UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ELETRÔNICA E COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GERAÇÃO DE TRIMAPS EM TEMPO REAL UTILIZANDO O KINECT

TRABALHO DE GRADUAÇÃO

Frederico Artur Limberger

Santa Maria, RS, Brasil

2011

GERAÇÃO DE TRIMAPS EM TEMPO REAL UTILIZANDO O KINECT

por

Frederico Artur Limberger

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Eletrônica e Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação.**

Orientador: Prof. Dr. Marcos Cordeiro d'Ornellas

Trabalho de Graduação N. 329

Santa Maria, RS, Brasil

2011

UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ELETRÔNICA E COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

GERAÇÃO DE TRIMAPS EM TEMPO REAL UTILIZANDO O KINECT

elaborado por
Frederico Artur Limberger

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Marcos Cordeiro d'Ornellas
(Presidente/Orientador)

Prof. Dr. Cesar Tadeu Pozzer (UFSM)

Profª. Dra. Lisandra Manzoni Fontoura (UFSM)

Santa Maria, 15 de dezembro de 2011.

“Bom mesmo é ir a luta com determinação, abraçar a vida com paixão, perder com classe e vencer com ousadia... Pois o triunfo pertence a quem se atreve.”

— CHARLES CHAPLIN

AGRADECIMENTOS

Primeiramente, gostaria de agradecer ao meu pai Décio e minha mãe Marta pela oportunidade, apoio e incentivo que deram a mim nesta etapa. Agradeço também a minha irmã Mabel pelo companheirismo e pelo exemplo de pessoa que é.

Este paragrafo de agradecimento é dela, minha namorada Lis, que me apoiou, me escutou e me deixou falar (as vezes) e que sempre esteve ao meu lado durante estes quatro anos. Por tudo que você significa para mim.

Agradeço a todos os amigos, tanto aqueles que puderam me conceder experiências quanto aqueles que cresceram comigo. Agradeço aos amigos que vivenciaram experiências comigo no Programa de Educação Tutorial, pois através deste que conheci pessoas de outros cursos e aprendi o real significado da palavra “parceria”. Gostaria de agradecer também aos que me ajudaram a crescer dentro do programa e da universidade Mega, Cogo, Adriano e Garcia e aos que pude conviver mais tempo, Kotoko e Candia. Agradeço aos amigos do DA que foi DACC e virou DAINF, mas as amizades continuaram as mesmas, Kreutz, Lunardi e Bass. Não posso deixar de falar dos amigos que compartilharam horas de trabalho e diversão no LaCA, Gottin, Victor (Pl) e Guilherme.

Aos caras legais, também um especial agradecimento, pois vivenciamos juntos esses anos de estudos e trabalhos contínuos que só vocês sabem dizer, Grahl, Bernardo, Reis e Cícero.

Agradeço também aos professores que contribuíram para minha formação. Em especial, agradeço ao professor Pozzer pela orientação e coleguismo que me proporcionou durante a graduação. Agradeço também ao Marcos pela orientação e tutoria, esta compartilhada pela professora Andrea e pelo professor Giovani no decorrer de minha estadia no PET.

Enfim agradeço a todos aqueles que estiveram presentes em algum momento de minha caminhada e contribuíram para o meu crescimento.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

GERAÇÃO DE TRIMAPS EM TEMPO REAL UTILIZANDO O KINECT

Autor: Frederico Artur Limberger
Orientador: Prof. Dr. Marcos Cordeiro d'Ornellas
Local e data da defesa: Santa Maria, 15 de Dezembro de 2011.

A segmentação de camadas precisas utilizando *alpha matting* é muito importante para aplicações nas áreas de efeitos visuais em filmes e televisão. O principal problema das abordagens de *matting* existentes é que estas requerem uma suposição de objetos em primeiro plano e regiões de segundo plano para obterem um bom resultado. Este trabalho apresenta a geração automática de *trimaps* em tempo real para especificar regiões de uma imagem a partir informações de profundidade adquiridas através dos sensores do Kinect, possibilitando a utilização de algoritmos de *alpha matting* em ambientes não controlados. Com os resultados desta técnica, imagens e vídeos poderão usufruir desta tecnologia, possibilitando a segmentação de corpos em tempo real, já que não será mais necessária a entrada manual do usuário para a especificação de *trimaps*.

Palavras-chave: Visão Computacional, Trimap, Kinect, Alpha Matting, Tempo Real.

ABSTRACT

Trabalho de Graduação
Undergraduate Program in Computer Science
Universidade Federal de Santa Maria

REAL-TIME TRIMAPS GENERATION WITH KINECT

Author: Frederico Artur Limberger
Advisor: Prof. Dr. Marcos Cordeiro d'Ornellas

The precise segmentation of layers using alpha matting is very important for applications in the areas of visual effects in movies and television. The main problem of existing matting approaches is that they require an assumption of objects in the foreground and background regions to obtain a good result. This work presents the automatic trimaps generation in real-time to specify regions of an image from depth information acquired through Kinect sensors to specify the foreground regions, background regions and unknown regions through trimaps, allowing the use of algorithms for alpha matting in uncontrolled environments. With the results of this technique, images and videos may take advantage of this technology, enabling the segmentation of bodies in real-time, since you no longer need manual user input for specifying trimaps.

.

Keywords: Computer Vision, Trimap, Kinect, Alpha Matting, Real-time.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

SDK	*Software Development Kit*
RGB	*Red Blue Green*
GLSL	*OpenGL Shading Language*
GPU	*Graphics Processing Unit*
OpenTK	*Open Toolkit Library*
RAD	Rapid Application Development
FPS	*Frames per Second*
CMOS	*Complementary metal–oxide–semiconductor*
NUI	*Natural User Interface*

SUMÁRIO

1 INTRODUÇÃO

Recentemente, a demanda por ambientes virtuais com composições de imagens mais realistas aumentou com o avanço das tecnologias digitais nas indústrias de jogos e cinema (KAWAKITA, IIZUKA, *et al.*, 2004). Em um ambiente virtual o jogador ou ator deve ter a sensação de fazer parte do mundo virtual. Muitas vezes ele é capaz de interagir e modificar o ambiente em que está inserido, mas para isso a virtualização do ambiente deve ser a mais real possível para que ele, em nenhum momento, se confunda com o mundo real. Para que isso aconteça, seu corpo deve ser transportado do mundo real para o mundo virtual sem perda de detalhes. Para que este transporte possa ser feito, devem-se utilizar métodos específicos de processamento de imagens para extrair o corpo do personagem de uma forma muito precisa.

Métodos de extração e composição de objetos são operações de edição fundamentais de imagens e vídeos (GASTAL, 2009). Contudo, devido ao processo de discretização envolvido na captura de imagens digitais, *pixels* localizados em regiões limite recebem iluminações de múltiplos elementos da cena. Consequentemente, estes *pixels* mantêm cores misturadas dos elementos que fazem parte desta cena. O processo de estimar as cores originais que preenchem cada *pixel* destas regiões é conhecido como *alpha matting*.

Para que corpos possam ser extraídos do ambiente real para o ambiente virtual, deve-se especificar alguma informação para que os algoritmos de *alpha matting* consigam saber qual objeto se quer extrair.

Um método muito conhecido chamado *Chroma key* (ROE, 1965-1980) utiliza-se de um ambiente controlado para compor duas camadas de uma cena. A cor de fundo, normalmente azul ou verde, capturada pela câmera e detectada pelo algoritmo é substituída por outra cor de fundo de outra imagem.

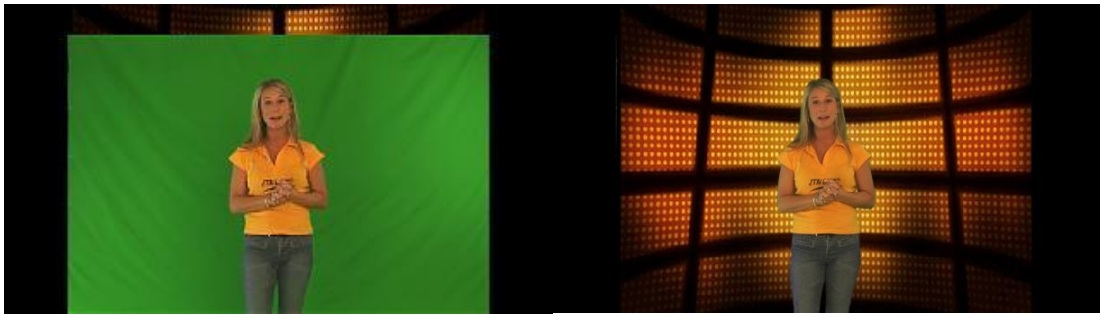

Figura 1 - Técnica *Chroma key* com fundo verde

Esta técnica possui algumas limitações, como por exemplo, o ambiente controlado que deve ser preparado e a limitação do apresentador em não poder vestir cores com os mesmos tons do plano de fundo, como pode ser visto na Figura 1.

Pode-se imaginar que as cores utilizadas no método *Chroma key* funcionam como um *trimap*, especificando o plano de fundo que deverá ser cortado da imagem e substituído. O presente estudo busca substituir este método de especificação de *trimaps* por um método automático utilizando o sensor de profundidade do Kinect.

1.1 Justificativa

Existem hoje inúmeras aplicações que usam algoritmos de *matting* de imagens, desde jogos, softwares de televisão ou quaisquer aplicativos que necessitem algum tipo de reconhecimento corporal ou segmentação de objetos. Este trabalho possibilitará a criação de uma nova classe de softwares que necessitem de *matting* de imagens em tempo real.

Este trabalho justifica-se ao passo da necessidade verificada para a execução de algoritmos de *alpha matting*, onde não é mais preciso a entrada manual do usuário para especificação de regiões de fundo e de frente da imagem.

A técnica que este trabalho propõe é uma nova forma de geração de informações que servirão para auxiliar algoritmos de *alpha matting* na segmentação de imagens em tempo real.

1.2 Objetivos

O objetivo principal deste trabalho é a criação de uma aplicação que gere *trimaps* de corpos humanos em tempo real, possibilitando que estes possam ser utilizados como entrada de dados para algoritmos de *alpha matting*.

Especificamente, o trabalho tem como objetivos:

- Estudar o funcionamento de algoritmos de processamento de imagens, algoritmos de geração de *trimaps* e algoritmos de alpha matting.

- Estudar o funcionamento do Kinect e de seu Pacote de Desenvolvimento de Software (Kinect for Windows SDK).

- Facilitar e deixar mais natural para um usuário, sem a necessidade de um ambiente controlado, a remoção de planos de fundo de cenas, para criar a sensação de imersão do jogador ou telespectador em ambientes virtuais.

- Avaliar os resultados de desempenho obtidos, a partir da execução do aplicativo em diferentes ambientes, a fim de verificar se os algoritmos conseguem executar em tempo real.

1.3 Organização do Texto

Este trabalho está organizado da seguinte maneira: o Capítulo 2 traz uma fundamentação teórica acerca dos assuntos abordados no trabalho, tratando do Kinect, *trimaps*, algoritmos de *alpha matting*, e das linguagens e ferramentas utilizadas. O Capítulo 3 descreve o desenvolvimento do trabalho, indicando os requisitos para a instalação e configuração das ferramentas, sobre os algoritmos utilizados e os passos realizados para a preparação dos dados adquiridos. O Capítulo 4 aborda os testes realizados e os resultados obtidos. Por fim, o Capítulo 5 conclui o trabalho, indicando sua contribuição e possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO

Este capítulo tem por objetivo a apresentação de tecnologias e conceitos utilizados nos estudos e no desenvolvimento deste trabalho. Primeiramente, explicar-se-á como funciona o dispositivo Kinect que será utilizado na captura dos dados.

