UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA PARA TRANSCRIÇÃO DO ALFABETO DATILOLÓGICO PARA TEXTO UTILIZANDO MICROSOFT KINECT

DIEGO MARCELO SANTIN

BLUMENAU
2013

2013/1 - 11

DIEGO MARCELO SANTIN

FERRAMENTA PARA TRANSCRIÇÃO DO ALFABETO DATILOLÓGICO PARA TEXTO UTILIZANDO MICROSOFT KINECT

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Aurélio Faustino Hoppe, Mestre - Orientador

BLUMENAU
2013

2013/1 - 11

FERRAMENTA PARA TRANSCRIÇÃO DO ALFABETO DATILOLÓGICO PARA TEXTO UTILIZANDO MICROSOFT KINECT

Por

DIEGO MARCELO SANTIN

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Aurélio Faustino Hoppe, Mestre – Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre - FURB
Membro:	Prof. Maurício Capobianco Lopes, Doutor - FURB

Blumenau, 10 de julho de 2013

Dedico este trabalho aos meus pais Vilson Santin e Carmen Santin, por todo apoio e amor.

AGRADECIMENTOS

Aos meus pais, Vilson Santin e Carmen Santin, por todo carinho, educação e paciência.

A minha noiva Ana Paula dos Santos Rios, por todo incentivo, compreensão e carinho.

Aos meus amigos Amando Breiter Jr e Marcel Horner por todas sugestões, conselhos e tempo dedicado.

A todos colegas de curso, pelo suporte, companheirismo e compartilhamento de conhecimento ao longo do curso.

Em especial para meu amigo Allan Camargo, pela amizade desde o começo do curso e companheirismo demonstrado durante o percurso.

Ao meu orientador, Aurélio Faustino Hoppe, por todo apoio e consideração, e principalmente por acreditar neste trabalho.

Success is the ability to go from failure to failure without losing your enthusiasm.

Sir Winston Churchill

RESUMO

Este trabalho apresenta um software para análise e transcrição dos símbolos presentes no alfabeto datilológico de forma automatizada. O grande diferencial deste trabalho é o uso do sensor Microsoft Kinect *for* Xbox 360 como ferramenta para captura de imagem. Os resultados obtidos demonstram que o software é capaz de transcrever diversos dos símbolos cadastrados, porém necessita ser aprimorado para detectar símbolos que exijam movimentação do usuário. Além disso, a detecção de símbolos cadastrados em conjunto não demonstrou um resultado satisfatório, criando mais um ponto para futuras melhorias.

Palavras-chave: Alfabeto datilológico. Kinect. Reconhecimento da mão.

ABSTRACT

This paper describes a software for analysis and transcription of the symbols present in fingerspelling in an automated manner. The great advantage of this work is the use of the Microsoft Kinect sensor for Xbox 360 as a tool for image capturing. The results show that the software is able to transcribe several of the registered symbols, but need to be enhanced to detect symbols that require movement of the user. Furthermore, the detection of symbols registered together did not show a satisfactory result, creating a point for further improvements.

Key-words: Fingerspelling. Kinect. Hand tracking.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

CAD – *Computer-Aided Design*

DSP – *Digital Signal Processing*

FPS – *Frames Per Second*

LIBRAS – LÍngua BRAsileira de Sinais

RGB – *Red Green Blue*

SDK - *Software Development Kit*

UML - *Unified Modeling Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A informação e o conhecimento são dois dos principais meios que movem o mundo. Este mundo, movido pela tecnologia, proporciona um acesso mais facilitado e aprimorado para todos, e a necessidade de novas tecnologias para pessoas com necessidades especiais vem se tornando maior a cada dia. Mas como trazer acesso a tanta informação, sendo que um dos principais meios é o som? Como pessoas com deficiência auditiva podem se privilegiar de algo comum a todos?

