponível em: <http://numpy.scipy.org/>. Acessado: 22 Out. 2010.

PENHA, D. O.; CORRÊA, J. B. T.; MARTINS, C. A. P. S. **Análise Comparativa do Uso de Multi-Thread e OpenMP Aplicados a Operações de Convolução de Imagem**. In: Anais Do WSCAD 2002: III Workshop Em Sistemas Computacionais De Alto Desempenho (WSCAD). p. 118-125. 2002.

PYTHON, S. F. **About Python**. 2010. Disponível em: <http://python.org/about/>. Acessado: 20 Out. 2010.

PYTHON. **Initialization, Finalization, and Threads**. 2010. Disponível em: <http://docs.python.org/c-api/init.html>. Acessado: 20 Out. 2010.

RITZ-TIMME, S.; CATTANEO, C.; COLLINS, M. J.; WAITE, E. R.; SCHÜTZ, H. W.; KAATSCH, H.-J.; BORRMAN, H. I. M. **Age estimation: The state of the art in relation to the specific demands of forensic practise.** International Journal of Legal Medicine, Volume 113, Number 3. p. 129-136. 1999.

ROSE, C. D.; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Porto Alegre: Sagra-Luzzato, 2003.

SCHROEDER, G. N. **Morphing aplicado ao envelhecimento de imagens faciais**. Dissertação (Mestre em Engenharia Elétrica) – Universidade Estadual de Campinas, São Paulo, 2007.

SILVA, A. G.. **Ambiente de suporte ao ensino de processamento de imagens usando a linguagem Python**. 2003. 101 f. Dissertação (Mestrado de Engenharia da Elétrica) – Universidade Estatual de Campinas, São Paulo, 2003. Disponível em: <http://libdigi.unicamp.br/document/?code=vtls000290727>. Acessado: 08 out. 2010.

SOBCHACK, V. **Meta-Morphing:** Visual Transformation and the Culture of Quick-Change. Minneapolis: University of Minnesota Press, 1st Edition. 2000.

TANENBAUM, A. S.; WOODHULL, A. S. **Sistemas Operacionais**: Projeto e Implementação. Porto Alegre: Bookman, 2000.

XIAO, J.; SHAH, M. **From Images to Video:** View Morphing of Three Images. In: Vision, Modeling, and Visualization (VMV). p. 495-502. 2003.

WEISSTEIN, E. W. **Morphism From *MathWorld* - A Wolfram Web Resource.** 2011. Disponível em <http://mathworld.wolfram.com/Morphism.html>. Acessado: 29 out. 2011.

WOLBERG, G. **Image morphing:** a survey. Journal The Visual Computer. Heidelberg: Springer Berlin. p. 360-372. 1998.

ZAMITH, M.; MONTENEGRO, A.; PASSOS, E.; CLUA, E.; CONCI, A.; LEAL-TOLEDO, R.; MOURÃO P. T. **Real time feature-based parallel morphing in GPU applied to textured-based animation**. 16th International Conference on Systems, Signals and Image Processing, IWSSIP 2009. Chalkida, Greece. p. 1-4. 2009.

ANEXO A

CONCEITO DE IMAGEM

Uma imagem digital pode ser considerada como uma matriz cujos índices de linhas e colunas identificam um ponto na imagem, e o correspondente valor do elemento da matriz identifica o nível de cinza naquele ponto. Os elementos dessa matriz digital são chamados de *pixels* (*Picture Elements*) (ver Figura 39). A imagem digital pode ser expressa pela função *f(x, y)*, em que o valor ou amplitude de *f* nas coordenadas espaciais *(x, y)* dá a intensidade (brilho) da imagem naquele ponto. Como a luz é uma fonte de energia, *f(x, y)* deve ser maior que zero e finita (GONZALEZ e WOODS, 2000).

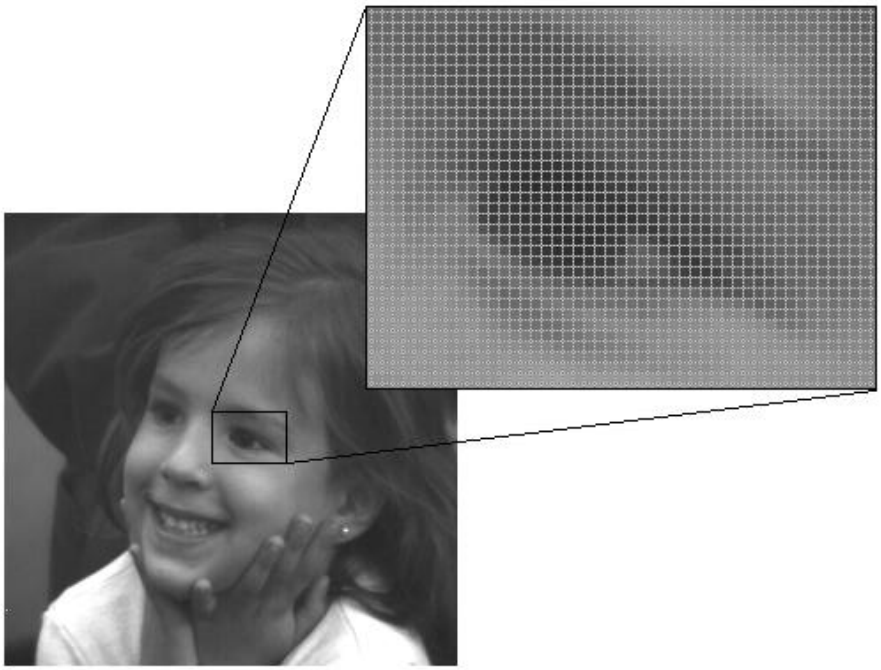


Figura 39 - Visualização dos *Pixels*

INTERPOLAÇÃO DE CORES

Os coeficientes c_i usados para criar a transformação espacial podem gerar valores não inteiros de *(x', y')*, e isso não é aceitável, pois os *pixels* são definidos através de coordenadas inteiras. Para contornar este problema, é preciso fazer uma

estimativa de cores para os *pixels* que não foram mapeados, baseando-se nos *pixels* vizinhos que foram mapeados.