2.1 Kinect

Kinect é um dispositivo inovador, desenvolvido pela Microsoft, que possui uma câmera que capta a posição em três dimensões do ambiente à sua frente, uma câmera de vídeo colorida que captura as cores nos três componentes: vermelho, verde e azul (RGB), quatro sensores de áudio capazes de capturar sons em três dimensões e um motor para movimentação vertical, que foi desenvolvido para ser utilizado com o videogame XBOX 360. Este equipamento permite ao usuário interagir com o XBOX 360 sem a necessidade de tocar em um controle, através de uma interface natural que é o próprio corpo humano. Através de gestos e comandos por voz, o jogador pode navegar em interfaces do videogame e jogar jogos que tenham compatibilidade com o dispositivo.

O Kinect foi lançado na América do Norte em quatro de novembro de dois mil e dez e bateu o recorde de dispositivo eletrônico vendido mais rapidamente ao consumidor, chegando a oito milhões de unidades vendidas em seus primeiros sessenta dias (GUINNESS WORLD RECORDS). Na Figura 2 pode ser visto uma foto do dispositivo Kinect.

Esta tecnologia inovadora é uma combinação de *hardware* e software contida dentro do aparelho. Além de trazer as câmeras e os sensores de áudio, estes possuem peculiaridades que os tornam muito eficientes.

Figura 2 - Partes do *hardware* do Kinect

A câmera de profundidade é composta por um projetor infravermelho que emite radiações infravermelhas invisíveis ao olho humano e um sensor CMOS monocromático que recebe os dados das radiações refletidas no ambiente. Deste modo, o Kinect fornece para o usuário uma imagem composta de pontos, onde cada ponto representa a distância real do aparelho até o primeiro objeto no qual as radiações infravermelhas foram refletidas. Esta tecnologia possui algumas limitações, como por exemplo, um limite da distância mínima e máxima que o sensor consegue alcançar. A Microsoft recomenda uma distância mínima de um metro, e distância máxima de cinco metros para os sensores funcionarem normalmente. A partir destes limiares os dados se tornam nulos ou pouco precisos. Para captura de todo corpo humano (dos pés à cabeça) a Microsoft aconselha a distância mínima de 1,8 metros.

A câmera de vídeo é capaz de capturar cenas com resolução VGA, ou seja, 640x480 *pixels* de definição e uma velocidade de 30 quadros por segundo (FPS).

Sem o software revolucionário que faz uso dos dados que o *hardware* capta (CRAWFORD, 2011), o Kinect não seria o mesmo. Este software é capaz de rastrear quarenta e oito pontos do corpo humano de cada jogador à frente do aparelho. Esta técnica foi desenvolvida com o pensamento de que todos os simples movimentos do corpo humano fossem entradas para o dispositivo. Sendo assim, os desenvolvedores decidiram não programar essa combinação de movimentos, e sim ensinar o sistema como reagir.

O Kinect se comunica com o computador e com a aplicação por meio de *drivers* e de uma biblioteca de funções, chamada Natural User Interface (NUI) Library. A arquitetura de funcionamento do dispositivo pode ser visualizada na Figura 3.

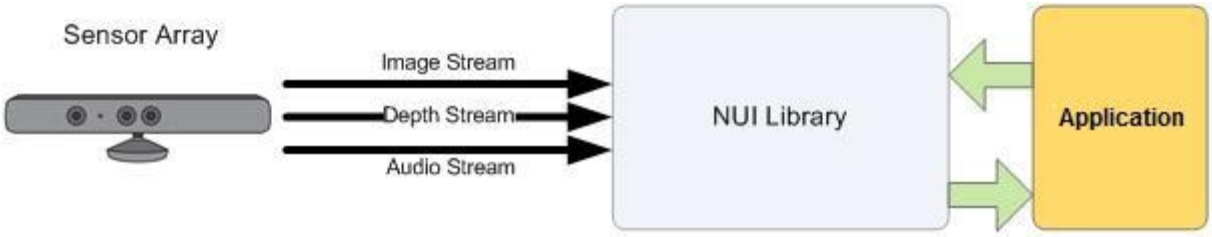

Figura 3 – Arquitetura do dispositivo Kinect

2.1.1 Real-Time Human Pose Recognition

Os desenvolvedores do sistema do Kinect processaram uma grande quantidade de dados de capturas de movimentos, em cenários da vida real, utilizando uma máquina de aprendizado. Esta foi capaz de mapear os dados para modelos que representavam pessoas em fisionomias diferentes. Com estes dados, os pesquisadores conseguiram ensinar o sistema a detectar milhares de tipos de poses humanas, mapeando-as para um esqueleto tridimensional (SHOTTON, FITZGIBBON, *et al.*, 2010). Este algoritmo gerado a partir do aprendizado do sistema foi chamado de *Real-Time Human Pose Recognition*.

Com o esqueleto gerado pelo Kinect, a partir de seu “cérebro”, foi possível a detecção de gestos e movimentos, alavancando assim, a gama de possibilidades de programas e jogos que pudessem ser gerados a partir deste dispositivo.

Os dados referentes ao reconhecimento de jogadores e de seus esqueletos são processados diretamente no *hardware* do Kinect, sem a necessidade de um software específico para trazer esta inovação ao computador ou ao XBOX.

2.1.2 Pacote de Desenvolvimento de Software para o Kinect

O Kinect for Windows SDK é uma ferramenta que permite ao desenvolvedor fácil acesso às capacidades oferecidas pelo dispositivo Kinect. Este pacote de desenvolvimento inclui drivers, acesso aos dados do sensor de profundidade e da

câmera colorida RGB, funções para movimentação do motor, funções para rastreamento de corpos humanos, documentação e exemplos de aplicações (MICROSOFT CORPORATION, 2011). Este pacote funciona apenas no sistema operacional Windows 7 e fornece capacidades para desenvolvedores criarem aplicativos utilizando as linguagens de programação C++, C# ou Visual Basic no Microsoft Visual Studio 2010. O SDK não fornece nenhum algoritmo de visão computacional ou processamento de imagens para serem aplicados nos dados adquiridos pelos sensores do Kinect. Caso o usuário queira utilizar filtros ou algoritmos de segmentação ou reconhecimento de elementos o desenvolvedor precisará adicionar outras bibliotecas ao seu projeto.

2.2 *Trimap*

Trimap é um mapa de cores que especifica regiões em uma imagem em três camadas. Ele é constituído por três cores que especificam regiões de fundo (*background*), regiões de primeiro plano (*foreground*) e regiões desconhecidas que contornam essas duas regiões. Algoritmos que se utilizam de *trimaps* operam sobre a região desconhecida para especificar se os *pixels* dessa região pertencem ao fundo ou ao primeiro plano da imagem.

É importante salientar que o *trimap* tem grande importância e influência na qualidade da extração de elementos da imagem. A região desconhecida do *trimap* deve ser a menor possível, ou seja, maiores serão as informações de *foreground* e *background* e menos variáveis desconhecidas precisam ser estimadas (GASTAL, 2009).

Existem dois tipos principais de *trimaps*, o *stroke trimap*, utilizado para diminuir a intervenção do usuário na hora de especificar o *trimap* e o *filled trimap*, utilizado para gerar resultados mais precisos para algoritmos de *matting*.

2.2.1 Stroke Trimap

O modelo de segmentação *stroke trimap* consiste em demarcar traços a fim de delimitar o *background* e o *foreground* da uma imagem de maneira simples, mas com pouca precisão.

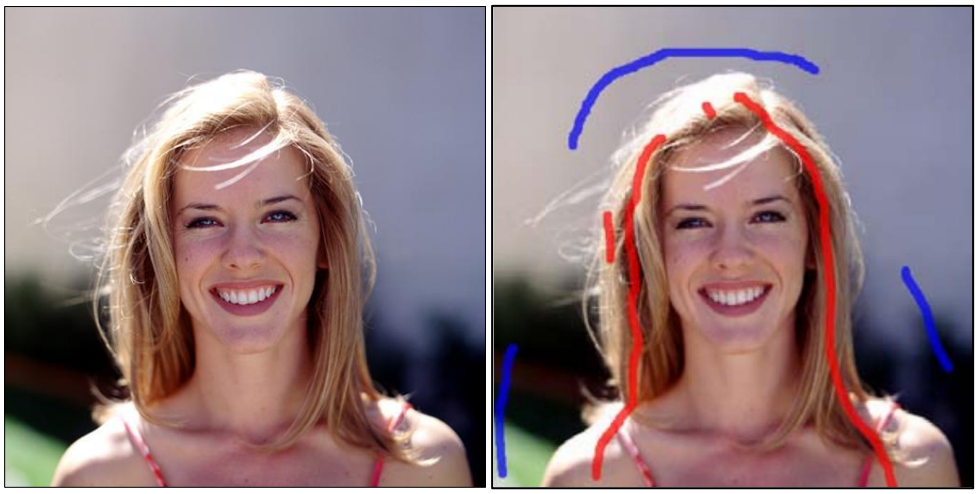

Figura 4 - Stroke *Trimap* gerado com a intervenção do usuário

Neste modelo, o usuário deve traçar linhas na imagem nas regiões de *foreground* e de *background* como pode ser visto na Figura 4. Todos os *pixels* que não foram marcados são tratados como desconhecidos, por isso, algoritmos que se utilizam desta abordagem podem precisar de muitos traços adicionais para gerar resultados satisfatórios, além de terem mais tempo de processamento (DIGITAL FILM TOOLS, LLC.).

2.2.2 Filled Trimap

O modelo de segmentação *filled trimap* é usado para marcar definitivamente o *background* e o *foreground* da imagem, deixando uma pequena quantidade de *pixels* para a região desconhecida. Deste modo, toda a imagem é pintada com uma das três cores: preta (região de fundo), cinza (região desconhecida) ou branca (região de primeiro plano).

Figura 5 - Filled *Trimap* gerado com a intervenção do usuário

Filled trimaps são melhores, comparados a *stroke trimaps*, pois geram resultados mais precisos e garantem um menor tempo de processamento para os algoritmos de *alpha matting* por deixarem uma menor região desconhecida e por segmentarem mais precisamente a imagem (DIGITAL FILM TOOLS, LLC.). Um exemplo de *filled trimap* pode ser visto na Figura 5.

2.3 Alpha Matting

O *matting* tradicional é o processo de separação de uma imagem em dois planos diferentes, ou seja, o processo de separação de uma imagem em duas camadas distintas, normalmente especificadas pela profundidade da imagem, que visa extrair objetos que aparecem em primeiro plano em uma imagem ou vídeo, separando-os de seu fundo. Em uma imagem normal não temos especificações da profundidade dos *pixels* visíveis, por isso este trabalho pode tornar-se muito custoso.

Existem várias técnicas que conseguem separar elementos de uma imagem de seu fundo, mas a maioria destes algoritmos necessita de dados que devem ser especificados pelo usuário. O *trimap* é um destes dados que especificam os objetos à frente e objetos ao fundo da imagem. A partir destas informações os algoritmos conseguem a partir da semelhança de *pixels* de borda segmentar o primeiro plano da imagem.

