Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Manipulação interativa de objetos em imagens

Diego Lemos de Almeida Melo

Trabalho de Graduação

Recife

Junho de 2009

Universidade Federal de Pernambuco
Centro de Informática

Diego Lemos de Almeida Melo

Manipulação interativa de objetos em imagens

Trabalho apresentado ao Curso de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Marcelo Walter
Co-Orientador: Silvio de Barros Melo

Recife
Junho de 2009

"To be successful, the first thing to do is fall in love with your work."

(Sister Mary Lauretta)

Aos meus pais, Marco e Alessandra.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, aos meus pais, Marco e Alessandra, e a minha irmã Gabriela, por compreenderem todos os momentos em que precisei estar ausente devido à minha busca pela minha construção profissional. Estes que com muito carinho e apoio, nunca mediram esforços para que eu chegasse a concluir esta etapa da minha vida.

A todos os meus verdadeiros amigos que fiz antes e durante a faculdade, por sempre estarem comigo nos piores e melhores momentos, me dando forças e me apoiando.

Ao meu orientador Marcelo Walter, pela sabedoria passada e excelente acompanhamento das minhas pesquisas. Pela paciência, dedicação e incentivo que sempre teve desde o início da realização deste trabalho.

Ao professor Silvio Melo, pelas dicas e ajudas preciosas que foram fundamentais para a conclusão deste trabalho.

A todos que acreditaram em mim e que de alguma forma me ajudaram na construção deste trabalho.

Muito Obrigado!

Resumo

O objetivo geral deste trabalho foi a investigação de técnicas para edição e manipulação de imagens interativamente, como se os objetos presentes nas imagens fossem tridimensionais. Estas técnicas simulam a manipulação da forma, garantindo ao usuário a sensação de estar manipulando um objeto real, e não uma imagem bidimensional. Para conseguir este feito, durante a manipulação, certas propriedades geométricas são mantidas, por exemplo, a relação espacial entre os vértices que definem a malha do objeto. A forma inicial torna-se “rígida”, e por este motivo estas técnicas ficaram conhecidas por manipulação “tão rígida quanto possível” (*as-rigid-as possible*) de objetos. A manipulação interativa de imagens é muito vantajosa comparada às técnicas manuais de edição de imagens, pois permite maior flexibilidade, acelerando o processo de criar animações em aplicações gráficas.

Palavras chave: Computação Gráfica, Manipulação Interativa de Imagens, Modelagem de Objetos 2D, Inpainting.

Abstract

The main goal of this work was the investigation of techniques for interactive editing and manipulation of images, as the objects in the images were three-dimensional objects. These techniques simulate the manipulation of form, giving the user a feeling of manipulating a real object, not a two-dimensional image. To achieve this end, during the manipulation, some geometric properties are maintained, for example, rigidity. The interactive manipulation of images has many advantages when compared to manual techniques for image editing, since it allows greater flexibility, speeding up the process of creating animations in graphics applications.

Keywords: Computer Graphics, Image Editing, Object Modeling, Shape Manipulation, Inpainting.

Sumário

Lista de figuras

1- Introdução

Em diversas aplicações gráficas com animações, particularmente as que são executadas em dispositivos móveis, cuidados especiais devem ser tomados devido ao *hardware* limitado, tais como otimizações no processamento, utilização de pouca memória, entre outros fatores que diminuem o custo computacional. O uso de imagens, ao invés de objetos tridimensionais realistas, ameniza o problema, já que os recursos necessários para o armazenamento de imagens são menores do que para objetos 3D. A dificuldade, entretanto, passa a ser a manipulação de imagens proporcionar o mesmo efeito da manipulação do objeto real. Neste contexto, uma manipulação inteligente de imagens é uma boa solução.

A técnica para manipulação interativa de imagens apresentada recentemente no trabalho de Igarashi e colegas [1] aborda este tema de forma original, ao tratar uma imagem como um objeto rígido, que pode ser manipulado como um objeto real, interativamente. Assim, esta abordagem facilitou tarefas até então de execução manual.

O método atual de manipulação de imagens utiliza a técnica de manipulação de *sprites* [21](especialmente quando se trata de jogos 2D) onde uma animação é feita por uma série de imagens que são criadas à mão, uma a uma, com a variação necessária para proporcionar a sensação de movimento. Abaixo segue um exemplo de *sprites* utilizados na criação do jogo *megaman* [20].

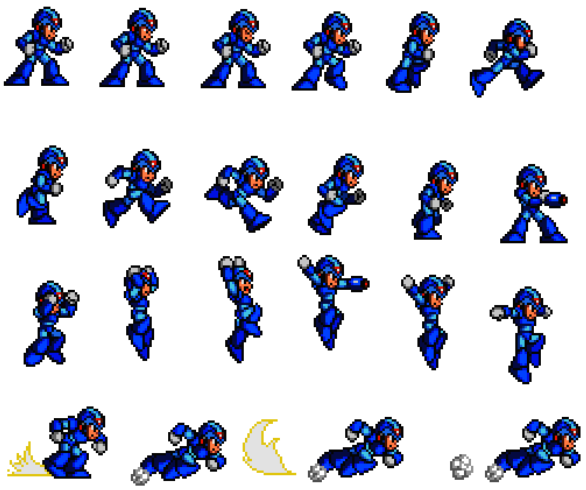


Figura 1 Exemplo de *sprites* do jogo *megaman*.

Este processo de criação manual dos *sprites* possui um custo computacional e artístico muito caro, logo uma manipulação inteligente de imagens é uma boa forma de reduzir o custo artístico além de garantir um melhor desempenho computacional. Além disto, é garantido um maior dinamismo, pois a manipulação pode ser feita de uma forma interativa, auxiliando assim no desenvolvimento de aplicações que utilizem imagens 2D, a grande maioria ainda em dispositivos móveis.

O trabalho de Igarashi e colegas [1] forneceu o embasamento teórico sobre o assunto e estabeleceu a metodologia a ser seguida, com a identificação precisa das etapas a serem seguidas durante o desenvolvimento do projeto, além de sugerir algumas técnicas e cálculos para execução de tais etapas. A idéia básica da técnica é representar a imagem-objeto como um conjunto de triângulos que estão sujeitos a determinadas restrições na sua manipulação, causando a sensação visual de manipulação rígida do objeto. Abaixo segue a interface da aplicação desenvolvida para o presente trabalho:

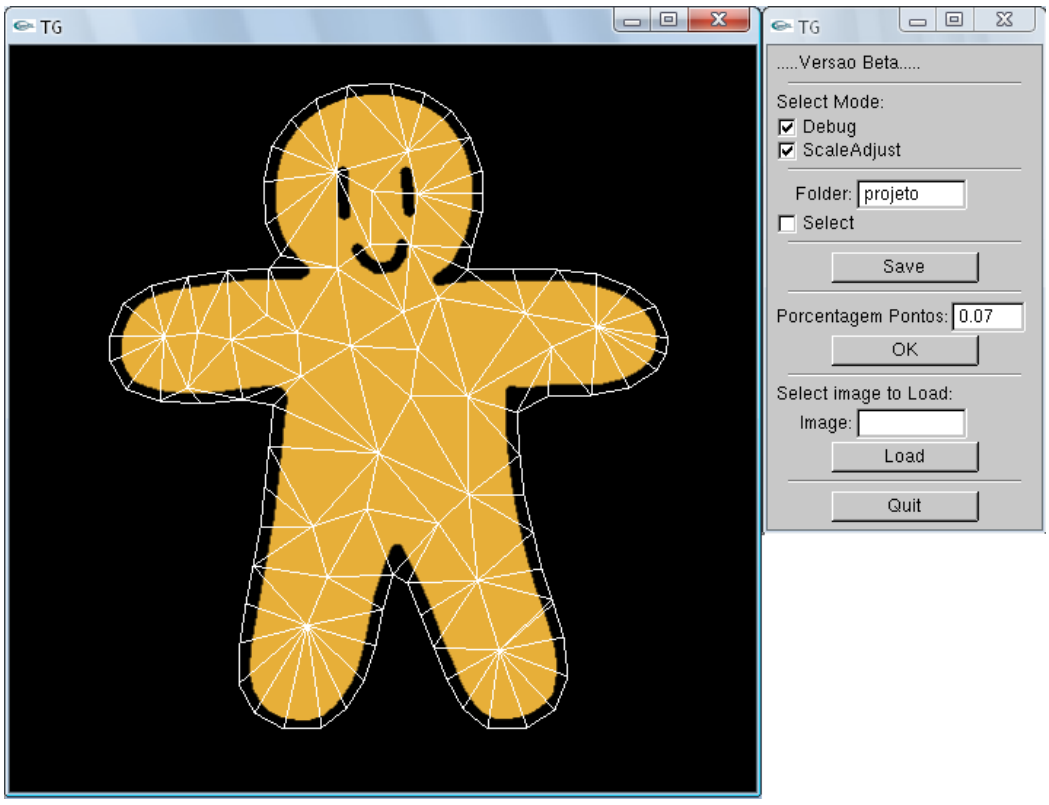


Figura 2 Interface da aplicação.

Este trabalho foi capaz de aliar duas técnicas de edição de imagens, a técnica de *shape manipulation* proposta por Igarashi e colegas [1] e técnicas de *inpainting* [13], criando assim um sistema capaz de manipular objetos em imagens com possíveis obstruções de partes destes objetos, sendo esta uma das principais contribuições deste trabalho.

No capítulo 2 é feita uma revisão de trabalhos relacionados, e em particular, o detalhamento das principais pesquisas que influenciaram fortemente este estudo. No capítulo 3 é descrita a abordagem proposta neste relatório através da ferramenta desenvolvida. No capítulo 4 são analisados os resultados obtidos com a ferramenta. No capítulo 5 encontram-se as conclusões do trabalho e são discutidas possibilidades de trabalhos futuros.

2- Revisão Bibliográfica

Existem inúmeros estudos sobre técnicas de *shape manipulation* e *inpainting* (este trabalho aliou essas duas técnicas a fim de alcançar os objetivos desejados), sendo descritos aqui os trabalhos de maior relevância. Na seção 2.1 descreveremos as principais abordagens sobre *shape manipulation* enquanto na seção 2.2 descreveremos as principais abordagens sobre *inpainting.*

2.1- Shape Manipulation

Constantemente são desenvolvidos estudos sobre edição, criação e manipulação de imagens, pois esta é uma área de extrema importância para a computação gráfica. Uma das abordagens adotadas foi a *shape manipulation*, que consiste basicamente em representar a imagem-objeto como um conjunto de triângulos que são manipulados interativamente de forma que se mantenha alguma das características do objeto original, causando a sensação de estar manipulando o objeto real.

2.1.1– As-Rigid-As-Possible Shape Manipulation

O trabalho do Igarashi e colegas [1] serviu como a principal base para o estudo aqui proposto. Nele é apresentado um sistema interativo em que o usuário move e deforma um objeto bidimensional mantendo propriedades da imagem, como por exemplo, a rigidez. O objeto é representado por uma malha formada de vértices e triângulos em que o usuário poderá fixar ou movimentar estes vértices (são os chamados vértices de controle). Assim, o sistema calcula a posição dos demais vértices da malha (são os chamados vértices livres) minimizando a distorção de cada um dos triângulos e conseqüentemente minimizando a distorção de toda a malha.

Existem outras maneiras de simular a manipulação aqui proposta, entretanto elas são muito lentas e a idéia é obter um sistema de manipulação de objetos em tempo real e interativo. Assim, o algoritmo foi dividido em duas

etapas, a fim de transformar um problema que possui complexidade quadrática em um sistema de equações lineares. A primeira etapa resolve o problema da rotação enquanto a segunda etapa resolve o problema da escala.

Mais detalhes sobre o funcionamento do algoritmo são demonstrados na seção 3.5. Na figura 3 abaixo podemos ver um exemplo da aplicação do método em funcionamento.

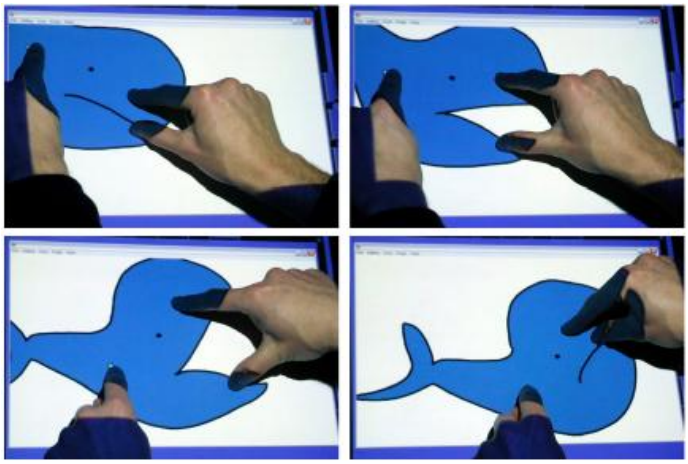


Figura 3 Aplicação do Igarashi e colegas [1].

2.1.2– 2D Shape Deformation Using Nonlinear Least Squares Optimization

O trabalho do Weng e colegas [6] surgiu posteriormente ao trabalho do Igarashi e colegas [1] e trouxe uma nova abordagem para o problema da manipulação interativa de objetos. Este algoritmo visa preservar duas propriedades do objeto, as coordenadas de Laplace [17] para as curvas da borda do objeto e as áreas locais do interior do objeto, que são representadas em conjunto por uma função não quadrática de energia. Assim, um método interativo de Gauss-Newton [18] é utilizado a fim de minimizar o erro da função não linear de energia.

O resultado disto foi um sistema de deformação interativa de objetos, em que esta movimentação seja fisicamente possível, ou seja, como se estivesse movimentando um objeto real. Além de preservar as propriedades

locais da forma do objeto, foi também criado um artifício para preservar a área global do objeto.