A técnica mais simples de interpolação de cores é baseada na atribuição da cor do vizinho mais próximo ao *pixel* (ver Figura 40). Ela mostra um mapeamento de coordenadas inteiras *(x, y)* em coordenadas fracionadas *(x', y')*. Por esse motivo, o *pixel* recebeu a cor do seu vizinho mais próximo. Uma variação desta técnica é atribuir a cor média de todos os *pixels* vizinhos.

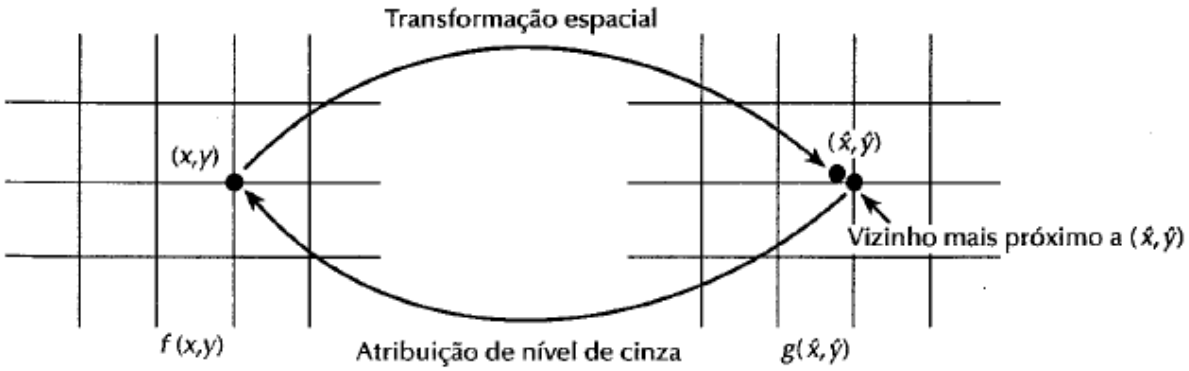


Figura 40 - Atribuição da cor do vizinho mais próximo

MÉTODO DE ELIMINAÇÃO DE GAUSS PARA RESOLUÇÃO DE SISTEMAS DE EQUAÇÕES LINEARES

O método de eliminação de Gauss, ou Eliminação Gaussiana, é um processo para resolver sistemas de equações lineares. A idéia base da eliminação de Gauss é reduzir uma matriz aumentada em uma matriz escalonada (ANTON e RORRES, 2005). Este método é o mais indicado para sistemas com poucas equações, que é o caso do algoritmo de morfismo abordado neste trabalho. Para resolver sistemas com, por exemplo, mais de 1000 equações, deve-se optar por métodos iterativos, como o método de Gauss-Siedel (CHIYOSHI; GALVÃO; MORABITO, 2001).

Entende-se como matriz escalonada uma matriz que satisfaz as seguintes condições:

1. Para um linha não nula, então o primeiro número não nulo da linha deve ser igual a 1. Este número é chamado de pivô;

2. Se existirem linhas nulas, elas devem ser agrupadas nas linhas inferiores da matriz;
3. Em quaisquer duas linhas sucessivas não nulas, o pivô da linha inferior deve estar mais a direita que o pivô da linha superior;
4. Cada coluna que possua um pivô, terá valor zero para todas as outras entradas.

Um exemplo de matriz escalonada pode ser visto na Figura 41.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \end{bmatrix}$$

Figura 41 - Matriz escalonada

Então, dada uma matriz $A = [a_{m,n}]$ de dimensões $m \times n$, temos três operações aplicáveis a ela, que são:

1. Multiplicar uma linha *m* por uma constante não nula;
2. Permutar duas linhas;
3. Somar o múltiplo de uma linha *m* a outra linha.

Estas operações são conhecidas como operações elementares sobre linhas (ANTON e RORRES, 2005).

O algoritmo de eliminação de Gauss pode ser escrito em 5 passos, descritos a seguir:

1. Identificar a coluna não nula mais a esquerda de A;
2. Permutar duas linhas se necessário, para deslocar para a primeira linha uma entrada não nula na primeira entrada da coluna;
3. Multiplicar a primeira linha pelo inverso da sua primeira entrada não nula, para obter 1;
4. Somar cada linha *m*, com *m* diferente de 1, a um múltiplo escalar apropriado da linha 1, buscando anular cada uma das entradas da coluna de *n*;

5. Repetir os passos 1 – 4, até chegar na última linha da matriz.

Após a aplicação da eliminação de Gauss, tem-se como saída uma matriz escalonada, onde será possível visualizar a solução do sistema linear.