Alpha Matting é uma técnica amplamente utilizada em visão computacional e processamento de imagens e uma das tarefas mais importantes da fotografia e

edição de vídeo, efeitos especiais de cinema e produção de conteúdos digitais para gerar imagens mais reais (CHO, GROSS, *et al.*, 2009). Esta se refere a *pixels* que possuem informações de cores de ambas as camadas (*background e foreground*) e, portanto, tratam do processo de extração de parte da informação presente no *pixel* de uma imagem I de acordo com um modelo de combinação convexa de cor do *foreground*, representado pela letra F_i, da cor do *background*, representado pela letra B_i e da cor visível de um *pixel*, representado pela letra I_i. Desta maneira, a equação estabelecida matematicamente por Porter e Duff (1984), que modela esta combinação, é dada por:

$$I_i = \alpha_i * F_i + (1.0 - \alpha_i) * B_i$$

onde o fator de mistura do *pixel* é dado pela opacidade α_i, chamado de *alpha matte*. O objetivo dos algoritmos de *matting* é estimar os melhores valores para a tupla (F_i, B_i, α_i) que representam cada *pixel* da imagem. Deste modo, calcula-se, para cada *pixel* pertencente a um objeto, uma estimativa da cor original e do nível de transparência do *pixel*.

2.3.1 Matting with a Depth Map

Um algoritmo desenvolvido por Pitié e Kokaram (2010) recentemente publicado, *Matting With a Depth Map*, utiliza-se de uma câmera de profundidade para gerar uma aproximação para valores próximos a regiões delimitadas pelo *stroke trimap*. Os autores citam que a geração de um *trimap* com alta precisão é uma das tarefas mais importantes para a operação de *matting* e propõem uma abordagem adaptativa para gerar a região desconhecida do *trimap* de acordo com a imprecisão do primeiro plano da imagem (PITIÉ e KOKARAM, 2010).

2.3.2 Shared Matting

Um algoritmo de *alpha matting* recentemente publicado que consegue realizar seu processamento em tempo real é o algoritmo *Shared Matting* (GASTAL e OLIVEIRA, 2010). A fundamentação deste algoritmo diz que, para um tamanho pequeno de vizinhança, *pixels* tendem a compartilhar atributos similares e, portanto,

não se devem tratar todos os *pixels* da borda desconhecida do *trimap* de forma independente, pois isso geraria muito processamento redundante. A quantidade de computação requerida pode então ser reduzida significativamente de forma segura por uma cuidadosa seleção de *pixels* candidatos para fundo e de primeiro plano. Este trabalho computacional pode ser distribuído entre os *pixels* vizinhos, levando a uma melhora considerável no desemprenho do algoritmo. Além disso, as operações necessárias podem ser realizadas em paralelo em unidades de processamento gráfico (GPUs) programáveis.

Figura 6 - Exemplo de resultado obtido a partir do algoritmo Shared Matting para remover o plano de fundo da imagem.

Este algoritmo é utilizado para validação dos *trimaps* gerados, ou seja, utilizar-se-á este algoritmo para segmentar as imagens geradas a partir do Kinect. Um exemplo de resultado gerado a partir do algoritmo *Shared Matting* pode ser visto na Figura 6.

2.4 Dilatação e Erosão

A maioria dos algoritmos de segmentação e de reconhecimento de padrões não é tolerante a pequenos ruídos e a pequenos defeitos. As operações morfológicas podem preparar a imagem para ser submetida por estes algoritmos (BAUERMANN, 2008).

A dilatação e a erosão são dois operadores básicos da área de morfologia matemática que são comumente aplicados a imagens binárias, mas também podem ser utilizados em imagens com tons de cinza.

Morfologia é o estudo da forma e em processamento de imagens, morfologia matemática é o nome que se dá aos métodos, desenvolvidos por Georges Matheron e Jean Serra em 1964, que têm como objetivo estudar a geometria de uma imagem.

Para os exemplos a seguir, consideramos o fundo como sendo a região branca e os objetos as regiões pretas.

O efeito do operador de dilatação em uma imagem é ampliar gradualmente as regiões de borda de *pixels* de *foreground*. Assim, áreas de primeiro plano crescem em tamanho, enquanto buracos no interior dessas regiões ficam menores. Sua principal função é preencher buracos em imagens que necessitem de uma suavização. Um exemplo de uma operação de dilatação pode ser visto na Figura 7. Sua implementação é baseada em um elemento estruturante que é iterado por todos os *pixels* de *foreground* da imagem. A cada iteração, o elemento estruturante é sobreposto ao *pixel* atual e *todos* os *pixels* do elemento estruturante são pintados da mesma cor do *foreground*.

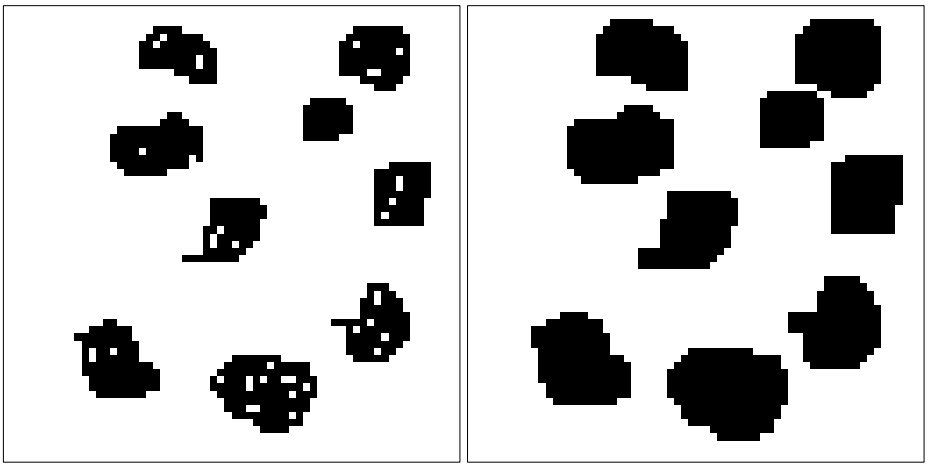

Figura 7 - Exemplo do operador morfológico de dilatação

O efeito do operador de erosão em uma imagem é desgastar (reduzir) gradualmente as regiões de borda de *pixels* de *foreground*. Assim, áreas de primeiro plano diminuem, enquanto buracos no interior dessas regiões se tornam maiores. Sua principal função é eliminar pequenos ruídos nas imagens a fim de destacar apenas elementos importantes na imagem. Um exemplo de uma operação de erosão pode ser visto na Figura 8. Sua implementação compartilha da mesma ideia do operador de dilatação. A única diferença é que quando se sobrepõe o elemento estruturante ao *pixel* atual que esta sendo iterado, verifica-se se todo elemento

estruturante pertence ao *foreground* da imagem. Caso todo ele pertença, o *pixel* central não é alterado, caso contrário ele é pintado da cor do *background*.

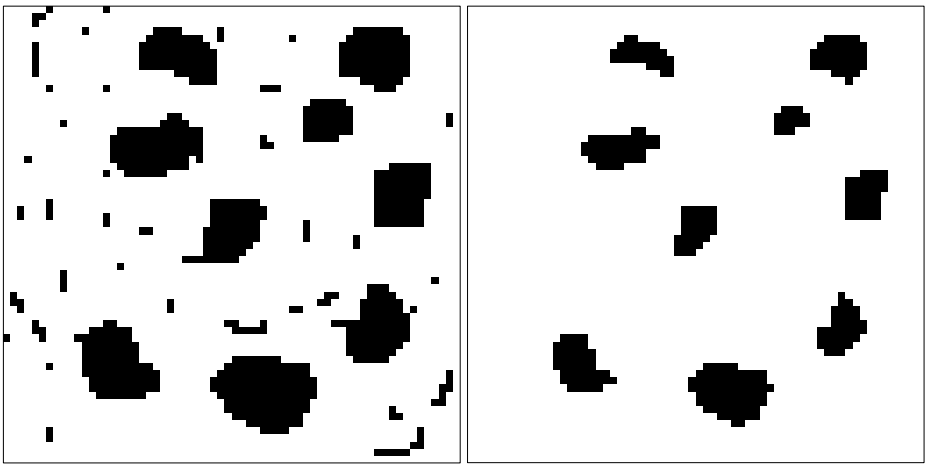

Figura 8 - Exemplo do operador morfológico de erosão

Combinações de dilatações e erosões podem ser aplicadas para formatar a imagem de acordo com o problema que se quer resolver. Estas combinações são chamadas de **abertura**, onde é aplicada uma erosão seguida por uma dilatação, e **fechamento**, onde é aplicada uma dilatação seguida por uma erosão. A abertura tem por objetivo o aumento de buracos internos e eliminação de ruídos e defeitos. O fechamento objetiva unir linhas próximas e o fechamento de buracos internos. Ambas as combinações são mais indicadas ao invés de aplicar apenas uma erosão ou uma dilatação, pois a abertura e o fechamento mantém a área de regiões de *foreground* mais próximas da imagem original.

2.5 Filtro de Mediana

O filtro de mediana pode ser visto como um coringa para processamento de imagens. Quando uma segmentação foi quase satisfatória e se quer apenas finalizar seu resultado, utiliza-se este filtro para tratar pequenas imperfeições (ruídos) na imagem. É possível utilizar também aberturas e fechamentos para tratar estes mesmos problemas, mas é preciso ter bastante cuidado para deixar as bordas das imagens onde elas estavam. O filtro de mediana não altera as bordas da imagem mantendo sua geometria original.

A implementação deste filtro também é bem simples. Devem-se percorrer todos os *pixels* da imagem e verificar qual a mediana dos valores do elemento estruturante aplicado ao *pixel* central. A mediana encontrada é aplicada ao *pixel* que se está iterando. Um exemplo de resultado deste filtro pode ser visto na Figura 9.

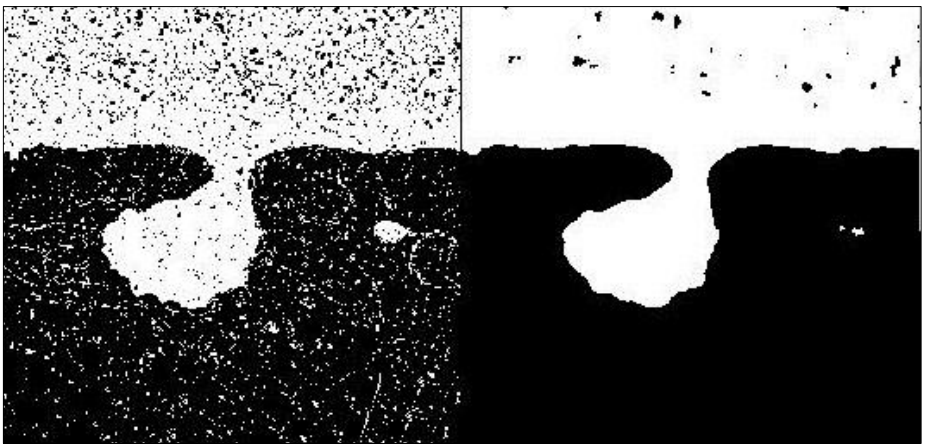

Figura 9 - Exemplo do filtro de mediana

2.6 Linguagens de Programação e Bibliotecas

As linguagens de programação escolhidas neste trabalho foram influenciadas pela necessidade de se programar com o SDK do Kinect. A linguagem escolhida entre as possíveis foi C# por se tratar de uma linguagem de mais alto nível e muito natural para criar e usar componentes de software (MICROSOFT CORPORATION, 2007). Esta linguagem foi escolhida também, por trazer uma interface mais agradável e mais intuitiva para trabalhar com o Kinect. Com o objetivo de renderizar os *trimaps* foi utilizado um *wrapper* chamado *Open Toolkit Library* (OpenTK) que fornece acesso às funções do OpenGL, OpenCL e OpenAL para a linguagem C#. Para o processamento do *trimap* foi utilizado o OpenGL Shading Language (GLSL) que é uma linguagem de *shader* de alto nível e tem sua sintaxe baseada na linguagem C.