Assim, a idéia básica do trabalho do Weng e colegas [6] foi desenvolver um algoritmo de manipulação de imagens em que sejam mantidas algumas propriedades originais da forma do objeto, tais como as formas das curvas da borda, as áreas locais do interior do objeto e a área global do objeto. Na figura 4 podemos observar um resultado obtido pelo método do Weng e colegas [6].

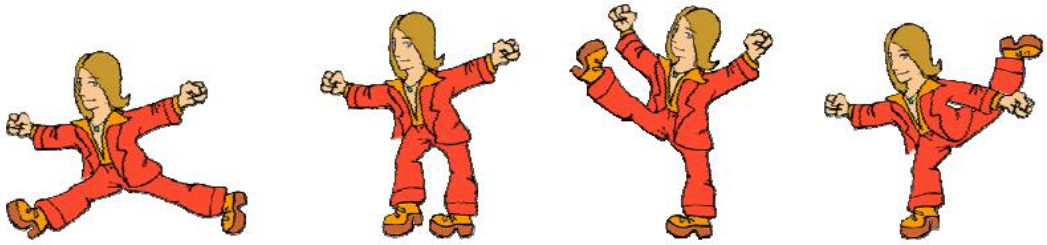


Figura 4 Resultado obtido pelo trabalho do Weng e colegas [6].

Comparativamente, podemos dizer que este trabalho trouxe um avanço em relação ao trabalho do Igarashi e colegas [1], pois ele é capaz de preservar as propriedades globais e locais dos objetos, ao contrário da técnica do Igarashi e colegas [1] que apenas preserva as propriedades globais, como podemos observar na figura 5:

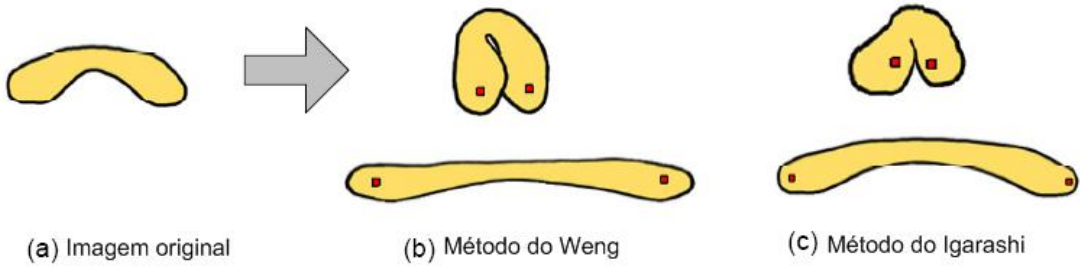


Figura 5 Comparação entre o método de Weng e colegas [6] e Igarashi e colegas [1].

2.2- Inpainting

A reconstrução de partes ou danificações nas imagens é uma antiga prática amplamente utilizada em trabalhos artísticos de restauração. Também conhecida como *inpainting* ou *retouching*, esta atividade consiste em preencher áreas vazias ou modificar áreas danificadas de maneira que não fiquem perceptíveis à um observador não familiarizado com as imagens originais.

As técnicas de *inpainting* são usadas na restauração de fotografias, filmes e pinturas, a fim de remover as oclusões, tais como texto, legendas, selos e publicidade, além de poder ser usadas para produzir efeitos especiais. Na figura 6 podemos observar um resultado obtido pela técnica de *inpainting* de Bertalmio e colegas [12].

Figura 6 Exemplo de técnica de *inpainting* de Bertalmio e colegas [12]. À esquerda temos uma imagem rabiscada e à direita a imagem após a reconstrução.

2.2.1– Texture Synthesis by Non-parametric Sampling

O primeiro trabalho que abordou a idéia de preencher buracos nas imagens (*hole filling*) foi o trabalho do Efros e colegas [11], que propôs um método não-paramétrico de síntese de textura. Este processo de síntese de textura gera uma nova imagem a partir de uma semente inicial que cresce *pixel* a *pixel*. É adotado o modelo randômico de Markov [16], sendo a condição de distribuição de um determinado *pixel*, dado o conjunto de todos os *pixels* vizinhos sintetizados, estimada através da consulta da imagem exemplo para que se encontre todas as áreas similares. O grau de aleatoriedade é controlado

por um único parâmetro que é perceptualmente intuitivo. O método visa preservar a estrutura local, tanto quanto possível, e produz bons resultados para uma ampla variedade de texturas sintéticas e do mundo real. A principal parte do algoritmo é apresentada abaixo:

```
function GrowImage(SampleImage,Image,WindowSize)
  while Image not filled do
    progress = 0
    PixelList = GetUnfilledNeighbors(Image)
    foreach Pixel in PixelList do
      Template = GetNeighborhoodWindow(Pixel)
      BestMatches = FindMatches(Template, SampleImage)
      BestMatch = RandomPick(BestMatches)
      if (BestMatch.error < MaxErrThreshold) then
        Pixel.value = BestMatch.value
        progress = 1
      end
    end
    if progress == 0
      then MaxErrThreshold = MaxErrThreshold * 1.1
  end
  return Image
end
```

Podemos observar que *SampleImage* contem a imagem que utilizamos de amostragem e *Image* é a imagem mais vazia que pretendemos preencher (se sintetizar a partir do zero, ela deve conter uma semente 3-por-3 no centro, escolhida aleatoriamente do *SampleImage*, caso contrário deverá conter todos os *pixels* conhecidos da imagem de amostra). A *WindowSize* é o tamanho da janela de vizinhos, sendo o único parâmetro a ser escolhido pelo usuário.

A função *GetUnfilledNeighbors* retorna uma lista de todos os *pixels* a serem preenchidos que contenham vizinhos preenchidos. A lista é então ordenada pelo número decrescente de vizinhos preenchidos (por exemplo, se um determinado *pixel* tiver 4 vizinhos preenchidos, aparecerá primeiro na lista do que um *pixel* que tiver 3 vizinhos preenchidos). A função *GetNeigborhoodWindow* retorna uma janela de tamanho *WindowSize* em torno de um determinado *pixel*. *RandomPick* escolhe aleatoriamente um elemento da lista e *FindMatches* é como segue:

```
function FindMatches(Template,SampleImage)
  ValidMask = 1s where Template is filled, 0s otherwise
  GaussMask = Gaussian2D(WindowSize,Sigma)
  TotWeight = sum i,j GaussiMask(i,j)*ValidMask(i,j)
  for i,j do
    for ii,jj do
      dist = (Template(ii,jj)-SampleImage(i-ii,j-jj))^2
      SSD(i,j) = SSD(i,j) +
dist*ValidMask(ii,jj)*GaussMask(ii,jj)
    end
    SSD(i,j) = SSD(i,j) / TotWeight
  end
  PixelList = all pixels (i,j) where SSD(i,j) <=
min(SSD)*(1+ErrThreshold)
  return PixelList
end
```

Gaussian2D gera uma janela Gausiana bidimensional de um determinado tamanho, centralizada e com um dado desvio-padrão (em *pixels*).

Seguem na figura 7 os resultados obtidos na síntese de texturas através da abordagem do Efros e colegas:

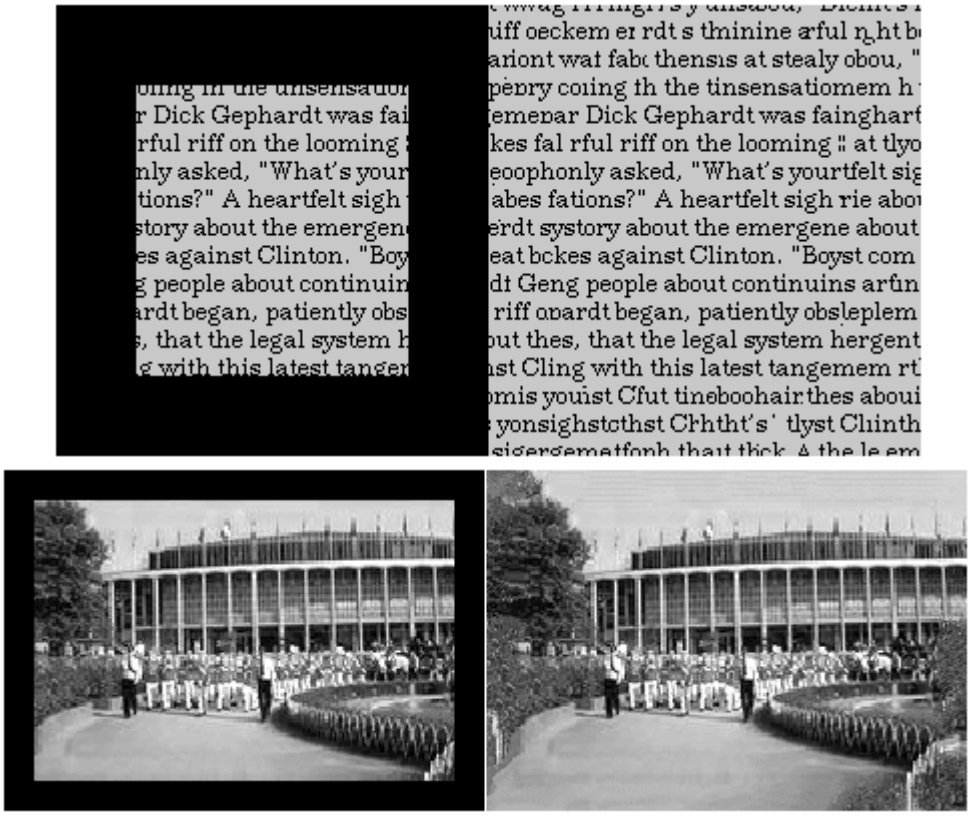


Figura 7 Resultados obtidos pelo algoritmo do Efros e colegas [11].

2.2.2– Image Inpainting

Várias outras técnicas de preenchimento de regiões surgiram desde então, como por exemplo algoritmos de restauração de filmes, os quais trabalham em regiões relativamente pequenas e utilizam as informações existentes em vários quadros. Surgiram também algoritmos baseados na síntese de textura que são capazes de preencher grandes regiões, mas exigem que o usuário especifique que parte da textura deve colocar onde.

A técnica proposta por Bertalmio e colegas [12] não exige qualquer intervenção do usuário, dado que a região a ser preenchida já tenha sido selecionada, e por isso ganhou uma certa relevância. O algoritmo é capaz de preencher simultaneamente regiões rodeadas por diferentes origens, sem que o usuário precise indicar "o que colocar onde", além de ser capaz de preencher regiões que possuam estruturas (por exemplo, duas ou mais regiões que possuem partes que se cruzam), embora não seja capaz de preencher grandes áreas de textura.

A idéia geral do algoritmo é a seguinte: supondo ω a região a ser preenchida, e $\theta\omega$ a sua fronteira (pode-se observar que nenhuma hipótese sobre a topologia foi feita), então é feito o prolongamento das linhas que chegam à sua fronteira $\theta\omega$, mantendo-se o ângulo de "chegada". Procede-se desenhando de $\theta\omega$ para ω desta forma, aumentando progressivamente o prolongamento das linhas a fim de impedir que se cruzem mutuamente. Nas figuras 6 e 8 podemos observar os resultados obtidos pelo algoritmo de Bertalmio e colegas [12].

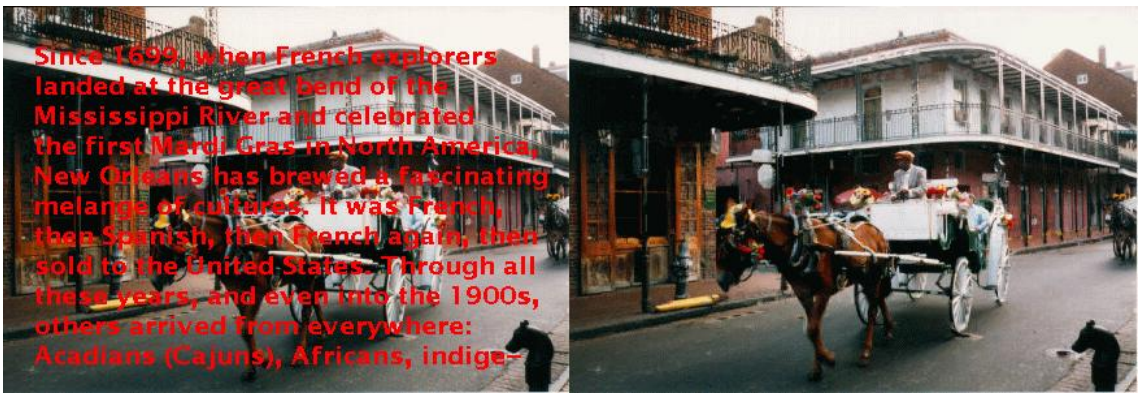


Figura 8 Resultado obtido pelo método de Bertalmio e colegas [12].

2.2.3 – Coordinates for Instant Image Cloning

A clonagem de partes de uma imagem em uma outra imagem é uma meta importante e útil em operações de edição de imagem, e por isso tem recebido consideráveis estudos nestes últimos anos. Esta operação é normalmente realizada através da resolução de uma equação de Poisson [17] utilizando a condição de fronteira de Dirichlet, que interpola suavemente as discrepâncias entre a fronteira da fonte e a área a ser clonada da imagem destino.

Destacamos aqui o trabalho de Zeev Farbman e colegas[13] que desenvolveu uma nova abordagem para o problema baseada em coordenadas, ao invés de resolver um grande sistema linear para realizar a supracitada interpolação, em que o valor da interpolação em cada *pixel* é determinado pela combinação ponderada dos valores ao longo da fronteira.