ANEXO B - ALGORITMO

####### ARQUIVO mape.py #######

```
#----------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
# Parallel Image Morphing
#----------------------------

from numpy import *
from adpil import *
from EliminacaoGauss import *
from matplotlib.nxutils import *
from trata_imagem import *
import warnings

warnings.simplefilter('ignore', DeprecationWarning)

'''
Cria matriz para a funcao de Eliminacao de Gauss
'''
def cria_matriz(v1, v2):
        '''m1 = [[v1[0][0], v1[0][1], 1],
                    [v1[1][0], v1[1][1], 1],
                    [v1[2][0], v1[2][1], 1]]'''

        m1 =[[v1[0][0], v1[0][1], 1],
                 [v1[1][0], v1[1][1], 1],
                 [v1[2][0], v1[2][1], 1]]

        for i in range(3):
                for j in range(3):
                        m1[i][j] = str(m1[i][j])

        m2 = [[v2[0][0], v2[1][0], v2[2][0]]]#m2 possui as coordenadas das linhas de
cada vertice
        m3 = [[v2[0][1], v2[1][1], v2[2][1]]]#m3 possui as coordenadas das colunas de
cada vertice

        for i in range(3):
                m2[0][i] = str(m2[0][i])
                m3[0][i] = str(m3[0][i])

        ret = [m1,m2,m3]
        return ret

'''
```

```
Calcula o Metodo de Eliminacao de Gauss
'''
def ElimGauss(v1, v2):
	matriz = cria_matriz(v1, v2)
	incog = EliminacaoGauss(3, 1, matriz[0], matriz[1])
	incog = incog.getMatrizIncognitas()

	res = []
	res.append([float(incog[0][0]),float(incog[0][1]),float(incog[0][2])])

	incog = EliminacaoGauss(3, 1, matriz[0], matriz[2])
	incog = incog.getMatrizIncognitas()

	res.append([float(incog[0][0]),float(incog[0][1]),float(incog[0][2])])

	return res

'''
Funcao de mapeamento
'''
def mapeia(le, vet, vet2, liga):
	mat = zeros(le.shape) #imagem de retorno
	#varre todos os poligonos da imagem
	for l in liga:
		vert = [vet[l[0]], vet[l[1]] , vet[l[2]]]
		vert2 = [vet2[l[0]], vet2[l[1]] , vet2[l[2]]]

		for x in range(3): #Tratamento para nao ocorrer divisao por zero
			if vet[x][0] == 0: vet[x][0] = 0.001
			if vet[x][1] == 0: vet[x][1] = 0.001

		box = minRet(vert) #Cria um retangulo minimo para impedir que pontos
desnecessarios da imagem sejam visitados
		calc = ElimGauss(vert,vert2)

		for i in range(box[1][0], box[1][0] + box[0].shape[0]+1):
			for j in range(box[1][1], box[1][1] + box[0].shape[1]+1):
				#points = array([[i,j]])
				if pnpoly(i, j, vert):
					xi = calc[0][0]*(i) + calc[0][1]*(j) + calc[0][2]
					yi = calc[1][0]*(i) + calc[1][1]*(j) + calc[1][2]
					mat[xi,yi] = le[i,j]
	mat = tratarImagem(mat)
	return mat
```

####### ARQUIVO trata_imagem.py #######

```
#----------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
# Parallel Image Morphing
#----------------------------

from numpy import *
from ia636 import *
import Image
import ImageOps
import ImageFilter


def minRet(vet):
    #Movido de TCC2
    #Retorna valores para criar um retangulo minimo de um triangulo qualquer
    #e a posicao que ele deve iniciar o mapeamento
    #Cria um retangulo minimo para impedir que pontos desnecessarios da imagem
sejam visitados

    #vet = [[0,0],[10,15],[20,50]]
    aux1 = []
    aux2 = []
    a = []
    b = []
    for i in vet:
        aux1.append(i[0])
        aux2.append(i[1])
    a = [abs(aux1[0] - aux1[1]),abs(aux1[1] - aux1[2]),abs(aux1[2] - aux1[0])]
    b = [abs(aux2[0] - aux2[1]),abs(aux2[1] - aux2[2]),abs(aux2[2] - aux2[0])]
    return zeros([max(a),max(b)]), [min(aux1),min(aux2)]


def tratarImagem(img):
        img = naolinear(img, (3,3), 'maximo') #Ainda eh meio lento =/
        #img = naolinearNormal(img, (3,3), 'mediana')
        a = Image.fromarray(img)
        #a = a.filter(ImageFilter.MaxFilter)
        a = a.filter(ImageFilter.MedianFilter)
        a = ImageOps.autocontrast(a)
        img = array(a)
        return img


def naolinear(f, dim_janela, op='minimo'):
        A, B = dim_janela
```

```
        g = zeros(f.shape, 'uint8')
        for i in range(f.shape[0]):
                for j in range(f.shape[1]):
                        if f[i, j] == 0:
                                lin_ini = i - A/2
                                lin_fim = lin_ini + A
                                col_ini = j - B/2
                                col_fim = col_ini + B
                                if (lin_ini < 0): lin_ini = 0
                                if (lin_fim >= f.shape[0]): lin_fim = f.shape[0]
                                if (col_ini < 0): col_ini = 0
                                if (col_fim >= f.shape[1]): col_fim = f.shape[1]
                                m = f[lin_ini:lin_fim, col_ini:col_fim]
                                if op=='maximo':
                                        g[i,j] = max(ravel(m))
                                elif op=='minimo':
                                        g[i,j] = min(ravel(m))
                                elif op=='mediana':
                                        lista = sort(ravel(m))
                                        g[i,j] = lista[len(lista)/2]
                        else:
                                g[i,j] = f[i,j]
        return g

def naolinearNormal(f, dim_janela, op='minimo'):
        A, B = dim_janela
        g = zeros(f.shape, 'uint8')
        for i in range(f.shape[0]):
                for j in range(f.shape[1]):
                        lin_ini = i - A/2
                        lin_fim = lin_ini + A
                        col_ini = j - B/2
                        col_fim = col_ini + B
                        if (lin_ini < 0): lin_ini = 0
                        if (lin_fim >= f.shape[0]): lin_fim = f.shape[0]
                        if (col_ini < 0): col_ini = 0
                        if (col_fim >= f.shape[1]): col_fim = f.shape[1]
                        m = f[lin_ini:lin_fim, col_ini:col_fim]
                        if op=='maximo':
                                g[i,j] = max(ravel(m))
                        elif op=='minimo':
                                g[i,j] = min(ravel(m))
                        elif op=='mediana':
                                lista = sort(ravel(m))
                                g[i,j] = lista[len(lista)/2]
        return g
```