A preocupação com a acessibilidade para os deficientes físicos é algo em crescimento. Para auxiliar a comunicação dos deficientes auditivos, foi desenvolvida a linguagem de sinais e o alfabeto manual, que utiliza os movimentos das mãos, braços e da face para simbolizar palavras e expressões (CUORE, 2009).

No Brasil, essa linguagem foi adaptada dando origem a Língua Brasileira de Sinais, mais conhecida como LIBRAS. Atualmente, a única forma de ensino dessa linguagem é realizada através de aulas presenciais, apostilas ou vídeos. Não existe uma forma dinâmica de ensino sem que haja outra pessoa envolvida (MONTEIRO, 2006).

Um dos meios para alcançar este objetivo é tentar de alguma forma interpretar o que a pessoa tenta dizer quando utiliza a linguagem dos sinais, e para isso se utiliza o reconhecimento de gestos. Quando ela utilizar do alfabeto datilológico para comunicar-se, os gestos podem ser capturados com algum tipo de câmera e trabalhados almejando transcrever o que a pessoa quis dizer.

Entretanto, existem diversos problemas nesta abordagem. Um dos mais discutidos é a influência da luz que altera drasticamente a interpretação da imagem trabalhada criando um desafio muito grande para câmeras simples (LOCKTON; FITZGIBBON, 2002). Para contornar este desafio será utilizado o sensor de movimentos Microsoft Kinect, que já possui todo tratamento na intensidade da luz, mapeamento do esqueleto humano e diversas outras funções que prometem aumentar muito a qualidade do reconhecimento de gestos.

Diante do exposto, propõe-se disponibilizar um software para interpretar a linguagem de sinais, mais especificamente o alfabeto datilológico, através da captura dos movimentos das mãos do usuário, sendo utilizados os algoritmos de Ramer-Douglas-Peucker, Bresenham e algoritmo da tartaruga para reconhecer os gestos.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um interpretador de sinais que transcreva o alfabeto datilológico para texto utilizando o Microsoft Kinect como mecanismo de captura.

Os objetivos específicos do trabalho são:

a) capturar o movimento das mãos do usuário utilizando os sensores do Kinect;
b) reconhecer os gestos baseando-se em um banco pré-definido com o alfabeto datilológico;
c) transcrever o movimento realizado para a correspondência em texto.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está subdivido em capítulos que serão abordados a seguir.

O segundo capítulo apresenta a fundamentação teórica do trabalho, introduzindo os conceitos de LIBRAS, métodos para transcrição de gestos, funcionamento do Microsoft Kinect e trabalhos correlatos.

O terceiro capítulo trata sobre a especificação do software, apresentando diagramas das classes e estruturas utilizadas, e a implementação, demonstrando algoritmos utilizados e abordagens criadas para solução do problema.

As conclusões são expostas no quarto capítulo, onde sugestões para trabalhos futuros também podem ser encontradas.

2 FUNDAMENTAÇÃO TEÓRICA

Nas próximas seções é detalhado o funcionamento do Kinect, métodos para detecção das mãos e transcrição do gesto reconhecido. Na seção 2.1 é abordada a história da língua de sinais no Brasil (LIBRAS) e suas características; o funcionamento detalhado do Kinect e a SDK serão abordados na seção 2.2. Já a seção 2.3 apresenta o algoritmo de Ramer-Douglas-Peucker, utilizado no desenvolvimento do software. Por fim, a seção 2.4 apresenta quatro trabalhos relacionados.

2.1 LÍNGUA DE SINAIS

As primeiras metodologias voltadas ao desenvolvimento e a comunicação dos deficientes auditivos surgiram na Europa por volta do século XVI. Nesta época, surgiram as primeiras representações através de gestos do alfabeto, dando origem à linguagem de sinais (OLIVEIRA, 2011).