Mais especificamente, a abordagem é baseada na média do valor das coordenadas (MVC). A utilização de coordenadas é vantajosa em termos de velocidade, de facilidade de implementação, utiliza menos memória e permite, em tempo real, a clonagem de grandes regiões e até a clonagem de vídeos interativos. Na figura 9, podemos observar o resultado da remoção de um objeto da imagem, enquanto na figura 10 podemos observar o resultado da clonagem de objetos da imagem, em que os chifres do animal foram duplicados.

Figura 9 À esquerda temos a imagem original. À direita temos o resultado da remoção do animal utilizando o método do Zeev e colegas [13].

Figura 10 Resultado da clonagem obtida pelo método do Zeev e colegas [13].

Vale ressaltar que esta abordagem, além de ser bem recente, trouxe uma nova solução para o problema da clonagem, uma solução muito mais rápida (podendo ser executada em tempo real) do que a abordagem tradicional utilizando Poisson. Essa abordagem serviu como base para a idéia do método de *inpainting* desenvolvida pelo nosso trabalho.

3- Aplicação

Neste capítulo, será detalhada a solução proposta para a manipulação e edição de objetos em imagens, incluindo a obstrução de partes do objeto. Foi criada então uma aplicação em C++ para implementar, testar e visualizar as técnicas que serão descritas nas seções a seguir.

3.1- Visão Geral do Sistema

A interface gráfica é bastante simples, ilustrada na figura 11. Basta iniciar a aplicação e escolher a imagem a ser carregada que o programa automaticamente triangula o objeto da imagem de acordo com a quantidade de pontos escolhida pelo próprio usuário, de acordo com a sua percepção (é requisito básico das imagens carregadas no programa que elas tenham o fundo transparente).

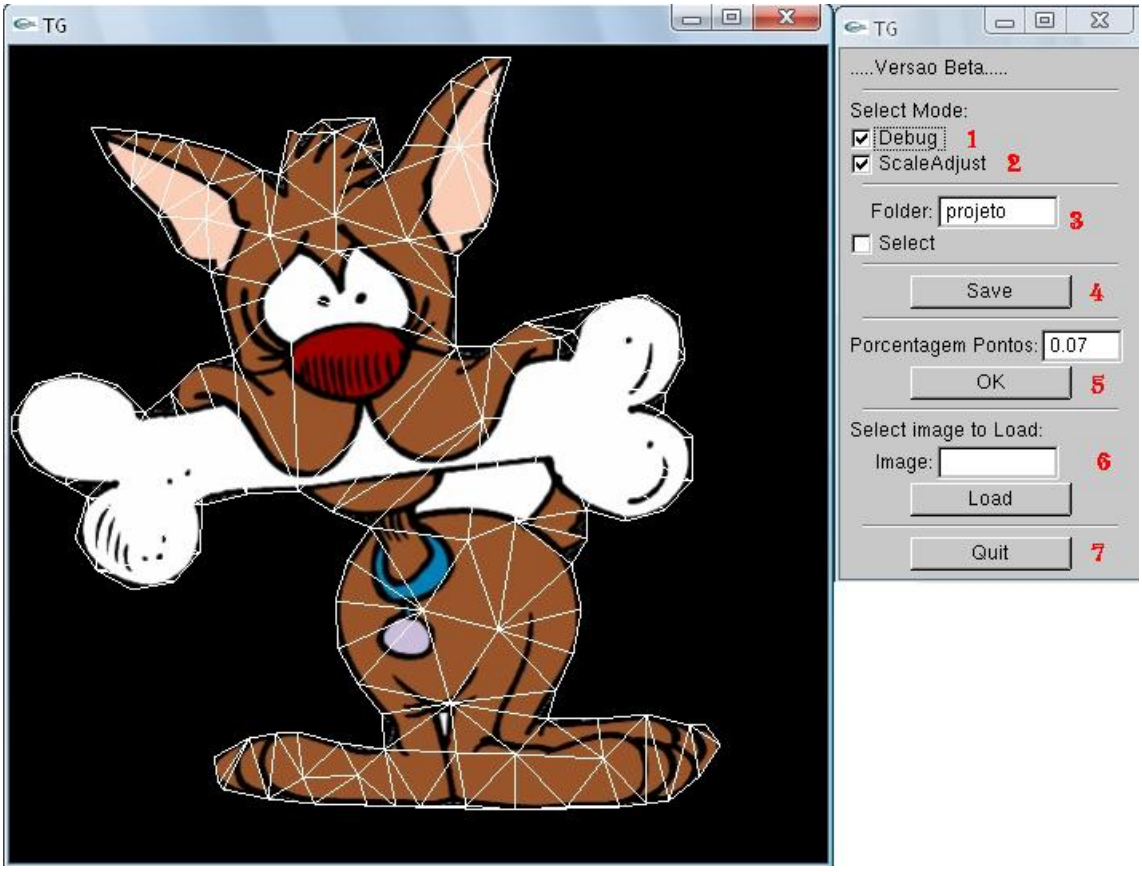


Figura 11 Interface da Aplicação.

A aplicação possui um menu que auxilia o usuário na utilização do programa. Segue abaixo a descrição de cada uma das opções do menu (observar os números na parte direita da figura 11):

(1) O usuário pode escolher se deseja mostrar ou não os triângulos.
(2) O usuário escolhe se deseja manipular a imagem com o ajuste de escala.
(3) O usuário escolhe o nome e cria a pasta onde serão salvas as imagens após a manipulação.
(4) O usuário ao manipular a imagem poderá salvar a imagem na posição em que ela se encontra.
(5) O usuário pode aumentar ou diminuir o número de pontos internos do objeto a fim de melhorar a triangularização.
(6) Se o usuário desejar carregar uma outra imagem, basta digitar o nome da imagem no campo e selecionar o botão Load.
(7) Para sair do programa.

Várias etapas foram seguidas durante o desenvolvimento da aplicação. Primeiramente foi feito um estudo sobre a biblioteca Devil C++ [2] que foi utilizada no carregamento dos dados da imagem na memória para a sua amostragem na tela e posterior manipulação. Precisou-se então identificar as bordas do objeto na imagem e para tanto foi utilizado o algoritmo *marching squares* [15] (seção 3.2). Essa identificação da borda é necessária para que sejam distribuídos pontos na borda de uma forma eqüidistante para que se crie um *"cordão de isolamento"* que será utilizado na etapa do relaxamento. Tendo os pontos da borda, são gerados pontos no interior do objeto de forma randômica para que estes sofram um relaxamento a fim de se distribuírem de uma forma eqüidistante (seção 3.3). Assim, com os pontos da borda e os pontos eqüidistantes no interior do objeto é feita a triangularização desses pontos (seção 3.4) para que se tenha o mapeamento do objeto da imagem em relação aos triângulos gerados.

Tendo agora o objeto e sua respectiva malha (pontos e triângulos gerados nas etapas anteriores) desenvolveu-se o algoritmo de manipulação do objeto (seção 3.5) de forma que se mantenham as propriedades originais do objeto como a rigidez, por exemplo. Existem casos em que a manipulação gera “furos” na imagem e para preencher esses espaços vazios foi desenvolvido um algoritmo de *inpainting* (seção 3.6).

3.2- Marching Squares

O algoritmo de *Marching Squares* [15] foi utilizado para identificação das bordas dos objetos que compõem a imagem. Para explicar este algoritmo partimos do princípio de que o objeto é um conjunto de *pixels* da imagem que possui uma cor diferente da cor do fundo (na nossa aplicação a diferenciação se dá, pois um dos requisitos para o carregamento das imagens é que elas possuam fundo com alfa transparente enquanto os objetos não). Para nossa explicação iremos considerar um exemplo bem simples em que o objeto consiste em um conjunto de *pixels* de cor preta enquanto o fundo consiste em um conjunto de *pixels* de cor branca.

Os *pixels* são percorridos, examinando-se quatro células adjacentes em uma janela 2x2. Para cada posição possível da janela, ela estará em algum dos 16 estados possíveis, como por exemplo, se ela estiver no fundo da imagem, todos os quatro pixels da janela estarão brancos. Por outro lado, se a janela estiver completamente dentro do objeto, todos os *pixels* estarão pretos. Em outro caso a janela será formada pela combinação de *pixels* pretos e brancos.

Esses 16 estados, ilustrados na figura 13, são usados para determinar em qual localização da imagem a janela está e para onde ela deve ir, para examinar outros *pixels*. Por exemplo, se começamos a percorrer o fundo da imagem da esquerda para a direita, até que se atinja a borda esquerda do objeto, que pode ter os dois *pixels* da esquerda brancos, e os dois da direita pretos. Este estado (o que poderíamos chamar 0110, ou 6) é utilizado para indexar uma tabela que nos diz qual será a próxima etapa. Neste caso, ele pode retornar 0 para a direção x, e 1 para a direção y, ou seja, a janela é movida para baixo por uma célula. Este processo é repetido, com cada combinação de branco e preto até que toda a borda do objeto seja percorrida, o

que acontece quando a janela volta para a posição em que começou (no nosso exemplo quando ela voltar para o estado 0101 e estiver sobre os mesmos *pixels* que começou), conforme ilustrado na figura 12.

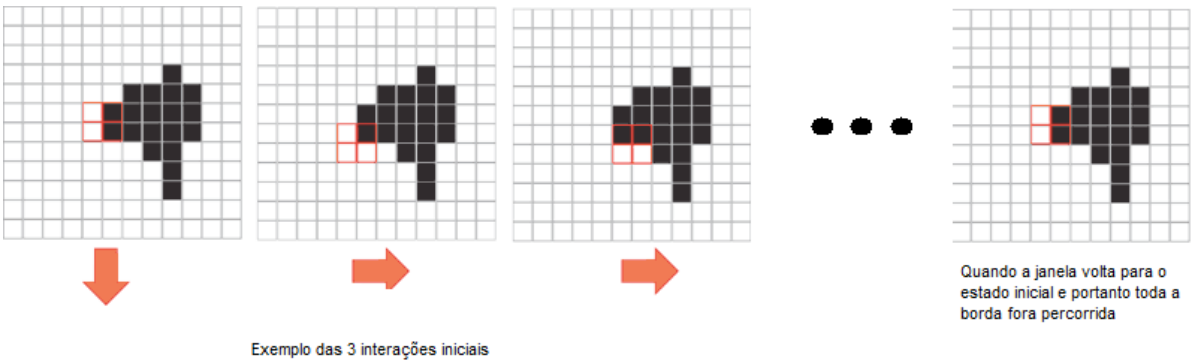


Figura 12 Exemplo da execução do algoritmo em uma estrutura simples (em que são formados apenas de *pixels* pretos e brancos).

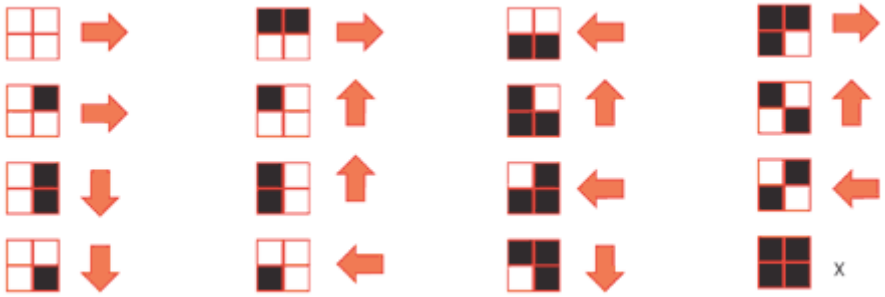


Figura 13 Os 16 estados possíveis da janela e a respectiva direção para onde a janela deve mover-se na próxima interação. O estado X é um estado ilegal.

No nosso caso, precisamos identificar a borda do objeto de forma que se forme um "*cordão de isolamento*" na hora de fazer o relaxamento dos pontos no interior do objeto. Contudo, estes pontos da borda também farão parte da malha, portanto estarão presentes na etapa da triangularização. Assim, devemos escolher apenas alguns pontos da borda para que pertençam à malha, de forma que estes pontos fiquem distribuídos de uma forma eqüidistante. Para a escolha destes pontos utilizamos um fator que determina o intervalo que será observado para definir se um determinado ponto da borda fará ou não parte da malha. Na figura 14 ilustramos um resultado do algoritmo de *marching squares* utilizando o fator de 0,02, enquanto na figura 15 o fator

utilizado foi de 0,01. Este fator representa a porcentagem de pontos da borda do objeto que pertencerá à malha (um fator de 0,02 significa dizer apenas 2% dos pontos da borda será escolhida para fazer parte da malha). Podemos confirmar isso ao observarmos que a borda da figura 15 (fator 0,02) possui exatamente o dobro de pontos da borda da figura 15 (fator 0,01).

Figura 14 Exemplo dos pontos encontrados pelo algoritmo de *marching squares* [15] utilizando o fator de 0,02 (51 pontos).

Figura 15 Exemplo dos pontos encontrados pelo algoritmo de *marching squares* [15] utilizando o fator de 0,01 (102 pontos).

3.3- Relaxamento dos pontos

De posse dos pontos que formam a borda do objeto e dos pontos gerados randomicamente no interior do objeto, precisamos relaxar estes pontos do interior de forma que eles se distribuam de forma eqüidistante para depois fazermos a triangularização. Para a criação dos pontos aleatórios no interior do objeto, o usuário especifica a quantidade desejada e o sistema distribui uniformemente na área do objeto.

A idéia básica do algoritmo consiste em criar um *"campo de forças repulsivas"* para cada ponto, de forma que os pontos se empurrem mutuamente a fim de encontrar uma posição ideal para eles (fiquem de forma eqüidistante, ou seja, nenhum ponto estará dentro do campo de força do outro), conforme ilustrado na figura 16. O desafio então passa a ser encontrar um raio adequado para os *"campos de força"* dos pontos de forma que se obtenha uma distribuição satisfatória.