####### ARQUIVO parallel_morph.py #######

```
#----------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
# Parallel Image Morphing
#----------------------------
from adpil import *
from numpy import *
from morfismo import *
from mape import *
from trata_imagem import *
from multiprocessing import Process

'''
Funcao que retorna a posicao dos vertices em um dado instante do tempo
'''
def  morfa(vi, vf, t, periodo):
    ret = []
    for i in range(len(vi)):
        ret.append([inter_linear(vi[i][0],vf[i][0],t,periodo),
inter_linear(vi[i][1],vf[i][1],t,periodo)])
    return ret

'''
Funcao responsavel por gerar o mapeando de cada imagem individualmente no
tempo
'''
def morfa3(ini, vet, vet2, liga, q, periodo):
    ret = morfa(vet,vet2, q, periodo)
    mp = mapeia(ini, vet, ret, liga)
    return mp

'''
Funcao responsavel por salvar as imagens geradas
'''
def morfa4(ini, fin, vfin, vini, liga, qtd, li, lf): #qtd = quantidade de periodos / ll =
periodo que inicia o processo / lf = periodo que termina o processo
        for i in range(li,lf+1):
                if i == 0:
                        mp = tratarImagem(ini)
                        mf = morfa3(fin, vfin, vini, liga, (qtd-1)-i, qtd)
                elif i == (qtd-1):
                        mp = morfa3(ini, vini, vfin, liga, i, qtd)
                        mf = tratarImagem(fin)
                else:
                        mp = morfa3(ini, vini, vfin, liga, i, qtd)
                        mf = morfa3(fin, vfin, vini, liga, (qtd-1)-i, qtd)
                alpha = ((100 * i) / (qtd-1))/100.0
```

```
                out = (mp * (1.0 - alpha)) + (mf * alpha) #Pinta os pixels conforme
posicao da imagem no tempo
                print 'Imagem', i+1,'Processada!'
                adwrite('results/r'+str(i)+'.pgm', out) #Resultado final
                adwrite('results/rrr'+str(i)+'.pgm', mp) #Deformacao da imagem inicial
                adwrite('results/rr'+str(i)+'.pgm', mf) #Deformacao da imagem final


def morphing(ini, fin, vfin, vini, liga, qtdPer, qtdProc): #qtd de processos deve ser
maior que qtd de periodos
        print '***********************************************'
        print '* Quantidade de Processos em Paralelo:', qtdProc
        print '* Quantidade de Imagens que Serao Geradas:', qtdPer
        print '* Resolucao da Imagem:', str(ini.shape), '(Altura X Largura) -
',str(ini.shape[0]*ini.shape[1]),'pixels'
        print '* Numero de Pontos:', str(len(vini))
        print '* Numero de Poligonos:', str(len(liga))
        print '***********************************************'
        print 'Aguarde o termino do processamento...'
        print ''
        processos = []
        passo = 0
        esc = zeros(qtdProc, int)
        j = 0
        for i in range(qtdPer):
                esc[j] = esc[j] + 1
                j = j + 1
                if j >= qtdProc:
                        j = 0
        if qtdProc > qtdPer:
                print 'Quantidade de processos deve ser maior que quantidade de
periodos!'
                return

        for i in range(qtdProc):
                if i == qtdProc-1 and (passo + qtdPer/qtdProc)-1 < qtdPer-1:
                        processos.append(Process(target = morfa4, args = (ini, fin, vfin,
vini, liga, qtdPer, passo, passo + esc[i] -1,)))
                else:
                        processos.append(Process(target = morfa4, args = (ini, fin, vfin,
vini, liga, qtdPer, passo, passo + esc[i] - 1,)))
                passo = passo + esc[i]

        for i in processos: #Inicia todos os processos
                i.start()

        for i in processos: #Aguarda o termino de todos os processos
                i.join()

        print '\nAs imagens foram salvas na pasta "results".'
```

####### ARQUIVO morfismo.py #######

```
#---------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
# Parallel Image Morphing
#---------------------------

from numpy import *
from math import *

'''
Morfismo
'''
def inter_linear(a, b, t, periodo):  #t eh um tempo qlqr no periodo, a e b sao apenas numeros
    ################
    # Funcao usada inter-imagens
    ################
    if t == 0: return a
    if t == periodo-1: return b
    c = b - a
    c = float(c) / float((periodo-1))#Float para nao perder a precisao
    res = float(a)
    for i in range(0, t):
        res = res + c
    return int(res)

#for i in range(10):
#    print inter_linear(0,500, i, 10)

def inter_linear_new(a, t, periodo):  #t eh um tempo qlqr no periodo, a eh uma tupla
    ################
    # Funcao usada inter-imagens
    ################
    if t == 0: return a[0]
    if t == periodo-1: return a[1]
    c = a[1] - a[0]
    c = c / (periodo-1)
    res = a[0]
    for i in range(0, t):
        res = res + c
    return res

def cria_linha(pi,pf): #retorna um vetor de pontos que forma uma linha ligando os pontos --> cria_linha([0,0], [10,12])
    res = []
```

```
    distancia = ((pi[0]-pf[0])**2 + (pi[1]-pf[1])**2)**0.5 #pitagoras
    distancia = distancia +1
    for i in range(distancia):
        res.append ((ceil(inter_linear(pi[0],pf[0], i, distancia)),ceil(inter_linear(pi[1],pf[1],
i, distancia))))#ceil arredonda pra cima
    return res
```