O instituto Nacional de Jovens Surdos, localizado em Paris na França, iniciou a aplicação da metodologia desenvolvida por Charles Michel de I'Epée, através da datilografia, alfabeto manual e gestos desenvolvidos por ele. Através desses métodos outros países passaram a desenvolver suas próprias metodologias para auxiliar na comunicação entre surdos e ouvintes (RAMOS, 2006).

A língua de sinais auxilia o deficiente auditivo no seu desenvolvimento e comunicação com outros deficientes e com os ouvintes. Ela não é baseada somente no alfabeto, mas também em expressões e gestos, que podem possuir diversas representações. Atualmente cada país possui uma língua de sinais particular, com movimentos significativos para cada expressão, ou seja, essa linguagem não é universal, nem padronizada, cada palavra pode ser interpretada de forma independente variando conforme o país. As palavras e sensações também podem ser expressas de formas distintas, dependendo da região e devido à necessidade de comunicação. (PACHECO; ESTRUC, 2008).

2.1.1 LIBRAS: Língua Brasileira de Sinais

No Brasil, a língua de sinais teve seu início por volta de 1856 através do Professor francês Ernest Huet. Sendo portador de deficiência auditiva, trouxe ao Brasil o alfabeto gestual francês, e através desse alfabeto a língua brasileira de sinais começou a se desenvolver (Figura 1). Atualmente, a linguagem brasileira de sinais é conhecida como LIBRAS (RAMOS, 2006).

Figura 1 - Alfabeto em LIBRAS (Língua de Sinais Brasileira)

Fonte: Araújo (2007).

O processo até a língua de sinais tornar-se de fato reconhecida passou por diversas dificuldades. Naquela época qualquer tipo de gesto era considerado algo fora dos padrões. Muitos pais tinham vergonha de ter filhos surdos, não permitindo sua inclusão na sociedade, e os próprios deficientes se sentiam desestimulados a se comunicar. Com o passar dos anos, a comunicação foi ganhando real importância e o deficiente auditivo passou a ser reconhecido na sociedade como um indivíduo. A primeira escola para surdos no Brasil foi fundada em 1857, Instituto Nacional da Educação de Surdos (INES), no estado do Rio de Janeiro. Esse Instituto abrigava deficientes auditivos de todas regiões brasileiras (MONTEIRO, 2006).

A comunicação através de sinais no Brasil passou por diversas dificuldades, e chegou a ser proibida, sendo praticada informalmente nas escolas. Somente em 24 de abril de 2002, foi sancionada a Lei nº 10.436 que regulamentaria a Língua Brasileira de Sinais, e após esse período, ela sofreu diversas alterações. Os deficientes auditivos vêm ganhando espaço na sociedade, mas ainda existe uma grande necessidade de modificações para que a comunicação entre ouvintes e surdos seja plena. Essa linguagem é desenvolvida através de gestos realizados por movimentos das mãos e expressões faciais utilizados como forma de comunicação, sendo extremamente complexa pois utiliza o próprio corpo e seus movimentos da forma mais expressiva possível, demonstrando atitudes e sentimentos dos mais diversos (RAMOS, 2006).

Em 22 de dezembro de 2005, pelo decreto nº 5.626, foi regulamentada a Lei que estabelece a LIBRAS como disciplina curricular de caráter obrigatório na formação de professores (CUORE, 2009).

A Língua Brasileira de Sinais possui uma estrutura gramatical independente, sendo um

sistema linguístico legítimo que oferece ao surdo a integração com a sociedade, o que demonstra que essa linguagem possui uma grande semelhança com a língua oral (OLIVEIRA, 2011).