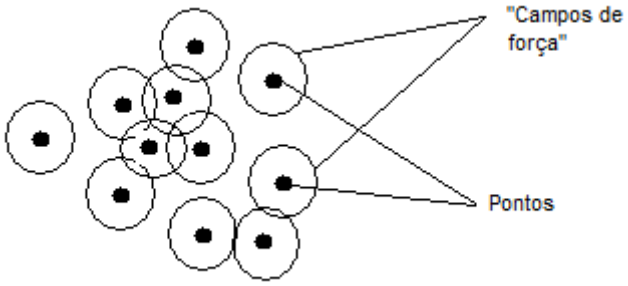


Figura 16 Exemplo de pontos com seus respectivos "campos de força".

O algoritmo é composto basicamente de duas funções: a função de *relaxamento* e a função *pegarPosicao* que retorna a posição ideal para um ponto desejado. Assim, a função *relaxamento* simplesmente pega a posição ideal de cada um dos pontos da malha através da função *pegarPosicao* e atualiza-os, como podemos observar no algoritmo abaixo:

```
relaxamento()
        Para cada ponto do array de pontos faça:
                pontos[indice] ← pegarPosicao(pontos[indice].x pontos[indice].y, indice)
end function
```

Por outro lado, a função *pegarPosicao* tem o dever de encontrar as posições dos pontos da malha, e para isso ela simula a criação de um *"campo de força"* para cada ponto. Assim, a variável *raio* representa o raio do *"campo de força"* de cada um dos pontos. É importante observar que todos os pontos da malha devem interagir entre si (ou seja, devem "empurrar" uns aos outros), e portanto, para encontrarmos a posição resultante de um determinado ponto devemos fazer a interação deste ponto com os demais pontos da malha. Essa interação se faz da seguinte forma: se a distância entre os dois pontos for maior do que o raio do "campo de força" não faça nada, caso contrário (neste caso o ponto estará dentro do "campo de força" e precisará ser "empurrado" para fora) "empurre" o ponto com uma "força" definida em x por *dxi2* e em y por *dyi2.* Ao final da interação com todos os pontos teremos a "força resultante" dos "campos de força" que empurrará o ponto. Essa "força resultante" é definida pelas variáveis *somaX* e *somaY* como podemos observar no algoritmo a seguir:

```
pegarPosicao(x, y, indice)
        i ← 0
        wd ← 0.07
        fator ← 2
        raio = fator * √(área da objeto / número de pontos)
        while(i < numeroDePontos){
                if(i != indice){
                        dxi ← x - pontos[i].pX
                        dyi ← y - pontos[i].pY
                        di ← √(dxi² + dyi²)
                                if (di < raio){
                                        fi ← 1 - (di/r)
                                        dxi2 ← (dxi/di) * fi * r
                                        dyi2 ← (dyi/di) * fi * r
                                        somaX ← somaX + dxi2
                                        somaY ← somaY + dyi2
                                end if
                end if
                i ← i + 1
        end while
                x2 ← x + (wd * somaX)
                y2 ← y + (wd * somaY)

                retorna Ponto(x2,y2)
end function
```

Vale observar que as variáveis *wd* e *fator* são variáveis de ajuste do algoritmo. Dependendo de como se queira o relaxamento (pontos mais próximos ou mais distantes uns dos outros) basta mudarmos estes parâmetros. Uma outra coisa a se observar é que a execução da função de relaxamento é feita em diversas iterações até que se obtenha um resultado satisfatório, visto que geralmente uma única execução não gera resultados bons. Em nossos exemplos utilizamos em média 10 iterações. A figura 17 ilustra a distribuição dos pontos durante as interações, em que inicialmente tem-se os pontos distribuídos randomicamente, depois tem-se a posição dos pontos após a primeira interação e por fim a distribuição dos pontos após a décima interação.

Importante observar que durante o relaxamento, os pontos da borda permanecem estáticos, impedindo que os pontos dentro do objeto saiam do mesmo.

Figura 17 Pontos distribuídos randomicamente (a). Pontos após a primeira execução do algoritmo de relaxamento (b). Pontos após a décima interação do algoritmo de relaxamento (c).

3.4- Triangularização de Delaunay

De posse do polígono convexo formado pelos pontos da borda do objeto e os pontos no interior do objeto já relaxados (distribuídos de forma eqüidistante, como podemos ver na figura 18), precisamos então fazer a triangularização dos mesmos a fim de criar a malha que representará o objeto, como visto na figura 19.

Figura 18 Exemplo de objeto em que já foram identificados os pontos da sua borda e já foram relaxados os pontos do seu interior.

Figura 19 O objeto com sua respectiva malha.

A triangularização utilizada no projeto foi a de Delaunay [14]. A seguir apresentamos uma breve explicação de como funciona este algoritmo.

Primeiramente, a partir da distribuição dos pontos, encontram-se todos os triângulos definidos por três pontos da distribuição, de tal forma que um círculo passando por estes três pontos não inclua nenhum outro ponto, conforme ilustrado na figura 20.

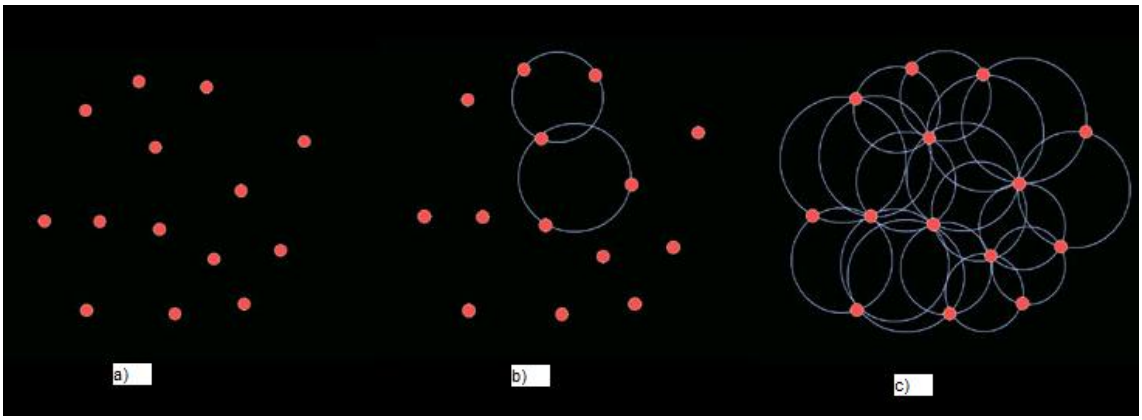


Figura 20 (a) Distribuição inicial dos pontos. (b) Encontrando os círculos que passam por três pontos sem incluir nenhum ponto no interior. (c) Final do processo em que todos os círculos são encontrados.

Depois, para cada conjunto de três pontos satisfazendo a condição de Delaunay acima, gere um triângulo, conforme ilustrado na figura 21.

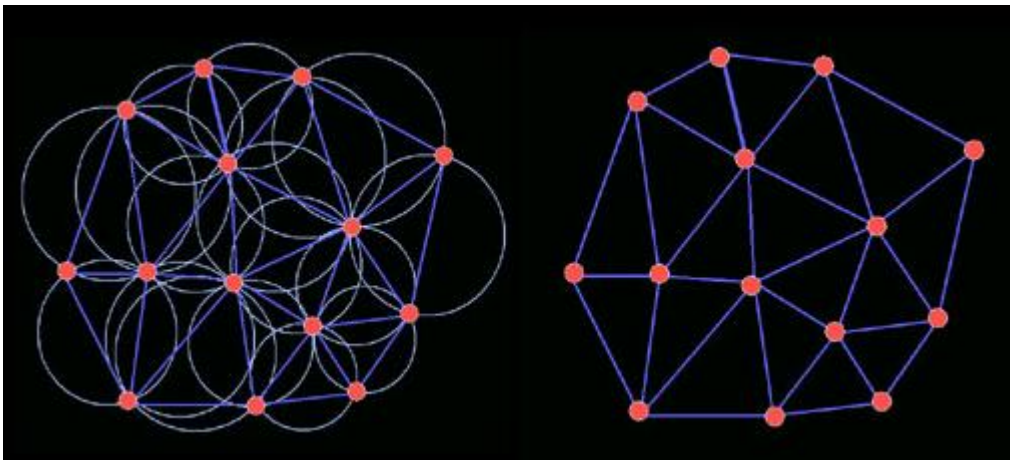


Figura 21 Geração dos triângulos através dos círculos anteriormente gerados.

Vale observar que o número de triângulos de uma triangularização de n vértices depende do número de vértices h na envoltória convexa. Na figura 22 ilustramos este conceito. Na parte esquerda da figura temos uma estrutura com 8 vértices sendo 6 deles pertencentes à envoltória convexa, enquanto na parte direita temos uma estrutura com 8 vértices sendo 5 deles pertencentes à envoltória convexa.

Para a implementação da técnica, utilizamos o programa Qhull [19]. Assim, fornecemos ao Qhull o conjunto de pontos da nossa malha e ele nos retorna o conjunto de triângulos formados por estes pontos.

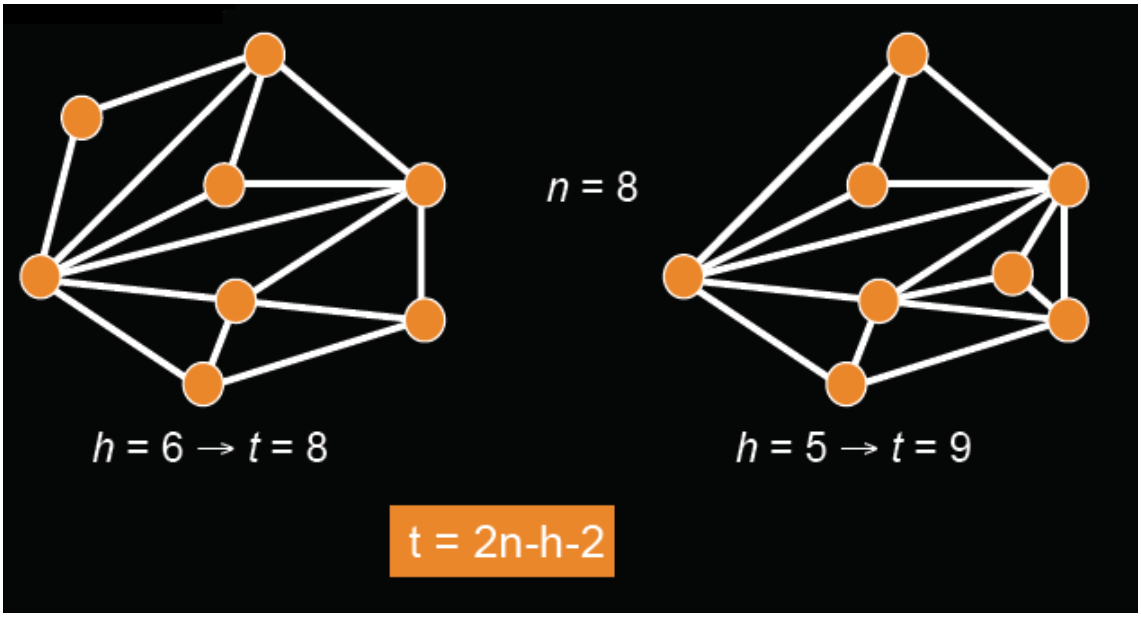


Figura 22 Calculando a quantidade de triângulos que serão gerados.

3.5- Algoritmo de Manipulação

A primeira coisa a ser feita quando carregamos uma imagem é gerar os vértices e triângulos que representarão a malha do objeto. Essa malha definirá todas as características e propriedades do objeto inicialmente. O usuário então poderá selecionar os pontos de controle e deformar a malha. Portanto, tendo os dados da malha, a entrada do algoritmo consiste no conjunto de vértices de controle (figura 23a) e a saída consiste nas novas posições dos vértices livres, de forma que seja minimizada a distorção (erro) associada à todos os triângulos da malha. Dessa forma, o desafio passa a ser encontrar a distorção (erro) para cada um dos triângulos da malha, pois assim teremos o erro de toda a malha.

A estratégia adotada consiste em representar o erro, que é uma função quadrática, através de operação de matrizes, simplificando assim a resolução do problema de minimização do erro. Por ser impossível representar todos os erros de distorção (rotação, translação e escala) em uma única função, o problema foi dividido em dois, problema de rotação (seção 3.5.1) e problema de escala (seção 3.5.2).

Então, o algoritmo funciona basicamente da seguinte forma: dadas as coordenadas dos vértices de controle, o primeiro passo gera um resultado intermediário que minimiza o erro e permite a rotação e a alteração uniforme da escala (figura 23a). A segunda etapa é dividida em dois processos que juntos resolvem o problema da escala. O primeiro processo encaixa cada triângulo da malha original em seu triângulo intermediário correspondente sem alterar a escala (figura 23b). O segundo processo calcula o resultado final minimizando o erro entre os triângulos do processo anterior e os triângulos do resultado final (figura 23c).

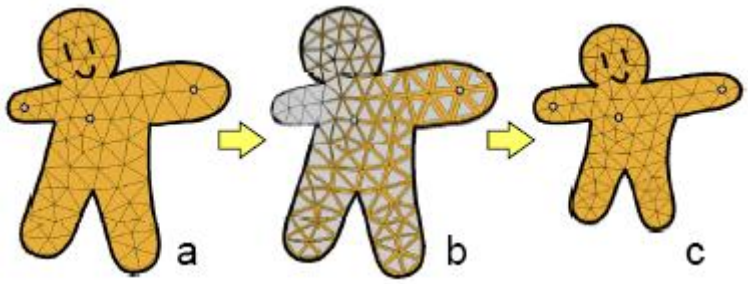


Figura 23 Resumo do algoritmo. Imagem obtida através do trabalho do Igarashi e colegas [1].