2.6.1 Linguagem C#

A linguagem C# é simples, moderna e orientada a objeto. Ela tem suas raízes na família de linguagens C e torna-se familiar para programadores de C, C++ e Java

(MICROSOFT CORPORATION, 2007). Os arquivos fonte em C# são do tipo “.cs” e todos os fontes em um projeto são compilados diretamente para um arquivo executável do tipo “.exe” ou do tipo “.dll”, quando é criada alguma biblioteca, portanto não existem arquivos intermediários como “.obj” ou “.o”.

Na linguagem C# várias características ajudam na construção de aplicativos robustos e duráveis: A coleta de lixo é automática recuperando a memória ocupada por objetos não utilizados, o tratamento de exceção fornece uma abordagem estruturada e extensível para a detecção de erros e recuperação e o projeto de linguagem segura torna impossível ler a partir de variáveis não inicializadas, ler índices de matrizes além de seus limites, ou realizar conversões de tipo não equivalentes.

A linguagem possui um sistema de tipo unificado. Em C#, todos os tipos, até mesmo os primitivos, assim como *int* e *double*, herdam de um tipo objeto. Desta maneira, todos os tipos compartilham um conjunto de operações e valores de qualquer tipo podem ser armazenados, transportados e operados de maneira consistente. Além disso, a linguagem suporta tipos definidos pelo usuário e tipos de referência (ponteiros), permitindo a alocação dinâmica de objetos.

2.6.2 OpenTK

OpenTK é um projeto livre implementado em C# que roda em vários sistemas operacionais, como Windows, Linux, Mac OS X, e pode ser utilizado por todas linguagens .Net: C#, VB.Net, C++/CLI, F#, Boo e muitas outras. Este projeto permite, através de um *wrapper*, a utilização de ambientes de programação como OpenGL | ES, OpenCL e OpenAL nas linguagens citadas. Esta ferramenta é adequada para jogos, visualizações científicas e todos os tipos de software que requerem gráficos avançados, áudio ou outras capacidades computacionais. Por ser desenvolvido na linguagem de programação C#, este projeto transforma os ambientes de programação citados acima em objetos de alto nível, facilitando a programação com gramáticas consistentes, métodos genéricos e enumerações fortemente tipadas, adequado para o Desenvolvimento Rápido de Aplicação (RAD) (OPENTK, 2006).

2.6.3 GLSL

GLSL é uma linguagem de programação de alto nível, baseada na sintaxe da linguagem C, que permite dar ao programador um controle mais direto do *pipeline* gráfico sem precisar usar linguagens de máquina, como Assembly ou linguagens específicas de *hardware*. Esta linguagem foi escolhida por ser extremamente rápida para processar imagens em tempo real, já que todos os *pixels* da imagem são calculados em paralelo na unidade de processamento da placa de vídeo (GPU).

Neste trabalho será substituído apenas o *fragment shader* do *pipline* gráfico original de processamento, pois serão enviadas texturas para a GPU com as informações adquiridas do Kinect a fim de gerar os *trimaps*.

3 DESENVOLVIMENTO

Este capítulo descreve os passos realizados durante o desenvolvimento deste trabalho. Primeiramente, o capítulo traz uma visão geral da implementação e os passos necessários para alcançar o objetivo proposto. Em seguida, são expostas as etapas necessárias e os algoritmos escolhidos, e as justificativas destas escolhas. Por fim, são abordadas as formas de utilização dos *trimaps* em algoritmos de *alpha matting*.

3.1 Visão Geral

3.1.1 Requisitos

Os requisitos para o correto funcionamento do aplicativo desenvolvido estão relacionados às linguagens de programação nas quais foi escrito e às ferramentas utilizadas. Como o aplicativo é dependente do SDK do Kinect, os requisitos de sistema mínimos para executar o programa são:

- *Hardware*:
    - Dispositivo Kinect Sensor;
    - Processador *dual core* 2.66 GHz;
    - Placa gráfica com suporte ao DirectX® 9.0c;
    - 2 GB de memória RAM.
- *Software*
    - Windows 7 (x64 ou x86);

- Visual Studio 2010 Express;
- Microsoft .NET Framework 4.0;
- *Drivers* (SDK).

Os *drivers* de comunicação do dispositivo Kinect com o computador podem ser obtidos gratuitamente no site oficial do SDK (MICROSOFT CORPORATION, 2011) em sua versão atual não comercial beta 2. A Microsoft garante que irá lançar uma versão comercial futuramente com mais recursos para desenvolvedores que desejarem vender seus aplicativos, embora a empresa não tenha dado prazos de quando isso irá acontecer.

Além do SDK do Kinect também é necessária a instalação da biblioteca OpenTK que pode ser obtida em seu site oficial (OPENTK, 2006).

3.1.2 Funcionamento

Este trabalho compreende as etapas de aquisição de dados, preparação de dados e geração de *trimaps*. A primeira refere-se à programação utilizando o SDK do Kinect para capturar os dados dos sensores e armazená-los em estruturas para serem encaminhados à segunda etapa. Na fase de preparação de dados, as imagens serão formatadas com algoritmos de processamento de imagens para serem segmentadas, finalmente, pela terceira etapa, a geração dos *trimaps*.

3.2 *Pipeline* de funcionamento

O aplicativo desenvolvido está dividido em três módulos, que podem ser vistos na Figura 10.

Figura 10 - Pipeline de funcionamento do aplicativo

3.2.1 Aquisição de Dados

A aquisição dos dados compreende a captura das imagens pelos sensores do Kinect e a interpretação desses dados para serem encaminhados à etapa de preparação de dados. Para isso, dividiu-se a etapa de aquisição de dados em quatro fases: Inicialização, Abertura, Captura e Calibração (Figura 11). Para estas fases, faz-se necessário compreender como funciona o SDK do Kinect.

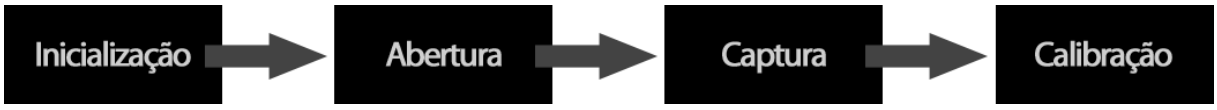

Figura 11 - Fases da etapa de Aquisição de Dados

A primeira atividade a ser feita é a inicialização do Kinect, para que ele fique pronto para captar dados de seus sensores. Nesta primeira etapa, escolhem-se quais capacidades do Kinect serão utilizadas durante a execução do programa. Para este trabalho, necessita-se da câmera de vídeo, do sensor de profundidade e da informação do corpo humano, por isso, para inicializar o Kinect utilizar-se-á a função **`Initialize`** com os seguintes parâmetros mostrados pela Figura 12.

```
private Runtime kinect = Runtime.Kinects[0];
kinect.Initialize(RuntimeOptions.UseColor|RuntimeOptions.UseDepthAndPlayerIndex);
kinect.DepthStream.Open(ImageStreamType.Depth, 2,
               ImageResolution.Resolution320x240, ImageType.DepthAndPlayerIndex);
kinect.VideoStream.Open(ImageStreamType.Video, 2,
               ImageResolution.Resolution640x480, ImageType.Color);
```

Figura 12 - Inicialização do SDK do Kinect em C#

Além da inicialização, deve-se abrir a câmera e os sensores com as configurações desejadas. Atualmente, a única configuração de resolução que o Kinect permite para poder calibrar as duas imagens é a resolução de 320x240 para o *stream* de profundidade e 640x480 para o *stream* da câmera de vídeo. Nota-se que estas resoluções são muito baixas para expressar pequenos detalhes nas imagens, sendo esta uma das limitações que serão abordadas neste trabalho.

Após esta fase inicial, deve-se capturar os *frames* dos sensores para que sejam interpretados e calibrados para dar fim a fase de aquisição de dados.

Existem duas maneiras para capturar os *frames* dos sensores. Uma delas é a captura orientada a eventos chamada de *Event Model*. Desta maneira, quando um novo quadro de imagem está pronto, um evento é sinalizado e é chamada uma função, especificada após a abertura do sensor, que tratará o que se deseja fazer com o quadro. A outra maneira de captura de quadros e que foi escolhida para ser utilizada neste trabalho é chamada de *Polling Model.* Neste modelo deve-se “ouvir” o Kinect, perguntando-o a todo instante se existe um novo quadro pronto para ser utilizado. Este modelo foi escolhido porque o *Event Model* não funcionava quando era utilizada a *render* da biblioteca OpenTK, já que o fluxo do código não desviava para os eventos do Kinect. A implementação do modelo *Polling Model* pode ser vista na Figura 13. Deve-se sempre testar o retorno da função **`GetNextFrame`** pois nem sempre existirá um quadro novo pronto para ser capturado.

```
// try to get a new depth frame
depthFrame = kinect.DepthStream.GetNextFrame(0);
if (depthFrame == null) return false;

// try to get a new color frame
colorFrame = kinect.VideoStream.GetNextFrame(0);
if (colorFrame == null) return false;
```

Figura 13 - Leitura de frames dos sensores do Kinect em C#

Após capturar os dois quadros, de cor e de profundidade, deve-se fazer a verificação dos corpos humanos reconhecidos pelo hardware do dispositivo. Para isso, deve-se percorrer todos os quadros de profundidade e fazer a verificação de cada pixel executando-se o código da Figura 14. Caso a variável *player* tenha seu valor igual a zero, este pixel não pertencerá a nenhum corpo, caso contrário, o número inteiro da variável representará o número do corpo reconhecido pelo Kinect.

```
/* extract the id of a tracked player from the first
   bit of depth data for this pixel */
int player = depthFrame.Image.Bits[depthIndex] & 7;
```

Figura 14 - Verificação de *pixel* de corpo humano

Depois de ter informações dos corpos deve-se calibrar os quadros para que cada pixel de mesma posição nos dois quadros represente informações do mesmo ponto do ambiente. Para converter as coordenadas de profundidade para as coordenadas da imagem colorida utiliza-se o método **`GetColorPixelCoordinatesFromDepthPixel`** transformando o espaço de resolução de 640x480 para 320x240. A função pode ser vista na Figura 15.

```
/* find a pixel in the color image which
   matches this coordinate from the depth image */
int colorX, colorY;
kinect.NuiCamera.GetColorPixelCoordinatesFromDepthPixel(colorFrame.Resolution,
                                                        colorFrame.ViewArea,
                                                        depthX, depthY, 0,
                                                        out colorX, out colorY);
```

Figura 15 - Calibração de *pixels* das imagens de cor e profundidade

As variáveis **`colorX`** e **`colorY`**, argumentos passados por referência, representam as novas coordenadas de cor do pixel de profundidade representado pelas coordenadas **`depthX`** e **`depthY`**.