####### ARQUIVO EliminacaoGauss.py #######

```
# -*- coding: latin1 -*-


# Autor: Jonnathan Weber
# Email: jonny172@Msn.com
# Data: 28/05/2010
# Objetivo: Calcular sistemas matriciais a partir do metodo de gauss-compacto

from fractions import Fraction as f

'''
Incognitas = 1
matrizInicial = [['2', '-4'], ['2', '3']]
matrizIndependentesInicial = [['2', '21']]
ordem = 2
resultado = EliminacaoGauss(ordem, Incognitas, matrizInicial,
matrizIndependentesInicial)
resultado.getMatrizFinal()
'''
class EliminacaoGauss():
	erro = 0
	def __init__(self, ordem, incognitas, matriz1, matriz2):
		self.ordem = ordem
		self.qIncognitas = incognitas
		self.matrizInicial = matriz1
		self.matrizIndependentesInicial = matriz2
		self.calculaMatrizMetodoEliminacao()

	def getErro(self):
		return self.erro

	def setErro(self,_nEr):
		self.erro = _nEr

	def defineMatrizAuxiliar(self):
		matriz = []
		for i in range(self.qIncognitas):
			linha = []
			for j in range(self.ordem):
				linha.append(self.matrizIndependentesInicial[i][j])
			matriz.append(linha)
		return matriz

	def defineMatrizIcognitas(self):
		matriz = []
		for i in range(self.qIncognitas):
			linha = []
			for j in range(self.ordem):
```

```
                        linha.append('0')
                matriz.append(linha)
        return matriz

    def defineMatrizFinal(self):
        matriz = []
        for i in range(self.ordem):
            linha = []
            for j in range(self.ordem):
                linha.append(self.matrizInicial[i][j])
            matriz.append(linha)
        return matriz

    def calculaMatrizMetodoEliminacao(self):
        self.matrizFinal = self.defineMatrizFinal()
        self.matrizIndependentesFinal = self.defineMatrizAuxiliar()
        self.matrizIndependentesFinal[0][0] =
self.matrizIndependentesFinal[0][0]
        k = self.ordem
        lista = []
        nI = 0
        while k > 0:
            i = self.ordem - k
            den = f(self.matrizFinal[self.ordem-k][self.ordem-k])
            #Verifica elemento e permuta a linha
            if (den == 0):
                if (i == 0):
                    self.setErro(1)
                    break
                else:
                    lista = self.matrizFinal[i+1]
                    self.matrizFinal[i+1] = self.matrizFinal[i]
                    self.matrizFinal[i] = lista
                    nI = self.matrizIndependentesFinal[0][i+1]
                    self.matrizIndependentesFinal[0][i+1] =
self.matrizIndependentesFinal[0][i]
                    self.matrizIndependentesFinal[0][i] = nI
            den = f(self.matrizFinal[self.ordem-k][self.ordem-k])
            if (den == 0):
                self.setErro(1)
                break
            while i+1 < self.ordem:
                contador = 0
                j = self.ordem - k + 1
                num = f(self.matrizFinal[i+1][self.ordem-k])
                divisor = num/den
                while contador < self.ordem:
                    if (j > self.ordem):
                        break
```

```
                                        self.matrizFinal[i+1][j-1] = f(self.matrizFinal[i+1][j-1])
- f(self.matrizFinal[self.ordem-k][j-1]) * divisor
                                        j += 1
                                        contador += 1
                                self.matrizIndependentesFinal[0][i+1] =
f(self.matrizIndependentesFinal[0][i+1]) -
f(self.matrizIndependentesFinal[0][self.ordem-k]) * divisor
                                i += 1
                        k -= 1
                else:
                        self.calculaXMetodoEliminacao()

        def calculaXMetodoEliminacao(self):
                self.matrizIncognitas = self.defineMatrizIcognitas()
                c = 0
                k = self.ordem-1
                while c < self.ordem:
                        termo = self.calculaX(self.matrizFinal,self.matrizIncognitas[0],k-
c,k)
                        if (f(self.matrizFinal[k-c][k-c]) == 0):
                                self.setErro(1)
                                break
                        else:
                                self.matrizIncognitas[0][k-c] =
(f(self.matrizIndependentesFinal[0][k-c]) - f(termo))/f(self.matrizFinal[k-c][k-c])
                                c += 1


        def calculaX(self, mA,x,j,k):
                if (j == k):
                        k -= 1
                if (k < 0):
                        return 0
                if (k < j):
                        return 0
                return f(mA[j][k])*f(x[k]) + self.calculaX(mA,x,j,k-1)

        def getMatrizFinal(self):
                return self.matrizFinal

        def getMatrizIndependentesFinal(self):
                return self.matrizIndependentesFinal

        def getMatrizIncognitas(self):
                return self.matrizIncognitas
```

####### ARQUIVO preparar_vetor_para_triangulacao.py #######

```
#----------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
#----------------------------

from numpy import *
from matplotlib.tri import *

def Triangulacao(a):
	x = []
	y = []

	for i in a:
		x.append(i[0])
		y.append(i[1])

	t = Triangulation(x, y)
	res = t.triangles

	res2 = []

	for i in range(len(res)):
		res2.append(append(res[i],res[i][0]))

	return res2
```