3.5.1- Manipulação sem escala

Esta etapa do algoritmo tem por objetivo resolver os problemas de rotação e translação durante a manipulação das imagens, não levando em consideração as deformações de escala.

Inicialmente partimos do princípio de que tendo os dois vértices de um triângulo e as projeções do terceiro vértice, podemos encontrar suas coordenadas através da equação abaixo:

$$v_2 = v_0 + x_{01} \overrightarrow{v_0 v_1} + y_{01} R_{90} \overrightarrow{v_0 v_1}$$

Em que $\boldsymbol{x_{01}}$ e $\boldsymbol{y_{01}}$ são as projeções do vértice $\boldsymbol{v_2}$no vetor $\boldsymbol{v_0 v_1}$e $\boldsymbol{R_{90}}$é a matriz anti-horária de rotação de 90 graus. Abaixo segue um triângulo ilustrativo:

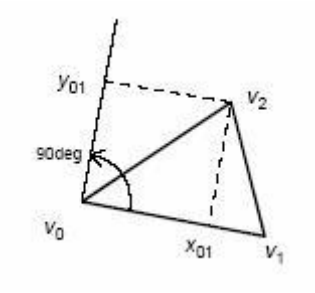


Figura 24 Triângulo exemplo.

Assim, manipulando-se os vértices $\boldsymbol{v_0}$ e v_1 para novas posições ($\boldsymbol{v_0}$' e $\boldsymbol{v_1}$') e tendo as projeções (x_{01}e y_{01}) das posições iniciais de $\boldsymbol{v_2}$, podemos encontrar a posição desejada para o vértice livre (nesse caso seria $\boldsymbol{v_2}$) através da fórmula:

$$\boldsymbol{v_2^d = v_0' + x_{01}\overrightarrow{v_0'v_1'} + y_{01}R_{90}\overrightarrow{v_0'v_1'}}$$

Contudo, nem sempre a posição para onde o vértice realmente foi ($\boldsymbol{v_2}$') é a posição desejada para o mesmo e, portanto, teremos um erro associado a este vértice que pode ser representado da seguinte forma:

$$\boldsymbol{E_{v_2} = ||v_2^d - v_2'||^2}$$

Nós podemos definir v_0^de v_1^dsimilarmente, e representar o erro associado a todo o triângulo como:

$$E_{v_0,v_1,v_2} = \sum_{i=0,1,2}^{i} \boldsymbol{||v_i^d - v_i'||^2}$$

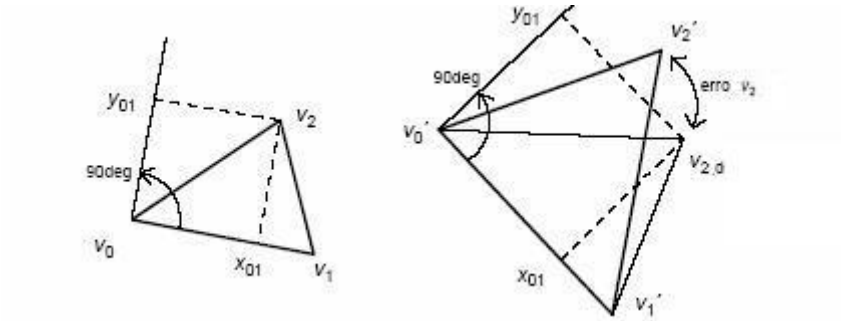


Figura 25 Métrica de erro de v_2.

Como a equação do erro é quadrática em v'=($\boldsymbol{v_{0x}}$', $\boldsymbol{v_{0y}}$', $\boldsymbol{v_{1x}}$', $\boldsymbol{v_{1y}}$', $\boldsymbol{v_{2x}}$', $\boldsymbol{v_{2y}}$') podemos expressá-la na forma matricial como:

$$\boldsymbol{E_{v'} = v'^{T} G_{triângulo} v'}$$

Portanto, cada triângulo da malha terá uma matriz $\boldsymbol{G_{triângulo}}$associada a ele. Sendo o erro de toda a malha o somatório dos erros de todos os triângulos, precisamos encontrar a matriz G que representa a malha através da concatenação das matrizes $\boldsymbol{G_{triângulo}}$de cada triângulo que a compõe. Essa concatenação se dá da seguinte forma: supomos os triângulos *T1* e *T2* na figura 26 e suas respectivas matrizes G (GT1 e GT2), devemos somar os valores das duas matrizes, obedecendo o critério de que cada linha da matriz representa uma coordenada de algum dos pontos, como por exemplo, a posição 1,1 das matrizes GT1, GT2 e G representa x_0, portanto somamos os valores que estão nesta posição em GT1 e GT2 e colocamos em G. De outra forma ocorre, por exemplo, quando queremos preencher a posição 8,7 da matriz G. Esta posição representa a linha y_3 com a coluna de x_3, pois devemos simplesmente copiar o valor que está na posição 6,5 de GT2, visto que nenhuma posição da matriz GT1 representa x_3 e y_3, uma vez que o vértice 3 não pertence ao triângulo T1, como podemos observar na figura 26:

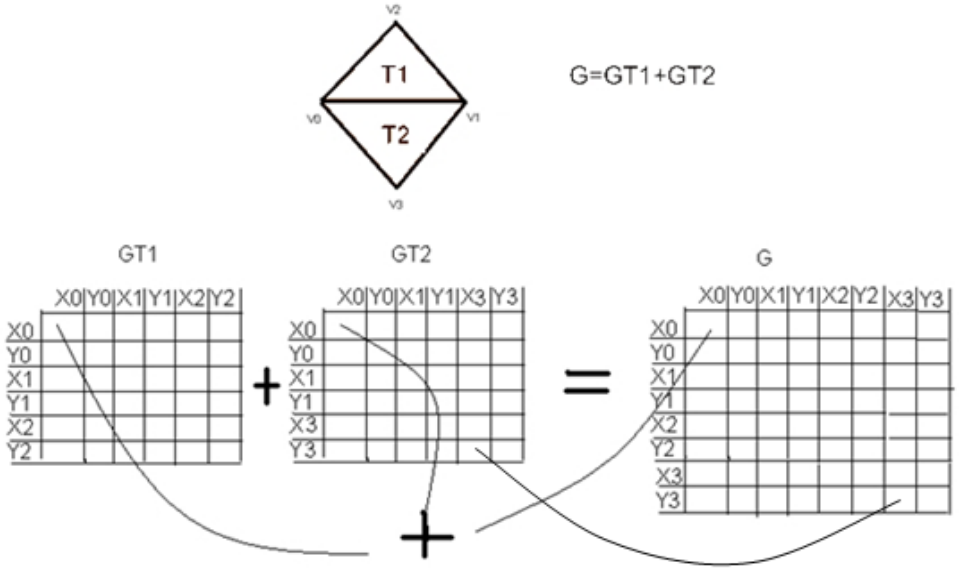
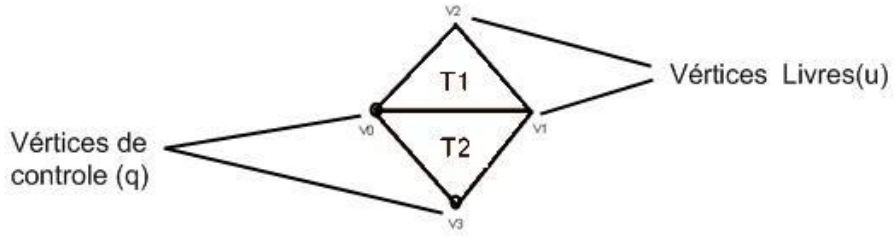


Figura 26 Concatenação das matrizes Gtriângulo.

Vale reforçar que cada elemento da matriz G faz referência a uma coordenada de algum vértice da malha (x ou y de algum vértice de acordo com a concatenação feita anteriormente) e, portanto reorganizamos a matriz em vértices livre e de controle de forma que as primeiras linhas e colunas representem os vértices livres (u) e as ultimas representem os vértices de controle (q), conforme ilustrado na figura 27:



	X0	Y0	X1	Y1	X2	Y2	X3	Y3
X0								
Y0								
X1								
Y1								
X2								
Y2								
X3								
Y3								

⇒

	X0	Y0	X3	Y3	X1	Y1	X2	Y2
X0								
Y0								
X3								
Y3								
X1								
Y1								
X2								
Y2								

Figura 27 Exemplo de reorganização da matriz G.

Assim, a equação do erro pode ser representada da seguinte forma:

$$E_1 = \mathbf{v'}^{\mathrm{T}}\mathbf{G}\mathbf{v'} = \begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix}^{\mathrm{T}}\begin{bmatrix}\mathbf{G}_{00} & \mathbf{G}_{01}\\ \mathbf{G}_{10} & \mathbf{G}_{11}\end{bmatrix}\begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix}$$

Como precisamos minimizar o erro, fazemos a derivada parcial da função $\boldsymbol{E_{v'}}$ em que $\mathbf{u} = (\boldsymbol{u_{0x}}, \boldsymbol{u_{0y}}, \ldots, \boldsymbol{u_{mx}}, \boldsymbol{u_{my}})^{\boldsymbol{T}}$ são os vértices livres, $\mathbf{q} = (\boldsymbol{q_{0x}}, \boldsymbol{q_{0y}}, \ldots, \boldsymbol{q_{nx}}, \boldsymbol{q_{ny}})^{\boldsymbol{T}}$ são os vértices de controle e $\boldsymbol{v'^{T}} = (\boldsymbol{u^{T}q^{T}})$ o conjunto de todos os vértices, conforme a equação abaixo:

$$\frac{\partial E_1}{\partial \mathbf{u}} = (\mathbf{G}_{00} + \mathbf{G}_{00}{}^{\mathrm{T}})\mathbf{u} + (\mathbf{G}_{01} + \mathbf{G}_{10}{}^{\mathrm{T}})\mathbf{q} = \mathbf{0}$$

Indo mais além, podemos simplificar a fórmula do erro, representada na equação acima, da seguinte forma:

$$\boldsymbol{G'u + Bq = 0}$$

Isolando-se a matriz u que é a matriz com os valores que desejamos encontrar, temos:

$$\boldsymbol{u = -(G')^{-1}Bq}$$

Vale ressaltar que u é uma matriz m x 2, G' é uma matriz m x m, B é uma matriz m x n e q é uma matriz n x 2 em que m é o número de vértices livres e n o numero de vértices de controle. Como o resultado de $-G'^{-1}B$ pode ser pré-calculado (assim que selecionados os vértices de controle), os cálculos da manipulação podem ser obtidos de uma forma rápida, pois depende apenas de uma simples multiplicação de matrizes já conhecidas.

3.5.2- Ajuste de escala

Nesta etapa utilizamos o resultado da etapa anterior (coordenadas x e y de todos os vértices) e encontramos o resultado final (coordenadas x e y de todos os vértices livres) ajustando a escala de todos os triângulos da malha. Esta etapa é dividida em duas sub-etapas que serão explicadas a seguir.

3.5.2.1- Transformando o triângulo original em um triângulo intermediário

Tendo um triângulo $\{v_0', v_1', v_2'\}$ obtido através da etapa anterior (chamamos de triângulo linha) e o seu triângulo original correspondente $\{v_0, v_1, v_2\}$, o primeiro problema é encontrar um novo triângulo $\{v_0^f, v_1^f, v_2^f\}$ (chamamos de triângulo *fitted*) que é congruente a $\{v_0, v_1, v_2\}$ e minimiza a seguinte função:

$$E_{\{v_0^f, v_1^f, v_2^f\}} = \sum_{i=0,1,2}^{i} ||v_i^f - v_i'||^2$$

Como é difícil obter o resultado diretamente, nós o aproximamos fazendo primeiramente a minimização do erro através de uma escala uniforme e posteriormente ajustando a escala (figura 28).

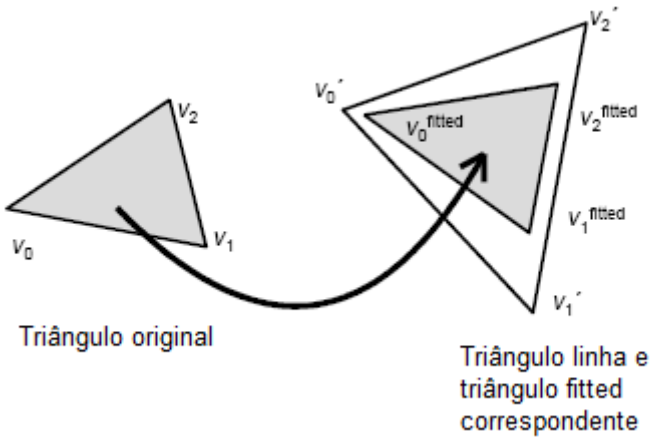


Figura 28 Transformando o triângulo original em um triângulo intermediário (fitted) através da translação e rotação.