Foram utilizados dois vetores estáticos do tipo *byte*, ambos com comprimento de 307.200 (320 * 240 * 4) posições, para armazenar os dois quadros capturados pelo Kinect. Desta forma os valores são apenas substituídos pelos valores antigos e não se perde tempo de processamento alocando espaço na memória.

3.2.2 Preparação de Dados

A preparação de dados compreende a interpretação e refinamento das imagens obtidas da etapa anterior. Esta fase é subdividida em outras três: a Criação, a Leitura e o Refinamento (Figura 16).

Figura 16 - Fases da etapa de Preparação de Dados

A primeira fase compreende a criação dos *shaders* e das texturas. Ambos são criados uma única vez para melhor desempenho do programa. Foram criados quatro programas de *shader*: o filtro de mediana, os operadores morfológicos de erosão e de dilatação e o *trimap*.

A fase de leitura é responsável por “perguntar” ao módulo do Kinect se os quadros estão prontos para serem lidos. Caso eles ainda não estiverem prontos, esta fase fica aguardando até que a resposta seja afirmativa, conforme o método *Pooling Model*.

Depois de lidos os quadros, entra-se na fase de refinamento dessas informações. Esta fase é muito importante, pois elimina pequenos resíduos que existem na imagem gerada pelo sensor do Kinect, deixando uma segmentação mais suave e real. Um exemplo de *trimap* gerado com e sem a etapa de refinamento respectivamente pode ser visto na Figura 17.

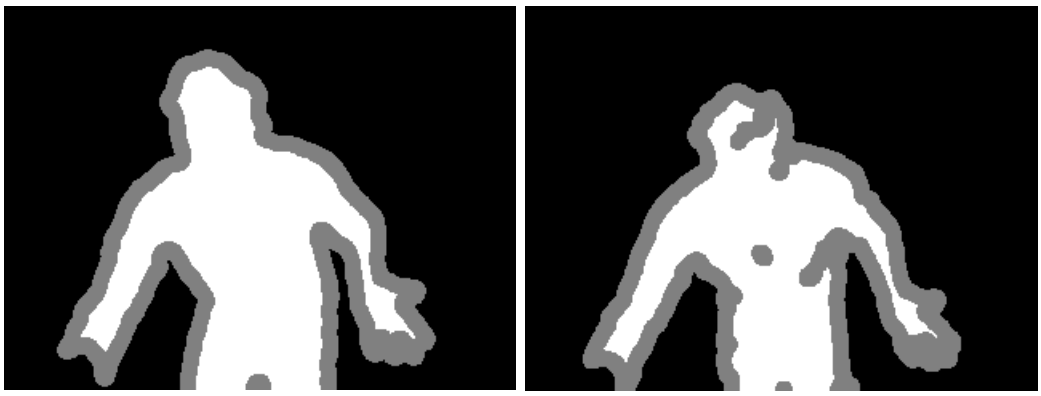

Figura 17 - Exemplo de *trimap* gerado com e sem a etapa de refinamento respectivamente

Pode-se perceber claramente que o refinamento da imagem dará mais segurança ao algoritmo de *alpha matting* utilizado para segmentar a imagem original, pois regiões desconhecidas serão menores e mais precisas.

Para esta etapa, realizaram-se testes de filtros para que se obtivesse um resultado bom para este problema. Os algoritmos utilizados são adaptados para se comportarem do jeito que é necessário. Ao final dos testes, encontrou-se uma sequência de filtros que geraram um resultado satisfatório. Aplicou-se primeiramente um filtro de medianas para que pequenos resíduos fossem eliminados. Posteriormente, foi aplicado um fechamento, para que pequenos buracos internos fossem completamente preenchidos. O fechamento aplicado é composto de uma dilatação e uma erosão. Estas operações são realizadas se utilizando do elemento estruturante no formato diamante de tamanho de raio dois, que pode ser visto na Figura 18.

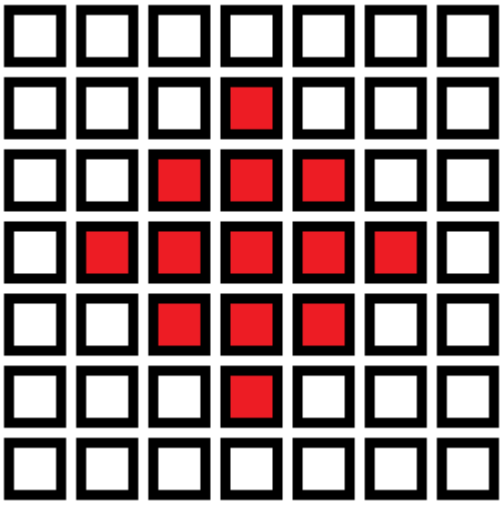

Figura 18 - Elemento estruturante utilizado na aplicação das erosões e dilatações

Para que estas operações pudessem ser executadas uma após a outra, utilizou-se da técnica *render to texture.* Este é um recurso muito útil para realizar processamentos genéricos em GPU, pois se deve recuperar o resultado obtido por um programa de *shader* para poder aplicar o próximo.

Para se utilizar a técnica *render to texture* define-se o *buffer* de escrita e leitura para o *back buffer* para que se possa ler o que foi escrito. Para cada filtro, desenha-se normalmente o resultado na tela e executa-se a função **`renderToTexture`** que pode ser vista na Figura 19.

```
int renderToTexture()
{
        GL.BindTexture(TextureTarget.Texture2D, renderTexture);

        GL.CopyTexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, 0, 0,
                          textureWidth, textureHeight, 0);

        GL.BindTexture(TextureTarget.Texture2D, 0);

        return renderTexture;
}
```

Figura 19 - Função que implementa a técnica *render to texture*

Nesta função, marca-se a textura que se quer substituir pela tela que foi desenhada anteriormente a esta chamada. Usa-se então a função **`CopyTexImage2D`** para copiar a região da tela especificada por seus parâmetros. Utiliza-se então a mesma função usada anteriormente, com a finalidade agora de desmarcar a textura, para que mais nenhuma modificação seja aplicada à textura que recém foi copiada.

Uma limitação que o algoritmo de reconhecimento de corpos do Kinect possui é que ele gera alguns erros de detecção quando existem objetos próximos aos corpos detectados. Desse modo, o algoritmo não consegue distinguir que o objeto próximo não é um corpo, por isso, aconselha-se um espaço amplo para a que a captura do ambiente e a segmentação dos corpos seja ideal.

3.2.3 Geração de *Trimaps*

A etapa de geração de *trimaps* compreende a segmentação das imagens em três regiões distintas, o *foreground*, representado pela cor branca, o *background*, representado pela região preta, e a região desconhecida, representada pela região cinza.

Esta última etapa é simples, pois a imagem já está no formato desejado. Um exemplo de imagem de entrada para o algoritmo do *trimap* pode ser vista na Figura 20.

Figura 20 - Exemplo de imagem de entrada para algoritmo do *trimap*

O algoritmo de *trimap* foi implementado também em *shader* para um melhor desempenho. Em pseudocódigo pode-se dizer que o algoritmo varre toda a imagem procurando *pixels* de borda (*pixels* brancos ao lado de pretos) e pintando estas regiões encontradas com pequenos círculos cinza. Mas esta ideia não pode ser usada para um programa em *shader*, pois nele se pode apenas modificar a cor do *pixel* a qual estamos iterando. Desta maneira, ao invés de testar se o *pixel* que está sendo iterado pertence a uma borda, deve-se testar uma redondeza de cada *pixel* procurando por uma diferença de cor. Caso esta for encontrada, o *pixel* deve ser pintado de cinza. O resultado deste algoritmo, usando como entrada a Figura 20, pode ser visto na Figura 21. Este algoritmo foi chamado de “detector de bordas próximas”.

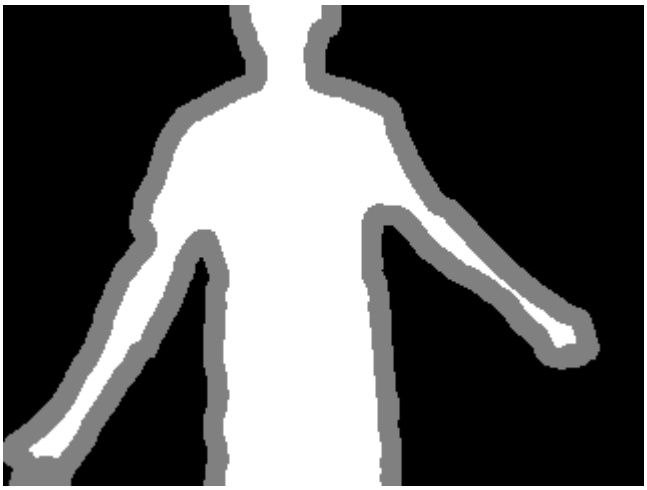

Figura 21 - Resultado do algoritmo de *trimap* gerado a partir da figura 19

4 RESULTADOS

A fim de alcançar os objetivos deste trabalho, testou-se o desempenho dos *trimaps* a partir de dois ambientes de teste e avaliou-se o desempenho de execução do programa implementado a partir de três casos de teste.

4.1 Ambiente de Teste

Para a realização dos testes de desempenho, utilizaram-se dois ambientes. Em ambas as máquinas foram utilizadas a plataforma Windows 7, de acordo com os requisitos da Seção 3.1.1. Na Tabela 1 são descritas as configurações de cada ambiente.

Tabela 1 - Ambientes de teste utilizados

	Processador	**Placa de Vídeo**	**Memória RAM**
PC 1	Intel Core i7 870 @ 2.93 GHz	ATI Radeon HD 5850	4,00 GB
PC 2	Intel Core i7 860 @ 2.80 GHz	NVIDIA GeForce GT 230	8,00 GB

4.2 Casos de Teste

Os casos de teste, utilizados para se avaliar a o desempenho dos *trimaps* gerados, foram determinados pela alteração das variáveis paramétricas do algoritmo de detecção de bordas próximas, as quais definem a largura da região desconhecida. Os casos de teste foram utilizados para determinar qual a largura da região desconhecida que geravam os melhores resultados para a segmentação do algoritmo de *alpha matting*. Os casos escolhidos podem ser vistos na Tabela 2.

Tabela 2 - Casos de teste utilizados para a largura da região desconhecida

Caso de Teste	**Largura da Região Desconhecida Interna ao Corpo**	**Largura da Região Desconhecida Externa ao Corpo**
1	2 *pixels*	3 *pixels*
2	3 *pixels*	2 *pixels*
3	3 *pixels*	3 *pixels*

A Microsoft confirma que não garante a sincronização completa dos quadros obtidos através do SDK, mas vem trabalhando para que esta seja a menor possível. Em sua versão atual, beta 2, a sincronização já é bem melhor que nas versões anteriores. Considerando esta diferença nos *frames*, os casos foram escolhidos para manter a largura total da borda próxima de cinco *pixels*, que foi o valor médio identificado que o *frame* de profundidade varia para o *frame* da imagem colorida.

Para cada caso de teste realizou-se a geração dos *trimaps* e a aplicação do algoritmo *Shared Matting*. Os testes realizados podem ser visualizados nas figuras 22, 23 e 24.