####### ARQUIVO ModeloExecucao.py #######

```
#----------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
#----------------------------

from adpil import *
from parallel_morph import *


ini = adread('Imagens/x.pgm')
fin = adread('Imagens/y.pgm')


periodo = 10

processos = 2

#Pontos da Imagem Inicial
vini = [[0, 0], [0, 295], [399, 295], [399, 0], [173, 54], [48, 147], [168, 223], [285, 84],
[281, 211]]

#Pontos da Imagem Final
vfin = [[0, 0], [0, 295], [399, 295], [399, 0], [154, 55], [57, 133], [161, 198], [277, 84],
[293, 240]]

#Ligacoes que representam os poligonos criados pelos pontos
liga = [array([0, 3, 4, 0]), array([3, 7, 4, 3]), array([0, 4, 5, 0]), array([5, 4, 6, 5]),
array([7, 6, 4, 7]), array([8, 6, 7, 8]), array([8, 7, 2, 8]), array([3, 2, 7, 3]), array([6, 1, 5,
6]), array([0, 5, 1, 0]), array([6, 8, 2, 6]), array([1, 6, 2, 1])]

#Executa a funcao com os parametros definidos.
morphing(ini, fin, vfin, vini, liga, periodo, processos)
```

####### ARQUIVO cria_pontos.py #######

```
#----------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
#----------------------------

from Tkinter import *
from preparar_vetor_para_triangulacao import *
from mostra_pol import *
import tkMessageBox

def center_window(w=300, h=200):
        ws = root.winfo_screenwidth()
        hs = root.winfo_screenheight()
        x = (ws/2) - (w/2)
        y = (hs/2) - (h/2)
        root.geometry('%dx%d+%d+%d' % (w, h, x, y))

class MyButton(Button):

    def __init__(self, parent, xx, yy, tipo): #tipo referente a imagem 1 ou 2
        self.xx = xx
        self.yy = yy
        if tipo == 1:
            self.b = Button(parent, background = 'red', command = self.pos )
        else:
            self.b = Button(parent, background = 'red')
        self.b.place_configure(x = xx, y = yy, width = 5, height = 5)

    def pos(self):
        global pol
        pol = pol + (self.xx+2, self.yy+2)
        if len(pol) >= 6:
            global p, p2, nmr_pol, vet_poligonos, vet
            vet_poligonos.append([vet.index([pol[1],pol[0]]) ,vet.index([pol[3],pol[2]]) ,
vet.index([pol[5],pol[4]]), vet.index([pol[1],pol[0]])])
            p.create_polygon(pol, fill='', outline='red')
            aux = ()
            for i in vet_poligonos[len(vet_poligonos)-1]:
                aux = aux + (vet2[i][1], vet2[i][0])
            p2.create_polygon(aux, fill='', outline='red')
            pol = ()
            nmr_pol = nmr_pol + 1
        print self.yy+2, self.xx+2
        return [self.yy+2, self.xx+2]
```

```
vet = []
vet2 = []
pol = ()
nmr_pol = 0
vet_poligonos = []
flag = 1
flag2 = 0 #flag que obriga a criacao de um ponto em cada imagem


def morfJanela():
    pass

def ImprimeIni():
    print vet

def ImprimeFin():
    print vet2

def ImprimeTudo():
    print ''
    print 'vini = ' + str(vet)
    print ''
    print 'vfin = ' + str(vet2)
    print ''
    print 'liga = ' + str(Triangulacao(vet))
    print ''

def Visu():
        if len(vet) == len(vet2):
                visualiza(vet, vet2, Triangulacao(vet), img, img2)
        else:
                tkMessageBox.showinfo("Atencao!", "Voce esqueceu de marcar \num
ponto na segunda imagem!")

def ImprimeLiga():
    print vet_poligonos

def mudaFlag():
    global flag, blq
    if flag == 1:
        flag = 0
        blq.configure(text = "Desbloquear Criacao de Pontos")
    else:
        flag = 1
        blq.configure(text = "Bloquear Criacao de Pontos")

def Posicao(event):
    global flag2
    if flag == 1 and flag2 == 0:
        posicao.set('('+str(event.y)+', '+str(event.x)+')')
```

```
        but = MyButton(p, event.x-2, event.y-2, 1)
        vet.append([event.y,event.x])
        flag2 = 1



def Posicao2(event):
    global flag2
    if flag == 1 and flag2 == 1:
        posicao2.set('('+str(event.y)+', '+str(event.x)+')')
        id = p2.create_rectangle((event.x-2, event.y-2,event.x+2, event.y+2), fill="red",
outline="#fb0", tags = 2)
        but2 = MyButton(p2, event.x-2, event.y-2, 2)
        vet2.append([event.y,event.x])
        flag2 = 0


root = Tk()

img = PhotoImage(file='Imagens/ian.pgm')
img2 = PhotoImage(file='Imagens/b.pgm')

posicao = StringVar()


p = Canvas(root, width = img.width()+15, height = img.height()+15)
p.place(x=20,y=20)
p.bind("<Button-1>", Posicao)
p.create_image(1, 1, anchor=NW, image=img)

posicao2 = StringVar()

p2 = Canvas(root, width = img.width()+15, height = img.height()+15)
p2.place(x=img.width()+20+20,y=20)
p2.bind("<Button-1>", Posicao2)
p2.create_image(1, 1, anchor=NW, image=img2)

center_window(img.width()*2+60, img.height()+200)

but = MyButton(p, 0-2, 0-2, 1)
vet.append([0,0])

but = MyButton(p, img.width()-1-2, 0-2, 1)
vet.append([0, img.width()-1])

but = MyButton(p, img.width()-1-2, img.height()-1-2, 1)
vet.append([img.height()-1, img.width()-1])

but = MyButton(p, 0-2, img.height()-1-2, 1)
vet.append([img.height()-1, 0])
```

```
but = MyButton(p2, 0-2, 0-2, 2)
vet2.append([0,0])

but = MyButton(p2, img.width()-1-2, 0-2, 2)
vet2.append([0, img.width()-1])

but = MyButton(p2, img.width()-1-2, img.height()-1-2, 2)
vet2.append([img.height()-1, img.width()-1])

but = MyButton(p2, 0-2, img.height()-1-2, 2)
vet2.append([img.height()-1, 0])


Button(root, text = 'Imprimir Tudo!!!', command = ImprimeTudo).place(x=20,
y=img.height()+30)
Button(root, text = 'Imprimir Pontos da Imagem Inicial', command =
ImprimeIni).place(x=20, y=img.height()+120)
Button(root, text = 'Imprimir Pontos da Imagem Final', command =
ImprimeFin).place(x=20, y=img.height()+60)
Button(root, text = 'Imprimir Poligonos', command = ImprimeLiga).place(x=20,
y=img.height()+90)
Button(root, text = 'Visualizar Triangulacao', command = Visu).place(x=350,
y=img.height()+90)
blq = Button(root, text = "Bloquear Criacao de Pontos", command = mudaFlag)
blq.place(x=350,y=img.height()+60)

root.mainloop()
```