Portanto, como o triângulo *fitted* é congruente ao triângulo original, podemos expressar o valor de v_2^f em função dos valores de v_0^f e v_1^f utilizando os valores das projeções de v_2 no triângulo original ($x_{01}\ e\ y_{01}$):

$$\boldsymbol{v_2^f = v_0^f + x_{01}\overrightarrow{v_0^f v_1^f} + y_{01} R_{90}\overrightarrow{v_0^f v_1^f}}$$

Assim, o problema se resumiria em encontrar v_0^f e v_1^f visto que o outro vértice está definido em função deles. Para isso, nós minimizamos o erro dos vértices v_0^f e v_1^f igualando a zero a derivada parcial do erro em função desses vértices (em que $w = (v_{0x}^f, v_{0y}^f, v_{1x}^f, v_{1y}^f)^T$):

$$\frac{\partial E_f}{\partial w} = Fw + C = 0$$

Portanto, ao fazermos a derivada parcial do erro em função de w, encontramos a matriz F (uma matriz 4x4) e a matriz C (uma matriz 4x1). Para encontrar w basta fazermos a multiplicação das matrizes $-F^{-1}C$ e assim teremos os valores dos vértices v_0^f e v_1^f e conseqüentemente podemos encontrar o valor do vértice v_2^f utilizando a fórmula descrita anteriormente. Tendo agora o triângulo $\{v_0^f, v_1^f, v_2^f\}$ que é similar ao triângulo original $\{v_0, v_1, v_2\}$ podemos fazê-lo congruente ao original aplicando um fator de escala igual a $\frac{||v_0^f - v_1^f||}{||v_0 - v_1||}$.

Vale ressaltar que F é fixo para uma determinada malha, a matriz C é definida pelos resultados obtidos na primeira etapa e para que seja aplicada a escala descrita anteriormente devemos primeiramente transladar o baricentro do triângulo até a origem para que depois seja aplicado o fator de escala. O que foi descrito nesta etapa é feito para cada um dos triângulos da malha.

Abaixo segue a composição das matrizes F e C:

Matriz F =

$$\begin{matrix} 4-4x_{01}+2x_{01}^2+2y_{01}^2 & 0 & -2x_{01}^2+2x_{01}-2y_{01}^2 & -2y_{01} \\ 0 & 4-4x_{01}+2x_{01}^2+2y_{01}^2 & 2y_{01} & -2x_{01}^2+2x_{01}-2y_{01}^2 \\ -2x_{01}^2+2x_{01}-2y_{01}^2 & 2y_{01} & 2+2x_{01}^2+2y_{01}^2 & 0 \\ -2y_{01} & -2x_{01}^2+2x_{01}-2y_{01}^2 & 0 & 2+2x_{01}^2+2y_{01}^2 \end{matrix}$$

Matriz C =

$$\begin{matrix} -2v'_{0x}-2v'_{2x}+2v'_{2x}x_{01}+2v'_{2y}y_{01} \\ -2v'_{0y}-2v'_{2y}-2v'_{2x}y_{01}+2v'_{2y}x_{01} \\ -2v'_{1x}-2v'_{2x}x_{01}-2v'_{2y}y_{01} \\ -2v'_{1y}+2v'_{2x}y_{01}-2v'_{2y}x_{01} \end{matrix}$$

3.5.2.2- Gerando o resultado final utilizando os triângulos fitted

Nesta etapa é calculado o resultado final das coordenadas x e y dos vértices livres da malha através do resultado obtido na etapa anterior (os triângulos *fitted*). Ela consiste na minimização do erro do resultado final do triângulo (iremos chamar de triângulo duas linhas) em relação ao triângulo *fitted* correspondente. Como anteriormente a explicação será feita em relação a um triângulo simples $\{v_0, v_1, v_2\}$ que tem seu correspondente *fitted* como $\{v_0^f, v_1^f, v_2^f\}$ e então podemos definir o erro deste triângulo como sendo:

$$E_{2\{v_0''v_1''v_2''\}} = \sum_{(i,j)\in\{(0,1),(1,2),(2,0)\}} \left\| \overrightarrow{v_i''v_j''} - \overrightarrow{v_i^{\text{fitted}} v_j^{\text{fitted}}} \right\|^2$$

Vale observar que o erro neste caso está associado às arestas e não aos vértices como nas etapas anteriores (figura 29). O erro do triângulo é minimizado (igual a zero) quando o triângulo *fitted* é idêntico ao triângulo duas linhas. Contudo, como resultado da etapa anterior, um vértice pode ter mais de um valor *fitted* para ele, dependendo da quantidade de triângulos a que ele pertence, portanto cabe a esta etapa encontrar a melhor posição para o vértice de forma que o erro seja minimizado.

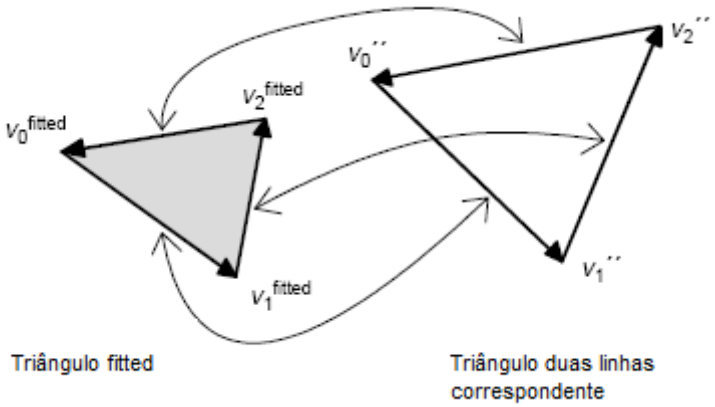


Figura 29 A métrica do erro entre o triângulo *fitted* e o triângulo duas linhas que se da através da métrica das arestas.

Como na primeira etapa, o erro do triângulo pode ser representado de uma forma matricial da seguinte maneira:

$$E_{\{v''\}} = v''^{T} H v'' + f v'' + c$$

Então, da mesma forma que na primeira etapa, cada triângulo terá uma matriz H, f e c que serão concatenadas entre as matrizes de todos os triângulos da malha de forma que se obtenha as matrizes H, f e c de toda a malha (do mesmo modo como fizemos com a matriz G da primeira etapa). Note que H é definida pela conectividade da malha original e é independente dos triângulos *fitted*, enquanto f e c são determinadas pelos triângulos *fitted* e, portanto mudam durante a manipulação. Também, como na primeira etapa, um elemento da matriz H e f faz referência a uma coordenada de algum vértice da

malha (x ou y de algum vértice de acordo com a concatenação feita anteriormente) e, portanto, precisamos reorganizar as matrizes em vértices livre e de controle de forma que as primeiras linhas e colunas representem os vértices livres (u) e as ultimas representem os vértices de controle (q):

$$E_2 = \mathbf{v''}^{\mathrm{T}}\mathbf{H}\mathbf{v''} + \mathbf{f}\mathbf{v''} + c = \begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix}^{\mathrm{T}}\begin{bmatrix}\mathbf{H}_{00} & \mathbf{H}_{01}\\ \mathbf{H}_{10} & \mathbf{H}_{11}\end{bmatrix}\begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix} + \begin{pmatrix}\mathbf{f}_0\mathbf{f}_1\end{pmatrix}\begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix} + c$$

Como precisamos minimizar o erro, fazemos a derivada parcial da função $\boldsymbol{E_v}$, em que $\mathbf{u} = (\boldsymbol{u_{0x}}, \boldsymbol{u_{0y}}, \ldots, \boldsymbol{u_{mx}}, \boldsymbol{u_{my}})^{\boldsymbol{T}}$são os vértices livres,$\mathbf{q} = (\boldsymbol{q_{0x}}, \boldsymbol{q_{0y}}, \ldots, \boldsymbol{q_{nx}}, \boldsymbol{q_{ny}})^{\boldsymbol{T}}$ são os vértices de controle e $\boldsymbol{v''^{T}} = (\boldsymbol{u^T q^T})$ o conjunto de todos os vértices:

$$\frac{\partial E_2}{\partial \mathbf{u}} = (\mathbf{H}_{00} + \mathbf{H}_{00}{}^{\mathrm{T}})\mathbf{u} + (\mathbf{H}_{01} + \mathbf{H}_{10}{}^{\mathrm{T}})\mathbf{q} + \mathbf{f}_0 = \mathbf{0}$$

Podemos re-escrever a equação acima da seguinte forma:

$$\mathbf{H'u} + \mathbf{Dq} + \mathbf{f}_0 = \mathbf{0}$$

Ou, isolando u teremos:

$$\boldsymbol{u} = -\boldsymbol{H'^{-1}}(\boldsymbol{Dq} + \boldsymbol{f_0})$$

Vale salientar que H’ e D são fixos, mas q e f_0 variam durante a manipulação. Abaixo teremos exemplos de uma matriz H e de uma matriz f de um triângulo:

$$\begin{matrix} 2 & 0 & -1 & 0 & -1 & 0 \\ 0 & 2 & 0 & -1 & 0 & -1 \\ -1 & 0 & 2 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ -1 & 0 & -1 & 0 & 2 & 0 \\ 0 & -1 & 0 & -1 & 0 & 2 \end{matrix}$$

Matriz H de um triângulo.

$$\begin{matrix} 2v_{1x}^{f} - 4v_{0x}^{f} + 2v_{2x}^{f} \\ 2v_{1y}^{f} - 4v_{0y}^{f} + 2v_{2y}^{f} \\ 2v_{0x}^{f} - 4v_{1x}^{f} + 2v_{2x}^{f} \\ 2v_{0y}^{f} - 4v_{1y}^{f} + 2v_{2y}^{f} \\ 2v_{1x}^{f} - 4v_{2x}^{f} + 2v_{0x}^{f} \\ 2v_{1y}^{f} - 4v_{2y}^{f} + 2v_{0y}^{f} \end{matrix}$$

Matriz f de um triângulo

3.5.3- Sumário do Algoritmo

O algoritmo apresentado nas seções anteriores pode ser resumido da seguinte forma:

1- Registro (quando uma nova figura é carregada)

1.1- Construa as matrizes G e H da malha.

1.2- Construa F e inverta ela para cada triângulo.

2- Compilação (quando são adicionados novos vértices de controle)

2.1- Construa G' e B através de G e compute $G'^{-1}B$.

2.2- Construa H' e D através de H.

3- Durante a manipulação.

3.1- Obtenha as coordenadas linha dos vértices livres através da operação $-G'^{-1}Bq$ onde q representa os vértices de controle.

3.2- Construa C de cada triângulo utilizando os valores linha encontrados. Multiplique $-F^{-1}C$ e ajuste a escala para obter os triângulos *fitted*.

3.3- Construa f_0 usando os triângulos *fitted* e obtenha o resultado final.

3.6- Inpainting

Existem casos em que a manipulação dos objetos gera “furos” na imagem que precisam ser preenchidos a fim de manter a coerência da mesma (figura 30). A técnica de *inpainting* foi justamente utilizada para preencher esses “furos” de forma que a imagem apresente uma aparência agradável após a manipulação. Ressaltamos que a técnica original não se preocupava com este aspecto do problema.

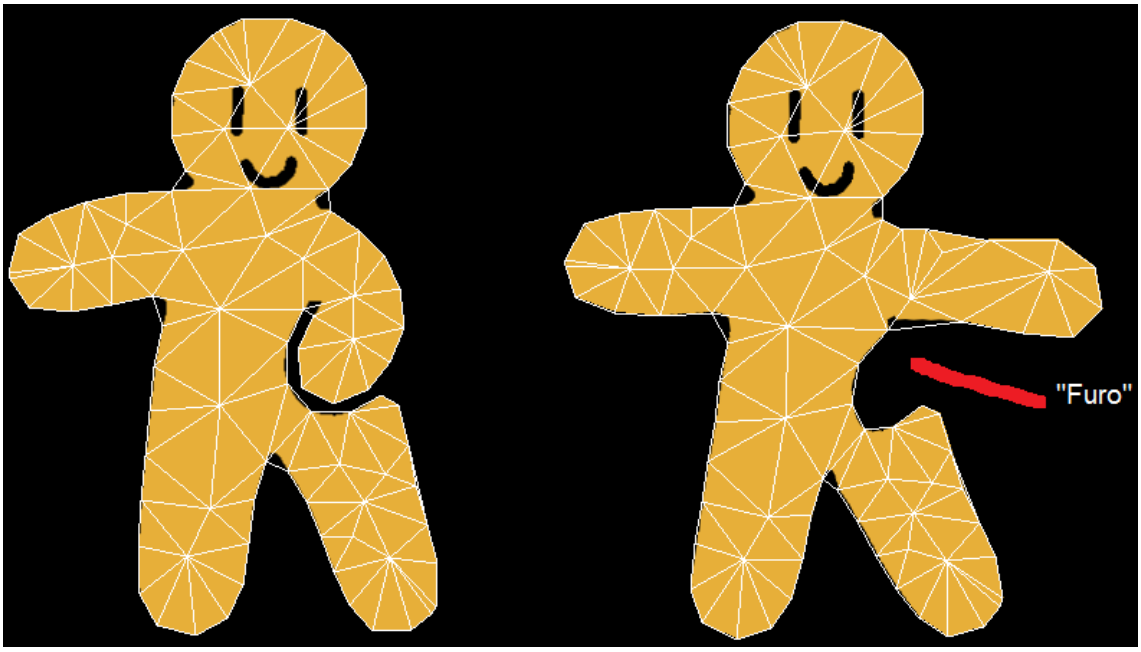


Figura 30 Exemplo de manipulação que gera “furo” na imagem.

O algoritmo de *inpainting* utilizado consiste em gerar um polígono com o formato do “furo” para que o usuário interativamente mova-o pela imagem a fim de escolher o melhor “pedaço” da imagem que preencha o “furo” (figura 31). Apesar de simples, esta solução é utilizada em alguns trabalhos relacionados, como por exemplo, no trabalho do Zeev Farbman e colegas [13]. Em tempo-real o usuário avalia se a região sendo copiada para tapar o furo satisfaz ou não.