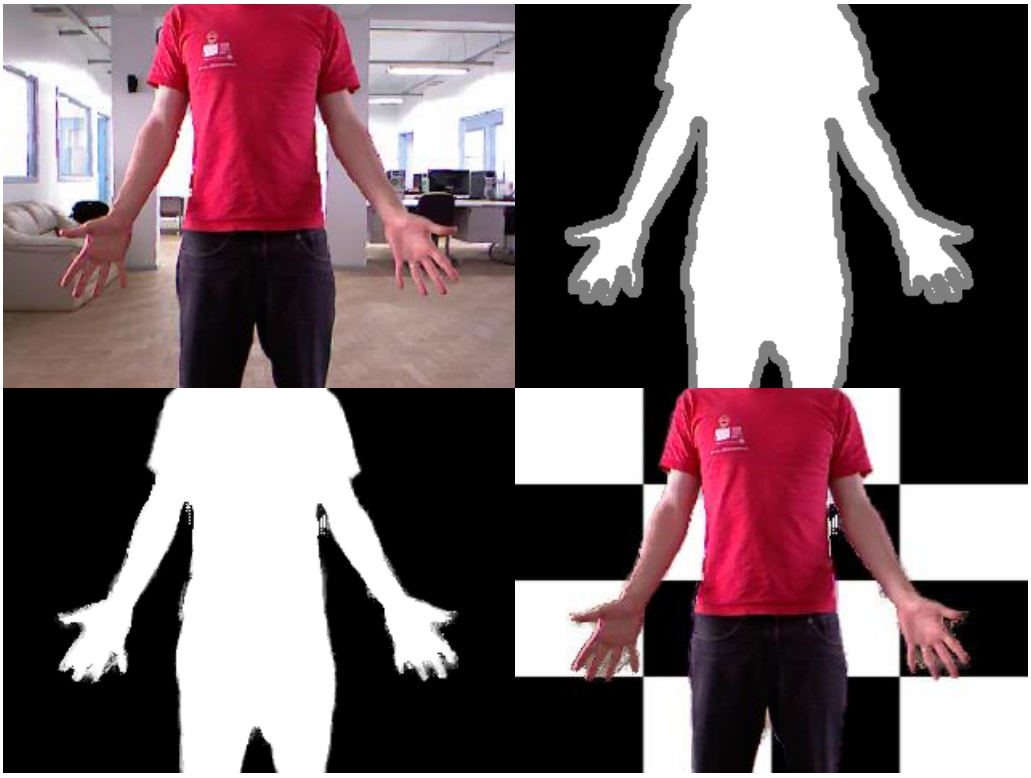

Figura 22 - Caso de teste 1

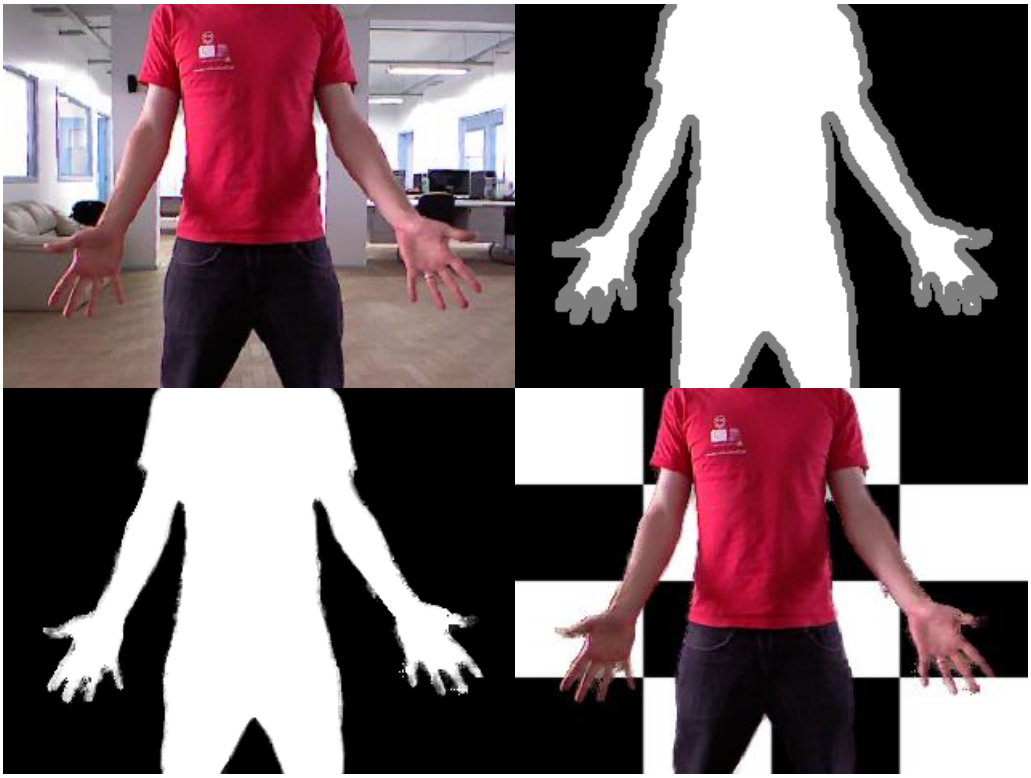

Figura 23 - Caso de teste 2

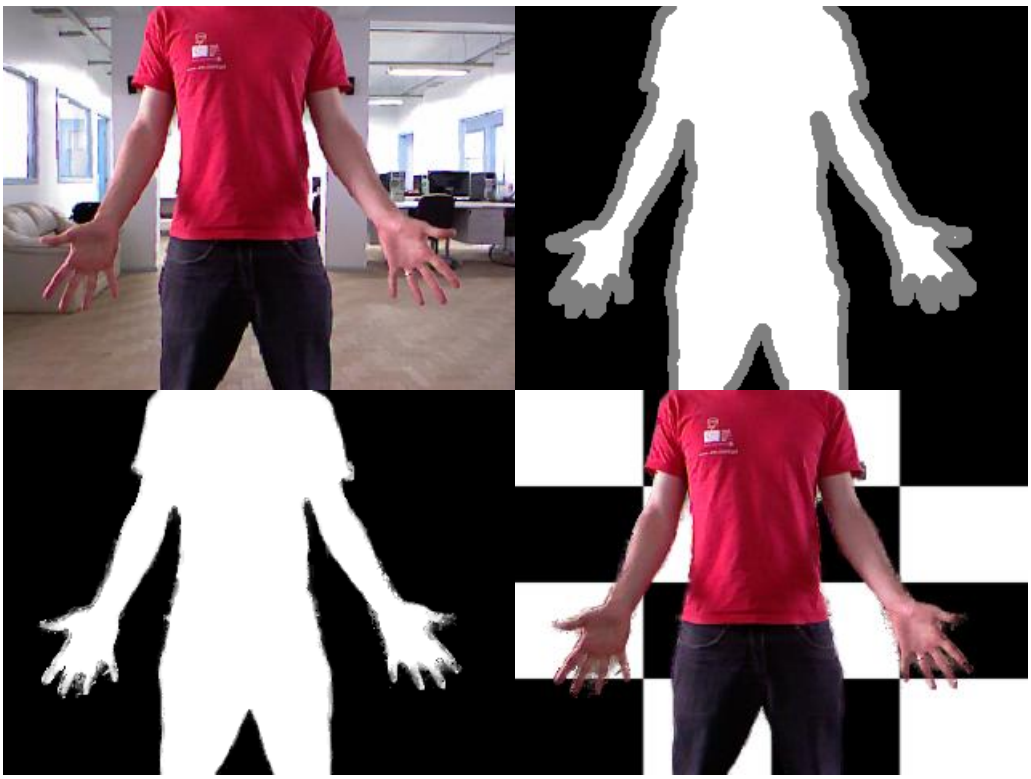

Figura 24 - Caso de teste 3

Houve uma grande quantidade de testes realizados com larguras menores que quatro *pixels* e maiores que seis *pixels*, mas os resultados foram logo descartados por segmentarem a imagem de forma irregular, ora segmentando regiões de fundo como se fossem regiões de frente, ora mantendo a região desconhecida muito grande e imprecisa.

Testou-se também o desempenho, nos dois ambientes, dos algoritmos implementados, para que todo processo pudesse ser executado em tempo real. Como o Kinect opera até um máximo de 30 FPS, não poderia se testar apenas quantos quadros por segundo o programa executaria. Calculou-se então o tempo de realizar uma iteração completa (desde a captura da imagem até a geração do trimap) para verificar o desempenho do programa. Os resultados obtidos podem ser visto na Tabela 3. Estimou-se um valor para o FPS, ou seja, quantos quadros seriam renderizados em mil milésimos de segundo, caso o Kinect não tivesse a limitação de 30 quadros. Para esta estimativa tomou-se como base o tempo médio para a execução e renderização de um *frame*.

Tabela 3 - Resultados de desempenho de execução obtidos nos ambientes de teste

Ambiente de teste	**PC 1**	**PC 2**
Tempo Máximo para execução e renderização de um *frame*	0.0315 s	0.017 s
Tempo Mínimo para execução e renderização de um *frame*	0.007 s	0.006 s
Tempo Médio para execução e renderização de um *frame*	0.0175 s	0.007 s
Estimação de quadros renderizados por segundo	57 FPS	142 FPS

A grande diferença entre os resultados obtidos nos dois ambientes se deve ao fato de que os filtros em *shader* executam mais rápido na placa de vídeo da NVIDIA, já que a diferença de memória RAM não é fundamental para esta aplicação.

O desempenho de execução do algoritmo *Shared Matting* é proporcional à quantidade de *pixels* da região desconhecida, por isso, verificou-se que a geração dos *trimaps* de forma precisa diminui o custo computacional do algoritmo de *alpha matting* aplicado posteriormente. Nos testes realizados, os trimaps gerados possuíram apenas 7% de regiões desconhecidas, aproximadamente. Em uma

imagem com resolução de 320x240 a região desconhecida possui aproximadamente 5.400 *pixels*.

Não foram testadas imagens com resoluções maiores que 320x240 já que o Kinect não suporta captura sincronizada deste tipo de imagens.

4.3 Avaliação dos Resultados

A partir dos resultados obtidos pela execução dos três casos de teste, pode-se observar que todos se comportaram razoavelmente bem, mas o caso que obteve os melhores resultados foi o de número 2. Isso se deve ao fato de que a segmentação ocorrida neste caso deixou pouca imprecisão para o algoritmo de *alpha matting*, já que a borda externa era a menor (dois *pixels*).

Pode-se observar também que a segmentação final em todos os casos não foi perfeita como nos resultados obtidos por Gastal e Oliveira (2010, p. 2 e 9). Esta é uma das maiores limitações encontradas no dispositivo Kinect, sua resolução de imagens. Como a técnica *Shared Matting* pode gerar segmentações para imagens de alta resolução em tempo real, o ideal seria ter um dispositivo capaz de capturar imagens em resoluções maiores, como por exemplo, 800x600 *pixels* de definição (aproximadamente 0.5 *megapixel*).

Para os testes que se referem ao desempenho do programa, pôde-se observar que tranquilamente a geração dos *trimaps* pode ser feita em tempo real. Mas deve-se lembrar de que é preciso fazer também a segmentação final da imagem com o algoritmo *Shared Matting*. Para que um programa execute sua rotina em tempo real cada iteração deverá ter em média um tempo de processamento de 33 milissegundos para que esta se repita no mínimo 30 vezes por segundo. Analisando o desempenho de todo processo de geração de *trimaps* e *matting* das imagens, verifica-se que o algoritmo *Shared Matting* necessita realizar sua execução em uma média de 16 milissegundos por iteração para que todo processo possa enfim ser realizado em tempo real.