####### ARQUIVO mostra_pol.py #######

```
#----------------------------
# Autor: Gustavo Frizzo
# Email: guto.smo@gmail.com
# 2011
#----------------------------

from Tkinter import *
from numpy import *

######################
#
# Funcao chamada por cria_pontos.py
#
######################

def visualiza(vini, vfin, liga, img, img2):

	top = Toplevel()

	imgg = img
	imgg2 = img2

	w = imgg.width()*2+60
	h = imgg.height()+40
	# get screen width and height
	ws = top.winfo_screenwidth()
	hs = top.winfo_screenheight()
	# calculate position x, y
	x = (ws/2) - (w/2)
	y = (hs/2) - (h/2)
	top.geometry('%dx%d+%d+%d' % (w, h, x, y))
	#top.geometry("1000x600")


	pp = Canvas(top, width = imgg.width()+15, height = imgg.height()+15)
	pp.place(x=20,y=20)
	pp.create_image(0, 0, anchor=NW, image=imgg)

	pp2 = Canvas(top, width = imgg.width()+15, height = imgg.height()+15)
	pp2.place(x=imgg.width()+20+20,y=20)
	pp2.create_image(0, 0, anchor=NW, image=imgg2)

	for i in range(len(vini)):
		vini[i] = [vini[i][1], vini[i][0]]
		vfin[i] = [vfin[i][1], vfin[i][0]]
		if vini[i][0] == 0:
			vini[i][0] = 1
```

```
                if vini[i][1] == 0:
                        vini[i][1] = 1
                if vfin[i][0] == 0:
                        vfin[i][0] = 1
                if vfin[i][1] == 0:
                        vfin[i][1] = 1


        for i in liga:
                id = pp.create_polygon([vini[i[0]], vini[i[1]], vini[i[2]]], fill='', outline='red')
                id = pp2.create_polygon([vfin[i[0]], vfin[i[1]], vfin[i[2]]], fill='', outline='red')

        #Por algum motivo desconhecido, eh necessario voltar do jeito que estava....
        for i in range(len(vini)):
                vini[i] = [vini[i][1], vini[i][0]]
                vfin[i] = [vfin[i][1], vfin[i][0]]
                if vini[i][0] == 1:
                        vini[i][0] = 0
                if vini[i][1] == 1:
                        vini[i][1] = 0
                if vfin[i][0] == 1:
                        vfin[i][0] = 0
                if vfin[i][1] == 1:
                        vfin[i][1] = 0

        top.mainloop()


##############################
#
# Funcao para ser executada na forma dos comentarios abaixo
#
##############################
def visualiza2(vini, vfin, liga, img, img2):

        top = Toplevel()

        imgg = PhotoImage(file=img)
        imgg2 = PhotoImage(file=img2)

        w = imgg.width()*2+60
        h = imgg.height()+40
        # get screen width and height
        ws = top.winfo_screenwidth()
        hs = top.winfo_screenheight()
        # calculate position x, y
        x = (ws/2) - (w/2)
        y = (hs/2) - (h/2)
        top.geometry('%dx%d+%d+%d' % (w, h, x, y))
        #top.geometry("1000x600")
```

```
        pp = Canvas(top, width = imgg.width()+15, height = imgg.height()+15)
        pp.place(x=20,y=20)
        pp.create_image(0, 0, anchor=NW, image=imgg)

        pp2 = Canvas(top, width = imgg.width()+15, height = imgg.height()+15)
        pp2.place(x=imgg.width()+20+20,y=20)
        pp2.create_image(0, 0, anchor=NW, image=imgg2)


        for i in range(len(vini)):
                vini[i] = [vini[i][1], vini[i][0]]
                vfin[i] = [vfin[i][1], vfin[i][0]]
                if vini[i][0] == 0:
                        vini[i][0] = 1
                if vini[i][1] == 0:
                        vini[i][1] = 1
                if vfin[i][0] == 0:
                        vfin[i][0] = 1
                if vfin[i][1] == 0:
                        vfin[i][1] = 1


        for i in liga:
                id = pp.create_polygon([vini[i[0]], vini[i[1]], vini[i[2]]], fill='', outline='red')
                id = pp2.create_polygon([vfin[i[0]], vfin[i[1]], vfin[i[2]]], fill='', outline='red')

        #Por algum motivo desconhecido, eh necessario voltar do jeito que estava....
        for i in range(len(vini)):
                vini[i] = [vini[i][1], vini[i][0]]
                vfin[i] = [vfin[i][1], vfin[i][0]]
                if vini[i][0] == 1:
                        vini[i][0] = 0
                if vini[i][1] == 1:
                        vini[i][1] = 0
                if vfin[i][0] == 1:
                        vfin[i][0] = 0
                if vfin[i][1] == 1:
                        vfin[i][1] = 0

        top.mainloop()

'''
vini = [[0, 0], [0, 295], [399, 295], [399, 0], [56, 79], [28, 122], [25, 159], [36, 197], [45,
231], [85, 240], [78, 165], [78, 204], [96, 57], [87, 102], [123, 124], [126, 195], [143,
108], [137, 124], [146, 141], [154, 124], [148, 173], [141, 192], [150, 203], [153, 190],
[177, 135], [174, 158], [182, 179], [192, 157], [199, 120], [202, 154], [207, 180], [216,
151], [132, 158], [119, 245], [150, 239], [196, 219], [235, 193], [139, 60], [171, 65],
```

```
[205, 91], [247, 132], [256, 170], [247, 77], [263, 39], [325, 55], [353, 135], [318, 240],
[273, 243], [283, 281], [232, 178], [225, 58], [10, 142], [222, 26]]

vfin = [[0, 0], [0, 295], [399, 295], [399, 0], [70, 40], [99, 77], [120, 127], [120, 179],
[80, 253], [119, 250], [161, 155], [155, 212], [126, 50], [151, 96], [180, 120], [181,
203], [205, 103], [191, 120], [204, 135], [219, 118], [210, 182], [193, 198], [203, 213],
[220, 200], [238, 144], [234, 163], [242, 175], [250, 162], [273, 136], [261, 161], [273,
177], [272, 159], [189, 159], [170, 240], [209, 234], [237, 229], [275, 208], [183, 55],
[225, 65], [261, 76], [298, 137], [300, 183], [279, 45], [317, 25], [334, 58], [347, 155],
[308, 231], [267, 254], [285, 279], [281, 188], [245, 34], [52, 146], [246, 10]]

liga = [array([ 0,  3, 52,  0]), array([52,  3, 43, 52]), array([ 3, 44, 43,  3]), array([ 0, 52,
12,  0]), array([12, 52, 37, 12]), array([43, 50, 52, 43]), array([37, 52, 38, 37]),
array([50, 38, 52, 50]), array([43, 42, 50, 43]), array([ 0, 12,  4,  0]), array([38, 50, 39,
38]), array([ 4, 12, 13,  4]), array([42, 39, 50, 42]), array([38, 16, 37, 38]), array([44,
42, 43, 44]), array([37, 13, 12, 37]), array([13, 37, 16, 13]), array([13, 16, 14, 13]),
array([14, 16, 17, 14]), array([17, 16, 19, 17]), array([39, 16, 38, 39]), array([39, 19,
16, 39]), array([39, 28, 19, 39]), array([0, 4, 5, 0]), array([28, 24, 19, 28]), array([42,
40, 39, 42]), array([28, 39, 40, 28]), array([17, 19, 18, 17]), array([13,  5,  4, 13]),
array([ 3, 45, 44,  3]), array([40, 31, 28, 40]), array([24, 18, 19, 24]), array([31, 29, 28,
31]), array([14, 17, 18, 14]), array([24, 28, 29, 24]), array([29, 27, 24, 29]), array([14,
18, 32, 14]), array([51,  5,  6, 51]), array([27, 25, 24, 27]), array([44, 40, 42, 44]),
array([ 0,  5, 51,  0]), array([18, 24, 25, 18]), array([40, 44, 45, 40]), array([ 5, 13, 10,
5]), array([14, 10, 13, 14]), array([32, 18, 20, 32]), array([25, 20, 18, 25]), array([ 6,  5,
10,  6]), array([40, 41, 31, 40]), array([32, 10, 14, 32]), array([41, 49, 31, 41]),
array([25, 27, 26, 25]), array([31, 30, 29, 31]), array([27, 29, 30, 27]), array([49, 30,
31, 49]), array([26, 27, 30, 26]), array([20, 25, 26, 20]), array([45, 41, 40, 45]),
array([23, 21, 20, 23]), array([26, 23, 20, 26]), array([32, 20, 15, 32]), array([21, 15,
20, 21]), array([41, 36, 49, 41]), array([ 6, 10,  7,  6]), array([10, 32, 15, 10]),
array([23, 22, 21, 23]), array([30, 49, 36, 30]), array([ 7, 10, 11,  7]), array([15, 11, 10,
15]), array([26, 30, 35, 26]), array([15, 21, 22, 15]), array([26, 22, 23, 26]), array([36,
35, 30, 36]), array([22, 26, 35, 22]), array([ 7, 11,  8,  7]), array([45, 46, 41, 45]),
array([36, 41, 47, 36]), array([46, 47, 41, 46]), array([15, 22, 34, 15]), array([35, 34,
22, 35]), array([34, 33, 15, 34]), array([15,  9, 11, 15]), array([ 8, 11,  9,  8]), array([ 9,
15, 33,  9]), array([51,  6,  7, 51]), array([35, 36, 47, 35]), array([46, 48, 47, 46]),
array([45,  2, 46, 45]), array([8, 1, 7, 8]), array([51,  7,  1, 51]), array([9, 1, 8, 9]),
array([35, 47, 48, 35]), array([48, 46,  2, 48]), array([34, 35, 48, 34]), array([ 3,  2, 45,
3]), array([33,  1,  9, 33]), array([33, 34, 48, 33]), array([48,  1, 33, 48]), array([ 0, 51,
1,  0]), array([ 1, 48,  2,  1])]


print 'Pontos:',len(vini)
print 'Poligonos:',len(liga)

visualiza2(vini,vfin,liga, 'Imagens/ian.pgm', 'Imagens/cat2.pgm')
'''
```