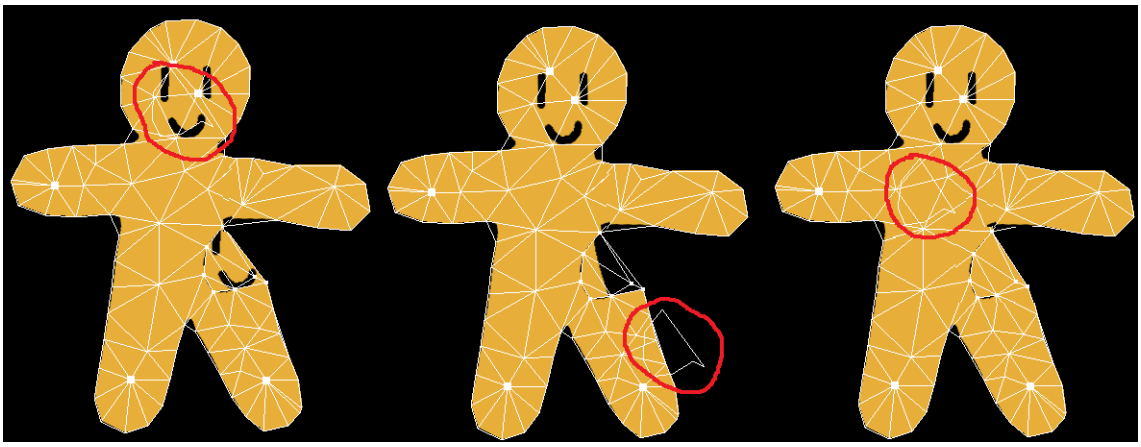


Figura 31 Exemplo da execução do inpainting. As regiões em vermelho ilustram onde o usuário busca por um "remendo" para o furo. Ao encontrar uma região satisfatória, o usuário informa ao sistema que copia, então, a informação visual para a área do furo.

Vale ressaltar que essa abordagem não resolve todos os problemas, contudo para os casos mais simples, em que os objetos não possuem um número muito grande de detalhes, ela apresentou resultados satisfatórios, inclusive para objetos com textura, como iremos apresentar nos resultados.

4- Avaliação e Resultados

Neste capítulo serão apresentados os resultados e suas respectivas avaliações. Dividimos os resultados em dois grupos principais: sem e com *inpainting*.

4.1- Sem Inpainting

Aqui serão apresentados os resultados obtidos nos casos mais simples, em que a manipulação não gera “furos” na imagem (esses “furos” ocorrem geralmente quando uma parte do objeto está se sobrepondo a outra e, portanto, quando essa parte é movida, há o surgimento de um espaço vazio no local).

Na figura 32, representamos algumas manipulações que podem ser feitas na imagem do boneco, como por exemplo, fazer ele dar “tchau” ou “abrir escala”. Na figura 33, foram feitas diversas manipulações com o famoso personagem de *Monstros SA*, tais como movimentação de seus braços e pernas, assim como o aumento e diminuição dos mesmos. Na figura 34, representamos um leão movimentando a sua cauda. Na figura 35, fizemos o cachorro movimentar a sua cabeça. E para finalizar, na figura 36, fizemos o nariz do *pinnochio* esticar, para representar ele contando uma mentira.

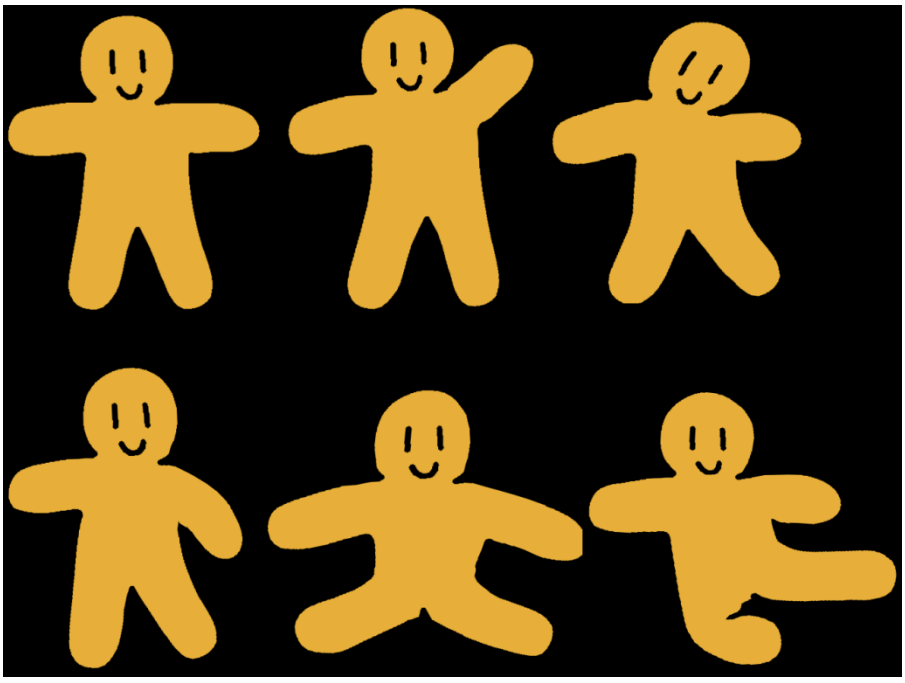


Figura 32 Manipulação do boneco.

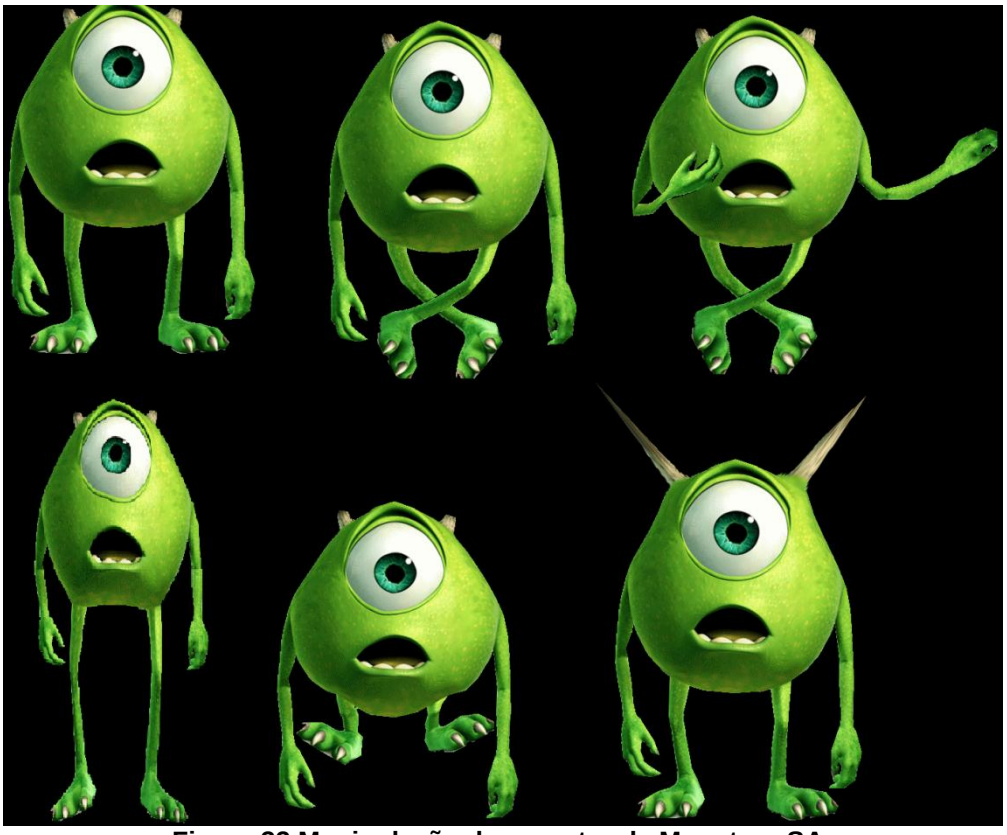


Figura 33 Manipulação do monstro de Monstros SA.

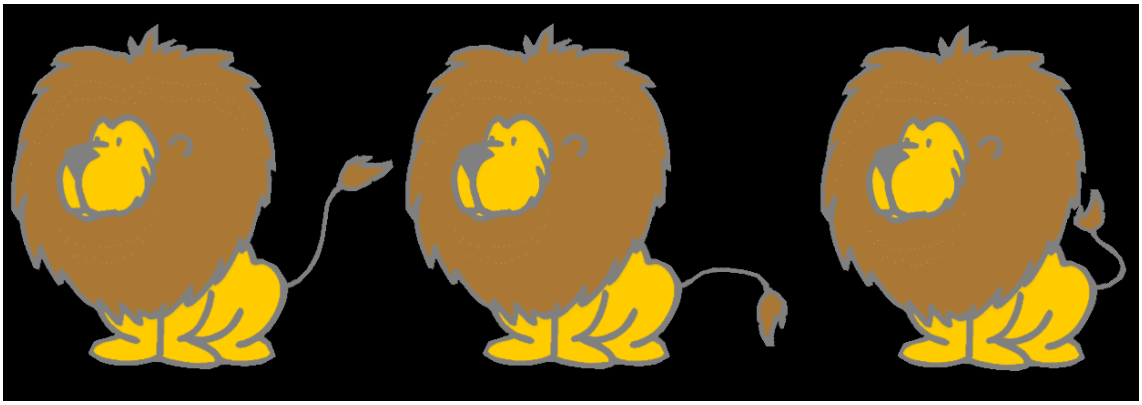


Figura 34 Manipulação da cauda do leão.

Figura 35 Manipulação do cachorro.

Figura 36 Manipulação simulando o *pinnochio* contando uma mentira.

Como podemos observar, nos casos em que nenhuma parte do objeto se sobrepõe a outra parte do mesmo objeto, os resultados foram muito satisfatórios. Através de uma imagem original conseguimos manipulá-la interativamente obtendo diversas outras imagens que mantêm as propriedades da imagem original. Assim, conseguimos gerar diversas imagens que simulam movimentos de objetos com apenas uma única imagem. Como por exemplo, na figura acima do leão, com apenas a imagem original conseguimos obter imagens que simulam a movimentação de sua cauda, assim, facilmente conseguimos criar uma animação do leão com apenas uma imagem.

4.2- Com Inpainting

Aqui serão apresentados os resultados obtidos nos casos mais complexos em que se necessita utilizar a técnica de *inpainting* (nas imagens em que parte do objeto está se sobrepondo a outra, portanto a manipulação do objeto pode gerar “furos”). Dividimos estes resultados em três casos que serão analisados a seguir.

4.2.1- Objetos sem textura

Neste grupo estão os objetos que apesar de apresentarem partes sobrepostas, são objetos bem simples, sem textura e sem muitos detalhes nas partes que foram sobrepostas (podemos observar um exemplo na figura 37).

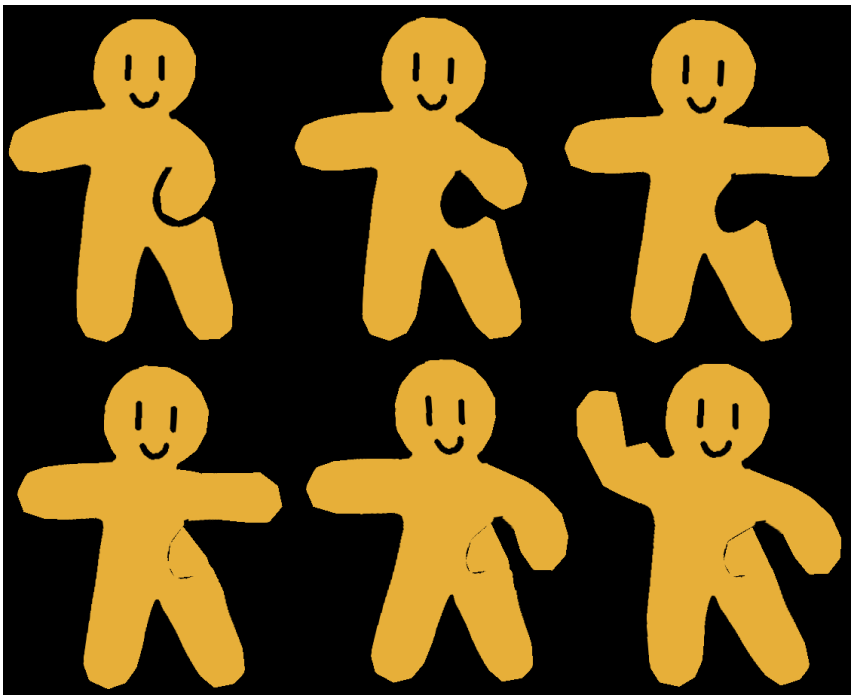


Figura 37 Exemplo de manipulação de objeto sem textura.