2.2 KINECT

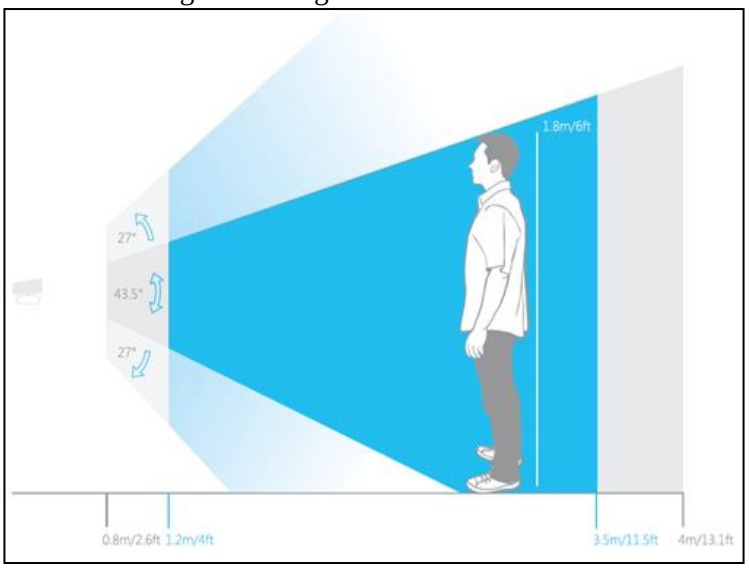
Conforme Microsoft (2012d), o Kinect é um dispositivo que retorna informações do ambiente real, utilizando: um projetor infravermelho, uma câmera infravermelha, uma câmera de vídeo e quatro microfones. Para auxiliar a captura de dados ele também tem um motor de inclinação, e através do Kinect *for* Windows SDK é possível acessar as informações coletadas.

A seguir é detalhado o funcionamento dos componentes do Kinect: vídeo, motor de inclinação, profundidade e Kinect *for* Windows SDK.

2.2.1 Vídeo e motor de inclinação

Conforme Microsoft (2012b), Kinect possui limitações de ângulo de visão que ele consegue coletar informações. O ângulo de visão é de 57,5º na horizontal e 43,5º na vertical. O motor de inclinação pode movimentar o ângulo de visão vertical em até 27º para cima ou para baixo. A Figura 2 representa os ângulos de visão, sendo o feixe em azul o ângulo de visão vertical.

Figura 2 - Ângulo de visão do Kinect



Fonte: Microsoft (2012f).

O Kinect possui uma câmera de vídeo para capturar imagens. Essa é comparada à uma *webcam* comum já que ela não possui nenhum recurso especial ou uma resolução superior a *webcams.* A resolução máxima é de 1280x1024, mas normalmente é utilizada em 640x480 devido à quantidade de *Frames Per Second* (FPS). Na resolução de 1280x1024 são 12 FPS e