A partir da tabela de desempenho do algoritmo *Shared Matting* (GASTAL e OLIVEIRA, 2010, p. 7) pôde-se estimar quantos *pixels* este método pode calcular em um tempo de 16 milissegundos. A análise de desempenho para a quantidade de *pixels* da região desconhecida pode ser vista na Tabela 4.

Tabela 4 - Estimativa de desempenho para o algoritmo *Shared Matting*

Número de *pixels* da região desconhecida	**Tempo para calcular um *frame***
96.320	0.032 segundos
85.888	0.029 segundos
77.384	0.028 segundos
23.000	0.016 segundos (estimativa)

Analisando a tabela, verifica-se que o algoritmo *Shared Matting* consegue segmentar uma imagem com aproximadamente 23.000 *pixels* de região desconhecida em 16 milissegundos. Visto que os *trimaps* gerados possuem aproximadamente 5.400 *pixels* para esta região, tranquilamente o objetivo inicial de gerar *trimaps* em tempo real é alcançado.

Mesmo não se testando todo o processo de geração dos trimaps e segmentação do algoritmo *Shared Matting* em imagens com resoluções maiores, fica evidente que este algoritmo consegue realizar este processamento em imagens maiores, pois se verificou que há uma sobra de processamento para o ambiente de teste mais lento.

5 CONCLUSÃO

Este trabalho apresentou a implementação de um programa que gera *trimaps* de corpos humanos em tempo real utilizando o dispositivo Kinect para captura de informações do ambiente para que este pudesse ser segmentado.

Ao longo deste trabalho verificou-se que o Kinect possui algumas limitações, como a sincronização e a resolução dos quadros obtidos através de seus sensores. Porém estas limitações não foram capazes de substituir o dispositivo de aquisição de dados, pela praticidade e custo que o Kinect possui. Como o público-alvo deste *software* é principalmente o consumidor doméstico, nada melhor que utilizar um dispositivo que já se encontra em muitos lares, devido à utilização em conjunto com o *videogame* XBOX 360.

Verificou-se que a geração dos *trimaps* e a segmentação do algoritmo *Shared Matting* podem ser realizadas em tempo real até mesmo no ambiente de teste mais lento (o que executa em maior tempo), validando o objetivo principal deste trabalho.

O programa desenvolvido é um novo recurso para o *matting* de imagens já que o processo de especificação de regiões de *background* e *foreground* das imagens são automáticos e precisos. Esta técnica pode ser utilizada como solução imediata para segmentação de corpos humanos, sem a necessidade de utilização de algoritmos de *Chroma key*.

5.1 Trabalhos Futuros

Como trabalhos futuros, sugere-se a integração do algoritmo de *Shared Matting* com o programa implementado para que o resultado possa ser visto e testado em tempo real.

Pode-se também utilizar o mesmo método para geração de *trimaps* para outros objetos a partir de características paramétricas de reconhecimento. Sugere-se também o aprimoramento da geração dos *trimaps* de corpos humanos pelo reconhecimento das partes do corpo, já que algumas partes exigem maior precisão para a segmentação assim como os dedos da mão. Pode-se também testar a utilização de outros dispositivos de aquisição de imagens com maior resolução para que a segmentação seja mais perfeita.

6 BIBLIOGRAFIA

BAUERMANN, G. Abertura e Fechamento. **ImageSurvey - Processamento de Imagens na prática**, 2008. Disponivel em: <http://www.imagesurvey.com.br/2008/12/abertura-e-fechamento/>. Acesso em: Outubro 2011.

CHO, J. et al. Improving alpha matte with depth information. **IEICE Electronics Express**, 2009. Vol. 6, No. 22 pp. 1602-1607.

CRAWFORD, S. How Microsoft Kinect Works. **How Stuff Works**, 2011. Disponivel em: <http://electronics.howstuffworks.com/microsoft-kinect1.htm>. Acesso em: outubro 2011.

DIGITAL FILM TOOLS, LLC. Online Support Center. **EZ Mask**. Disponivel em: <http://support.digitalfilmtools.com/support/index.php?_m=knowledgebase&_a=viewarticle&kbarticleid=78&nav=0>. Acesso em: Setembro 2011.

GASTAL, E. S. L. Real-Time Alpha Matting for Natural Images and Vídeos, Porto Alegre, 2009.

GASTAL, E. S. L.; OLIVEIRA, M. L. M. Shared Sampling for Real-Time Alpha Matting. **Proceedings of Eurographics, Volume 29 (2010)**, 2010. pp. 575-584.

GUINNESS WORLD RECORDS. Kinect Confirmed As Fastest-Selling Consumer Electronics Device. **Guinness World Records**. Disponivel em: <http://community.guinnessworldrecords.com/_Kinect-Confirmed-As-Fastest-Selling-Consumer-Electronics-Device/blog/3376939/7691.html>. Acesso em: Setembro 2011.

KAWAKITA, M. et al. Real-time three-dimensional video image composition by depth information. **IEICE Electronics Express**, Agosto 2004. Vol.1, No.9, pp. 237–242.

MICROSOFT CORPORATION. C# Language Specification. **Microsoft Developer Network (MSDN)**, 2007. Disponivel em: <http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharp%20Language%20Specification.doc>. Acesso em: Outubro 2011.

MICROSOFT CORPORATION. Home. **Kinect for Windows SDK**, 2011. Disponivel em: <http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/about.aspx>. Acesso em: Setembro 2011.

OPENTK. OpenTK. **The Open Toolkit Library | OpenTK**, 2006. Disponivel em: <http://www.opentk.com/doc/chapter/2>. Acesso em: outubro 2011.

PITIÉ, F.; KOKARAM, A. Matting With a Depth Map. **Proceedings of 2010 IEEE 17th International Conference on Image Processing**, Hong Kong, 26-29 Setembro 2010. pp. 21-24.

ROE, G. D. Chroma-key Development. **Ibid**, 1965-1980. pp. 19-20.

SHOTTON, J. et al. Real-Time Human Pose Recognition in Parts from Single Depth Images. **Microsoft Research Cambridge & Xbox Incubation**, Cambridge, 2010.

7 APENDICE A - CÓDIGOS FONTE

Kinect.cs – Arquivo de comunicação com o Kinect

```
using System;
using System.IO;
using System.Diagnostics;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Drawing;
using Microsoft.Research.Kinect.Audio;
using Microsoft.Research.Kinect.Nui;

public class Kinect
{

    private Runtime kinect = Runtime.Kinects[0];
    //private Runtime kinect = new Runtime();

    private byte[] depthFrameBits = new byte[320 * 240 * 4];
    private byte[] colorFrameBits = new byte[320 * 240 * 4];

    private ImageFrame depthFrame = null;
    private ImageFrame colorFrame = null;

    public Kinect()
    {
        try
        {
            kinect.Initialize(RuntimeOptions.UseColor |
RuntimeOptions.UseDepthAndPlayerIndex);
        }
        catch (InvalidOperationException)
        {
            System.Windows.MessageBox.Show("Runtime initialization failed. Please make
sure Kinect device is plugged in.");
```

```
            System.Environment.Exit(0);
        }
        try
        {
            kinect.DepthStream.Open(ImageStreamType.Depth, 2,
ImageResolution.Resolution320x240, ImageType.DepthAndPlayerIndex);
            kinect.VideoStream.Open(ImageStreamType.Video, 2,
ImageResolution.Resolution640x480, ImageType.Color);
        }
        catch (InvalidOperationException)
        {
            System.Windows.MessageBox.Show("Failed to open stream. Please make sure to
specify a supported image type and resolution.");
            System.Environment.Exit(0);
        }
    }

    public Boolean refreshKinectFrames()
    {
        // try to get a new depth frame
        depthFrame = kinect.DepthStream.GetNextFrame(0);
        if (depthFrame == null) return false;

        // try to get a new color frame
        colorFrame = kinect.VideoStream.GetNextFrame(0);
        if (colorFrame == null) return false;

        // loop over each pixel in the depth image
        int outputIndex = 0;
        for (int depthY = 0, depthIndex = 0; depthY < depthFrame.Image.Height;
depthY++)
        {
            for (int depthX = 0; depthX < depthFrame.Image.Width; depthX++, depthIndex
+= 2)
            {
                // combine the 2 bytes of depth data representing this pixel
                short depthValue = (short)(depthFrame.Image.Bits[depthIndex] |
(depthFrame.Image.Bits[depthIndex + 1] << 8));

                // extract the id of a tracked player from the first bit of depth data
for this pixel
                int player = depthFrame.Image.Bits[depthIndex] & 7;

                // find a pixel in the color image which matches this coordinate from
the depth image
                int colorX, colorY;

kinect.NuiCamera.GetColorPixelCoordinatesFromDepthPixel(colorFrame.Resolution,
colorFrame.ViewArea, depthX, depthY, 0, out colorX, out colorY);

                // ensure that the calculated color location is within the bounds of
the image
                colorX = Math.Max(0, Math.Min(colorX, colorFrame.Image.Width - 1));
                colorY = Math.Max(0, Math.Min(colorY, colorFrame.Image.Height - 1));

                // set alpha value to a player image
                depthFrameBits[outputIndex + 0] = player > 0 ? (byte)255 : (byte)0;
                depthFrameBits[outputIndex + 1] = player > 0 ? (byte)255 : (byte)0;
                depthFrameBits[outputIndex + 2] = player > 0 ? (byte)255 : (byte)0;
                depthFrameBits[outputIndex + 3] = player > 0 ? (byte)255 : (byte)0;

                // set a real pixel color coordinate to a pixel in depth image
```

```
                colorFrameBits[outputIndex + 0] = colorFrame.Image.Bits[(4 * (colorX +
(colorY * colorFrame.Image.Width))) + 2];
                colorFrameBits[outputIndex + 1] = colorFrame.Image.Bits[(4 * (colorX +
(colorY * colorFrame.Image.Width))) + 1];
                colorFrameBits[outputIndex + 2] = colorFrame.Image.Bits[(4 * (colorX +
(colorY * colorFrame.Image.Width))) + 0];
                colorFrameBits[outputIndex + 3] = 255;

                // increment one pixel
                outputIndex += 4;
            }
        }
        return true;
    }

    public void unload()
    {
        kinect.Uninitialize();
    }

    public byte[] getTrimapOutput()
    {
        return depthFrameBits;
    }

    public byte[] getColorOutput()
    {
        return colorFrameBits;
    }
}
```

Program.cs – Arquivo de criação e execução dos shaders

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Drawing;
using System.Threading;
using TexLib;
using OpenTK;
using OpenTK.Input;
using OpenTK.Graphics;

public class TrimapShader : GameWindow
{
    private Kinect kinect = null;
    private int kinectTexture, colorTexture, renderTexture = 0;
    private int trimapProgram, medianProgram, dilationProgram, erosionProgram = 0;
    private const int textureWidth = 320, textureHeight = 240;
    private byte[] tmp = new byte[textureWidth * 4];
    private byte[] tmpcolor = new byte[textureWidth * textureHeight * 4];
    private byte[] renderTextureBits = new byte[textureWidth * textureHeight * 4];

    public TrimapShader() : base(textureWidth, textureHeight, new GraphicsMode(),
"Real-time trimaps generation") {}
```