Os resultados obtidos nestes casos foram satisfatórios, pois, apesar de poder haver ainda pequenas falhar no processo de pintura dos “furos”, elas podem ser facilmente corrigidas com um simples editor de imagens sem necessitar de muita mão-de-obra nem de muito conhecimento em edição de imagens. Os pequenos detalhes pretos na área onde o “furo” foi preenchido, são decorrentes do mapeamento da malha em relação a imagem original, pois como a malha é composta de triângulos, não se consegue mapear perfeitamente uma curva com um conjunto de retas (as arestas dos triângulos) e portanto, alguns pequenos “pedaços” fora da objeto são mapeados, como podemos observar na figura 38 abaixo:

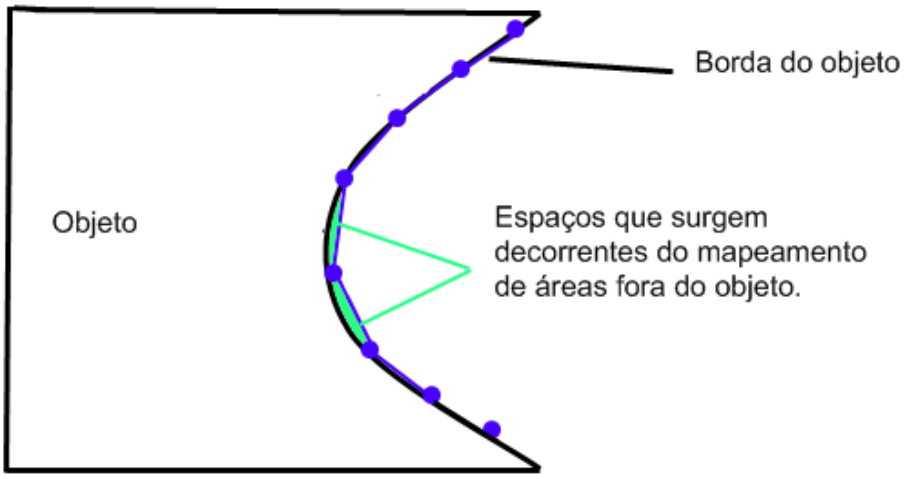


Figura 38 Erros decorrentes do mapeamento da textura.

4.2.2- Objetos sem textura, mas que apresentam detalhes sobrepostos

Nestes casos estão enquadrados aqueles objetos que, apesar de não apresentarem textura, possuem em suas partes sobrepostas um certo nível de detalhes. A figura 39 abaixo ilustra um destes casos. Aqui podemos observar que a região do olho estava sobreposta pelo braço, portanto a imagem não possuía informações relativas a este olho. Ao movimentarmos o braço esquerdo, um “furo” foi gerado justamente onde se encontrava este olho, que pôde ser reconstituído graças às informações fornecidas pelo olho direito (que estava completo). Vale observar que após o preenchimento do espaço vazio, a manipulação pode continuar normalmente, como podemos ver na figura abaixo:

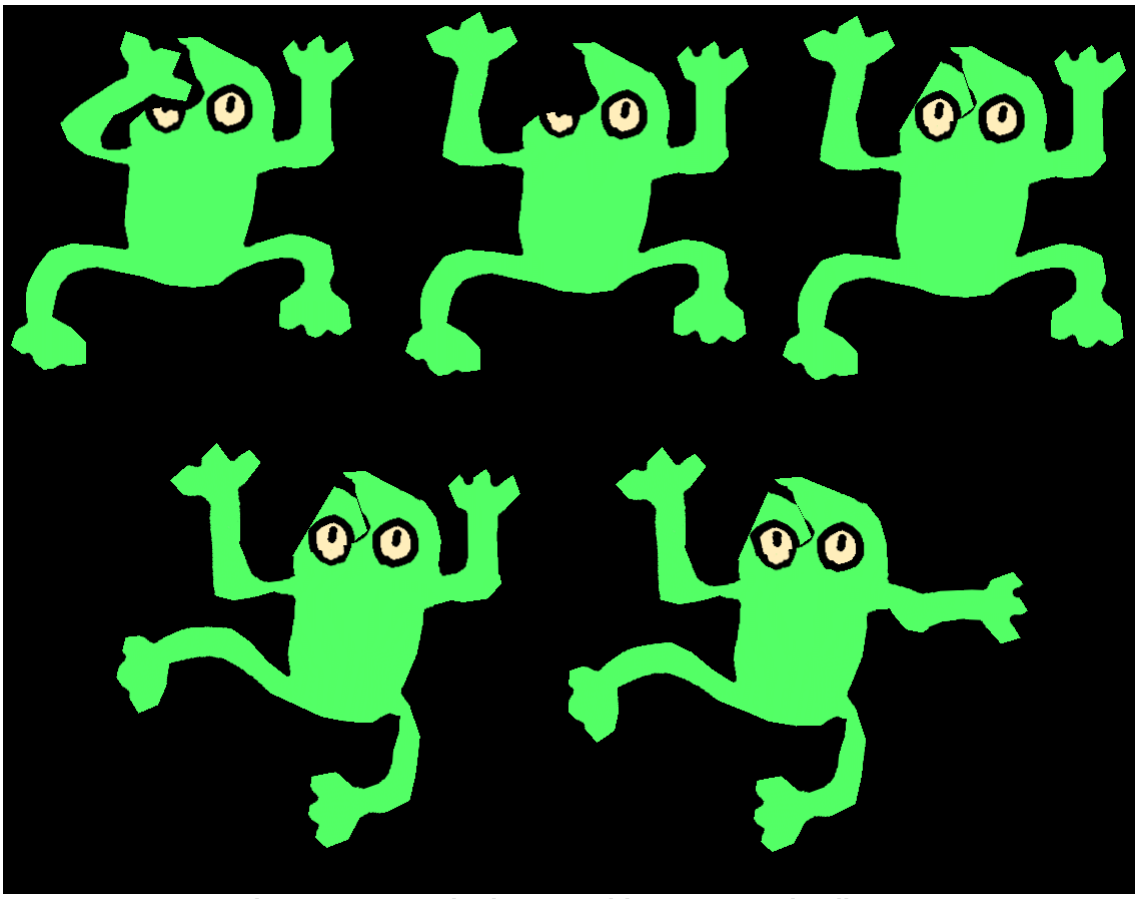


Figura 39 Exemplo de preenchimento com detalhes.

Consideramos os resultados obtidos nestes casos como bons, pois apesar de não corrigirem perfeitamente os “furos” deixados pela manipulação do objeto, a parte principal do “furo” foi recuperada. Como por exemplo, na imagem acima, ao mover o braço esquerdo do sapo deixamos um buraco justamente onde fica o olho esquerdo, contudo, conseguimos restaurar este olho que era a parte mais importante. Assim, as partes menos importantes, que não puderam ser recuperadas, podem ser facilmente criadas com um simples editor de imagens, agilizando o processo de criação de animações.

4.2.2- Objetos com textura

Nestes casos estão enquadrados aqueles objetos, que diferentemente dos grupos anteriores, possuem textura. Assim, o processo de recuperação da imagem precisa ser mais cuidadoso, pois deve-se encontrar o melhor "pedaço" da textura que preencha o "furo" de forma que este fique o melhor possível imperceptível ao usuário (figura 40), ou seja, que esta parte preenchida fique como se fizesse parte da imagem original. Como podemos observar na figura 41, o urso foi manipulado de forma que gerou um "furo" em sua estrutura que foi preenchido.

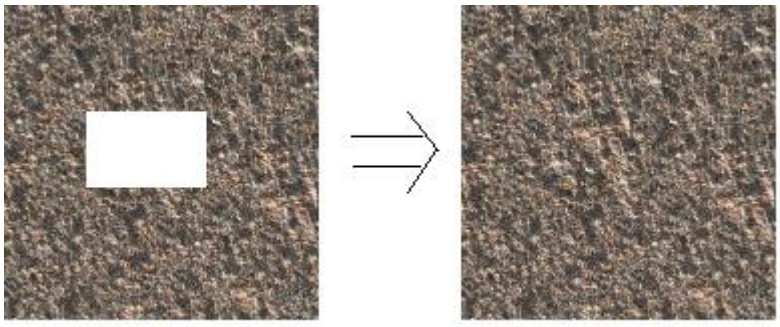


Figura 40 Ilustração de um preenchimento de textura.

Figura 41 Exemplo de um preenchimento de objetos com textura.

Os resultados obtidos nestes casos também foram bons, pois apesar de não corrigir perfeitamente os "furos" deixados pela manipulação do objeto (em todos os 3 casos a borda do local a ser preenchido continua preta) tem-se um resultado visualmente agradável. Para se ter um preenchimento praticamente

perfeito, deve-se adotar uma abordagem *pixel* a *pixel* de *inpainting* que garantiria um maior controle sobre a estrutura da imagem, podendo-se fazer um preenchimento mais detalhado, visto que, com o preenchimento *pixel* a *pixel* poder-se-ia combinar várias partes da imagem para preencher o “furo” ao invés de utilizar dados de uma única região.

5- Considerações Finais

Pode-se dizer que os objetivos traçados para este trabalho foram atingidos satisfatoriamente. Foi proposto e implementado um sistema que simule a manipulação interativa de objetos em imagens como se o objeto fosse um objeto real, assim sendo, características como a rigidez devem ser preservadas durante a manipulação.

Outro propósito do trabalho, que também obteve resultados satisfatórios, foi desenvolver uma solução para os casos dos objetos que possuem partes oclusas, ou seja, aqueles objetos em que suas partes se sobrepõem.

Entretanto, existem vários espaços para melhoras e aperfeiçoamentos. A técnica de *inpainting* adotada*,* por exemplo, apesar de obter bons resultados em uma boa quantidade de casos, não mostrou resolver perfeitamente o problema (como podemos observar nas figuras 37 e 41), pois as bordas da área a ser preenchida permanecem ainda com pequenos “furos”. Além disso, muito dos casos em que a área do objeto a ser preenchida apresentar muitos detalhes, a técnica se mostrou ineficiente.

Para resolver esses casos, seria necessário partir para uma abordagem de preenchimento *pixel* a *pixel*. Assim ter-se-ia um controle maior sobre a estrutura da imagem, além de poder fazer um preenchimento mais detalhado, visto que seria feito o preenchimento *pixel* a *pixel* ao invés de toda a área de uma única vez. Assim, ao invés de escolher uma única parte da imagem para preencher o “furo”, seria possível juntar diversas partes da imagem para preenchê-lo, melhorando a qualidade do *inpainting.* Contudo, deve-se observar que uma abordagem *pixel* a *pixel* seria mais custosa computacionalmente e portanto, é necessário analisar o impacto desta decisão na eficiência, para evitar que a perda de eficiência prejudique a interação com a ferramenta e conseqüentemente a manipulação em tempo real do objeto.

A técnica de manipulação aqui proposta apresentou resultados muito satisfatórios, apesar disso, podemos destacar uma grande melhoria a ser feita, como por exemplo, estender a técnica para modelos 3D, aumentando assim o domínio da aplicação. Com isso, poderíamos ter uma manipulação perfeita, pois não estaríamos manipulando uma imagem 2D (que não apresenta todos os detalhes limitando a manipulação, como por exemplo, se tivermos um

boneco virado de frente, não podemos manipulá-lo de forma que ele fique de costas, pois a imagem não possui as informações necessárias sobre as costas do boneco) e sim um objeto 3D que possui todos os detalhes de todos os ângulos de visão.

Com uma abordagem 3D, a ferramenta desenvolvida poderia auxiliar não só na criação e edição de imagens 2D, mas também na manipulação de estruturas 3D, sendo de grande utilidade para a indústria de jogos 3D, pois seria possível simular e testar as movimentações realizadas pelos objetos na própria ferramenta, antes mesmo de serem introduzidas nos jogos.

Referências

[1] T. Igarashi, T. Moscovich, and J. F. Hughes. As-rigid-as-possible shape manipulation. ACM Transactions on Graphics, 24(3):1134–1141, Aug. 2005.

[2] Devil a Full featured cross-platform image library. Disponível em: http://openil.sourceforge.net/about.php.

[3] William E. Lorensen, Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. Computer Graphics (Proceedings of SIGGRAPH 87), July 1987, pp. 163-169.

[4] Mark de Berg et al. Computational Geometry: Algorithms and Applications. Springer, 3rd edition

[5] GLUI User Interface Library. Disponível em: http://www.cs.unc.edu/~rademach/glui/.

[6] Weng. Y et al. 2D Shape Deformation Using Nonlinear Least Squares Optimization. Proceedings of Pacific Graphics 2006, 2006

[7] S. Schaefer, T. McPhail, and J. Warren. Image deformation using moving least squares. ACM Trans. Graph., 25(3):533. 540, 2006.

[8] Cuno A. et al. 3D As-Rigid-As Possible Shape Deformations Using MLS. Proceedings of Computer Graphics International 2007.

[9] Decoret X. et al. Multi-layered impostors for accelerated rendering. Computer Graphics Forum, 18(3), September 1999, pp. 61-73.

[10] Astle, D., Durnil, D. OpenGL ES Game Development. Course Technology PTR, 2004

[11] Alexei A. Efros and Thomas K. Leung. Texture Synthesis by Non-parametric Sampling. IEEE International Conference on Computer Vision (ICCV'99), Corfu,Greece, September 1999

[12] Bertalmio M., Sapiro G., Caselles V., Ballester C.. Image Inpainting. SIGGRAPH 2000, pages 417-424.

[13] Zeev Farbman et al. Coordinates for Instant Image Cloning. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)

[14] F.P. Preparata and M.I. Shamos, *Computational geometry: An introduction*, Texts and Monographs in Computer Science, Springer-Verlag, New York, 1985.

[15] Wikipedia, the free encyclopedia. Marching Squares. Disponível em: http://en.wikipedia.org/wiki/Marching_squares

[16] Rue, Håvard; Held, Leonhard (2005). *Gaussian Markov random fields: theory and applications*. CRC Press. *ISBN 1584884320*.

[17] L.C. Evans, *Partial Differential Equations*, American Mathematical Society, Providence, 1998. *ISBN 0-8218-0772-2*

[18] Wikipedia, the free encyclopedia. Gauss-Newton Algorithm. Disponível em: http://en.wikipedia.org/wiki/Gauss-Newton_algorithm

[19] Qhull. Disponível em: http://www.qhull.org/

[20] Mega Man 2. Disponível em: http://www.nintendo8.com/game/512/mega_man_2/

[21] Sprite. Disponível em: http://blog.hboaventura.com/tag/sprite/

Assinaturas

Marcelo Walter

Orientador

Diego Lemos de Almeida Melo

Aluno