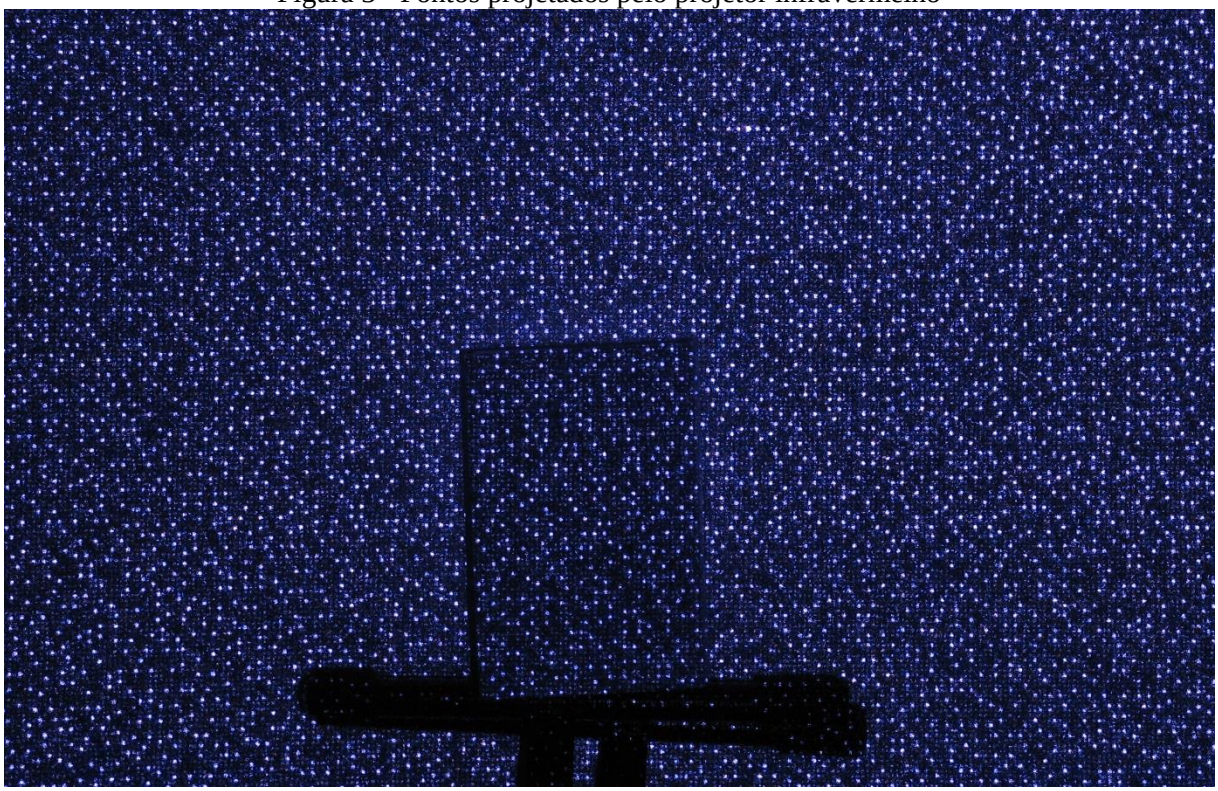
em 640x480 são 30 FPS (MILES, 2012, p. 5-6).

2.2.2 Profundidade

O Kinect foi desenvolvido pela Microsoft em parceria com a PrimeSense, uma empresa israelense que produz câmeras que utilizam métodos para detectar a profundidade na cena, baseando-se em luz estruturada, para retornar mapas de profundidade (QUEIROZ, 2010).

Para usar o método de luz estruturada o Kinect possui dois elementos. O primeiro é um projetor infravermelho que projeta vários pontos no ambiente a sua frente (Figura 3). O segundo é uma câmera infravermelha que detecta estes pontos e conforme a distância entre eles calcula a distância desta área até o sensor (MILES, 2012, p. 7).

Figura 3 - Pontos projetados pelo projetor infravermelho



Fonte: Microsoft (2012d).

A distância que o Kinect retorna informações de profundidade é de acordo com a aplicação, podendo essa escolher entre dois modos: distância normal e modo curta distância (Figura 4). O Kinect vai retornar dados de profundidade que estiverem além desses limites, mas serão de baixa qualidade e precisão (MICROSOFT, 2012a).

A Figura 4 apresenta os limites em metros que o Kinect retorna informações de profundidade, sendo a primeira barra horizontal do modo normal e a segunda do modo curta distância. As cores possuem a seguinte notação:

a) marrom: indica que a distância do objeto até o Kinect é desconhecida, os valores