```
    protected override void OnLoad(EventArgs e)
    {
        if (!GL.GetString(StringName.Extensions).Contains("EXT_geometry_shader4"))
        {
            System.Windows.Forms.MessageBox.Show(
                "Your video card does not support EXT_geometry_shader4. Please update
your drivers.",
                "EXT_geometry_shader4 not supported",
                System.Windows.Forms.MessageBoxButtons.OK,
System.Windows.Forms.MessageBoxIcon.Exclamation);
            Exit();
            throw new NotSupportedException();
        }

        trimapProgram = createShader("trimap");
        dilationProgram = createShader("dilation");
        medianProgram = createShader("median");
        erosionProgram = createShader("erosion");

        WindowBorder = WindowBorder.Resizable;

        kinect = new Kinect();

        GL.ClearColor(Color.Black);
        GL.Color3(Color.White);

        kinectTexture = TexUtil.CreateRGBATexture(textureWidth, textureHeight,
renderTextureBits);
        colorTexture = TexUtil.CreateRGBATexture(textureWidth, textureHeight,
renderTextureBits);

        renderTexture = GL.GenTexture();
        GL.BindTexture(TextureTarget.Texture2D, renderTexture);
        TexUtil.Upload(textureWidth, textureHeight, true, renderTextureBits);
        GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.GenerateMipmap,
1);
        GL.BindTexture(TextureTarget.Texture2D, 0);
    }

    protected override void OnRenderFrame(FrameEventArgs e)
    {
        if (kinect == null) return;
        if (!kinect.refreshKinectFrames()) return;

        TexUtil.InitTexturing();
        GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);
        GL.Color3(Color.White);

        GL.DrawBuffer(DrawBufferMode.Back);
        GL.ReadBuffer(ReadBufferMode.Back);

        GL.MatrixMode(MatrixMode.Projection);
        GL.LoadIdentity();
        Glu.Ortho2D(0.0, ClientRectangle.Width, 0.0, ClientRectangle.Height);

        GL.MatrixMode(MatrixMode.Modelview);
        GL.LoadIdentity();
        GL.Viewport(ClientRectangle);

        byte[] result = kinect.getTrimapOutput();
        GL.BindTexture(TextureTarget.Texture2D, kinectTexture);
        TexUtil.Upload(textureWidth, textureHeight, true, result);
```

```
            GL.BindTexture(TextureTarget.Texture2D, 0);

            int texture = MedianFilter(kinectTexture);
            texture = Closing(texture);
            Trimap(texture);

            texture = renderToTexture();
            GL.BindTexture(TextureTarget.Texture2D, texture);
            drawTexture();
            GL.BindTexture(TextureTarget.Texture2D, 0);

            SwapBuffers();
        }

        public int Closing(int texture)
        {
            GL.ClearColor(Color.Black);
            useShader1(dilationProgram, texture);
            texture = renderToTexture();
            GL.ClearColor(Color.Black);
            useShader1(erosionProgram, texture);
            texture = renderToTexture();
            return texture;
        }

        public int MedianFilter(int texture)
        {
            GL.ClearColor(Color.Black);
            useShader1(medianProgram, texture);
            texture = renderToTexture();
            return texture;
        }

        public void Trimap(int texture)
        {
            GL.ClearColor(Color.Black);
            useShader1(trimapProgram, texture);
        }

        int renderToTexture()
        {
            GL.BindTexture(TextureTarget.Texture2D, renderTexture);
            GL.CopyTexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, 0, 0,
    ClientRectangle.Width, ClientRectangle.Height, 0);
            GL.BindTexture(TextureTarget.Texture2D, 0);
            return renderTexture;
        }

        void useShader1(int shaderprogram, int texture)
        {
            GL.UseProgram(shaderprogram);
            {
                GL.ActiveTexture(TextureUnit.Texture0);
                GL.BindTexture(TextureTarget.Texture2D, texture);
                GL.Uniform1(GL.GetUniformLocation(shaderprogram, "texture0"), 0);
                drawTexture();
            }
            GL.UseProgram(0);
        }
        public void drawTexture()
        {
            GL.Color3(Color.White);
```

```
        GL.Begin(BeginMode.Quads);
        {
            GL.TexCoord2(0, 0); GL.Vertex2(0.0, ClientRectangle.Height);
            GL.TexCoord2(1, 0); GL.Vertex2(ClientRectangle.Width,
ClientRectangle.Height);
            GL.TexCoord2(1, 1); GL.Vertex2(ClientRectangle.Width, 0.0);
            GL.TexCoord2(0, 1); GL.Vertex2(0.0, 0.0);
        }
        GL.End();
    }

    int createShader(string name)
    {
        int program = GL.CreateProgram();
        int vert = GL.CreateShader(ShaderType.VertexShader);
        int frag = GL.CreateShader(ShaderType.FragmentShader);

        String fragSource = new StreamReader("../../Shaders/" + name +
".frag").ReadToEnd();
        String vertSource = new StreamReader("../../Shaders/default.vert"
).ReadToEnd();

        compileShader(frag, fragSource);
        compileShader(vert, vertSource);

        System.Console.WriteLine(GL.GetShaderInfoLog(frag));
        System.Console.WriteLine(GL.GetShaderInfoLog(vert));

        GL.AttachShader(program, frag);
        GL.AttachShader(program, vert);
        GL.LinkProgram(program);

        string info;
        GL.GetProgramInfoLog(program, out info);
        Debug.WriteLine(info);

        if (frag != 0) GL.DeleteShader(frag);
        if (vert != 0) GL.DeleteShader(vert);

        return program;
    }

    void compileShader(int shader, string source)
    {
        GL.ShaderSource(shader, source);
        GL.CompileShader(shader);
        string info;
        GL.GetShaderInfoLog(shader, out info);
        Debug.WriteLine(info);

        int compileResult;
        GL.GetShader(shader, ShaderParameter.CompileStatus, out compileResult);
        if (compileResult != 1)
        {
            Debug.WriteLine("Compile Error!");
            Debug.WriteLine(source);
        }
    }

    protected override void OnUnload(EventArgs e)
    {
        kinect.unload();
```

```
            base.OnUnload(e);
        }

        protected override void OnResize(EventArgs e)
        {
            base.OnResize(e);
        }

        protected override void OnUpdateFrame(FrameEventArgs e)
        {
            base.OnUpdateFrame(e);
            if (Keyboard[Key.Escape]) this.Exit();
        }

        #region public static void Main()
        [STAThread]
        public static void Main()
        {
            TrimapShader prog = new TrimapShader();
            prog.Run(30.0, 0.0);
        }
        #endregion
}
```

Dilation.frag – Shader de dilatação morfológica

```
uniform sampler2D texture0;
const float textureWidth  = 320.0;
const float textureHeight = 240.0;
const float pixelX = 1.0/320.0;
const float pixelY = 1.0/240.0;

#define WHITE 1.0

void main( void )
{
       vec4 color = texture2D(texture0, gl_TexCoord[0].xy);

       if (color.r == WHITE) {
              gl_FragColor = color;
              return;
       }

       float cor = 0.0;

       for (float i = -pixelY; i <= pixelY ; i += pixelY) {
              for (float j = -pixelX; j <= pixelX ; j += pixelX) {
                     color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX,
gl_TexCoord[0].x + j)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y + i))));
                     cor = max(cor,color.r);
              }
       }

       color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX, gl_TexCoord[0].x
- 2.0 * pixelX)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y))));
       cor = max(cor,color.r);
       color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX, gl_TexCoord[0].x
+ 2.0 * pixelX)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y))));
       cor = max(cor,color.r);
```

```
	color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX,
gl_TexCoord[0].x)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y - 2.0 * pixelY))));
	cor = max(cor,color.r);
	color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX,
gl_TexCoord[0].x)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y + 2.0 * pixelY))));
	cor = max(cor,color.r);

	gl_FragColor = vec4 (cor,cor,cor,1.0);

}
```

Erosion.frag – Shader de erosão morfológica

```
uniform sampler2D texture0;
const float textureWidth  = 320.0;
const float textureHeight = 240.0;
const float pixelX = 1.0/320.0;
const float pixelY = 1.0/240.0;

#define BLACK 0.0

void main( void )
{
	vec4 color = texture2D(texture0, gl_TexCoord[0].xy);

	if (color.r == BLACK) {
		gl_FragColor = color;
		return;
	}

	float cor = 1.0;

	for (float i = -pixelY; i <= pixelY ; i += pixelY) {
		for (float j = -pixelX; j <= pixelX ; j += pixelX) {
			color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX,
gl_TexCoord[0].x + j)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y + i))));
			cor = min(cor,color.r);
		}
	}

	color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX, gl_TexCoord[0].x
- 2.0 * pixelX)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y))));
	cor = min(cor,color.r);
	color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX, gl_TexCoord[0].x
+ 2.0 * pixelX)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y))));
	cor = min(cor,color.r);
	color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX,
gl_TexCoord[0].x)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y - 2.0 * pixelY))));
	cor = min(cor,color.r);
	color = texture2D(texture0,vec2(max(pixelX, min(1.0 - pixelX,
gl_TexCoord[0].x)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y + 2.0 * pixelY))));
	cor = min(cor,color.r);

	gl_FragColor = vec4 (cor,cor,cor,1.0);

}
```

Median.frag – Shader do filtro de mediana

```
uniform sampler2D texture0;
const float textureWidth  = 320.0;
const float textureHeight = 240.0;
const float pixelX = 1.0/320.0;
const float pixelY = 1.0/240.0;
const vec4 black = vec4(0.0,0.0,0.0,1.0);
const vec4 white = vec4(1.0,1.0,1.0,1.0);

void main( void )
{
	vec4 color = texture2D(texture0, gl_TexCoord[0].xy);

	int whiteCount = 0, blackCount = 0;

	for (float x = -1.0 ; x <= 1.0; x+=1.0) {
		for (float y = -1.0 ; y <= 1.0; y+=1.0) {

			vec2 offset = vec2(max(pixelX, min(1.0 - pixelX, gl_TexCoord[0].x
+ x*pixelX)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y + y*pixelY)));

			if (texture2D(texture0, offset).r == 1.0)
				whiteCount++;
			else
				blackCount++;

		}
	}

	if (blackCount >= whiteCount)
		gl_FragColor = black;
	else
		gl_FragColor = white;

}
```

Trimap.frag – Shader de geração do *trimap*

```
uniform sampler2D texture0;
const float textureWidth  = 320.0;
const float textureHeight = 240.0;
const float blurSize = 1.0/64.0;
const float grayColor = 0.5;
const float pixelX = 1.0/320.0;
const float pixelY = 1.0/240.0;
const int numberOfPixelsThatNeedToBeFound = 4;

void main( void )
{
	vec4 color = texture2D(texture0, gl_TexCoord[0].xy);

	int pixelsFound = 0;

	float raio;
	if (color.r == 1.0) raio = 3.0;
	else raio = 2.0;
```

```
	for (float x, y; raio > 0.0 ; raio -= 1.0) {
		for (float ang = 0.0 ; ang < 6.27 ; ang += 0.1) {

			x = (cos(ang)*raio)/textureWidth;
			y = (sin(ang)*raio)/textureHeight;

			if (texture2D(texture0, vec2(max(pixelX, min(1.0 - pixelX,
gl_TexCoord[0].x + x)), max(pixelY, min(1.0 - pixelY, gl_TexCoord[0].y + y)))).r !=
color.r)
			{
				pixelsFound++;
			}
		}
	}

	if (pixelsFound > numberOfPixelsThatNeedToBeFound) {
		gl_FragColor = vec4(grayColor, grayColor, grayColor, 1.0);
	} else {
		gl_FragColor = color;
	}

}
```