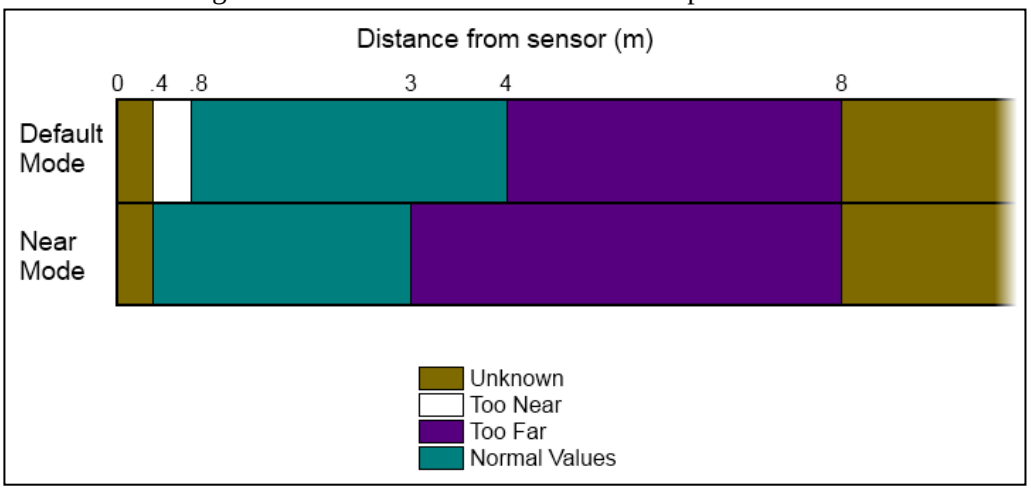
da distância estará incorreto;

b) branca: indica que o objeto está muito perto do Kinect, os valores de distância estarão incorretos;

c) roxo: indica que o objeto está muito longe do Kinect, os valores de distância serão de baixa precisão;

d) ciano: indica que os valores da distância terão uma boa precisão.

Figura 4 - Distância de coleta de dados de profundidade



Fonte: Microsoft (2012d).

O Kinect *for* Windows SDK foi lançada em junho de 2011 pela Microsoft como versão beta. Em fevereiro de 2012 foi lançada a primeira versão oficial do SDK, permitindo a utilização de mais de um Kinect em uma aplicação. Também foi melhorada a detecção de esqueleto e adicionado o modo de curta distância (EISLER, 2012).

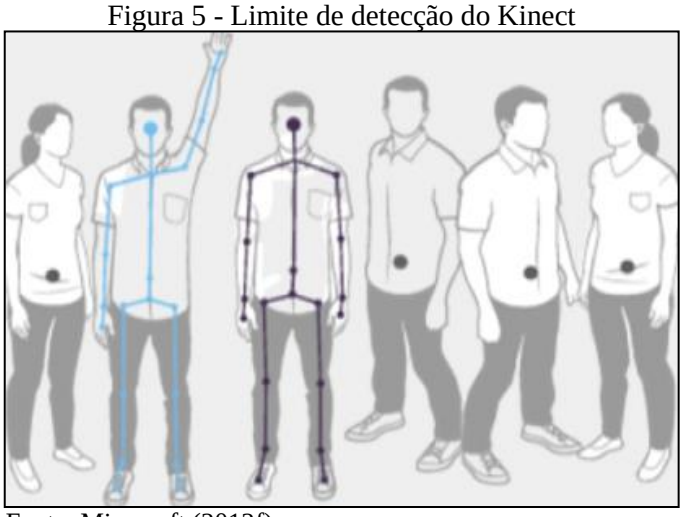
A aplicação que utilizar o Kinect *for* Windows SDK pode ser desenvolvida nas linguagens de programação C#, VB.NET e C++ (MICROSOFT, 2012e).

O Kinect *for* Windows SDK contém duas *Application Programming Interface* (API): a *Natural User Interface*[1](NUI) API e a áudio API. Estas API permitem acesso aos *streams* de áudio, imagem e profundidade do Kinect (MICROSOFT, 2012c). Além de acesso aos *streams*, as API permitem acessar o motor de inclinação e ter acesso a dados já processados.

Skeletal tracking é um desses dados já processados pelo SDK que detecta até seis pessoas em frente ao Kinect, e retorna o esqueleto, que são as coordenadas de 20 ou 10 *joints*, de até duas pessoas dentre as seis destacadas (Figura 5).

[1] Interface de usuário projetada para utilizar comportamentos para interagir diretamente com o conteúdo digital.

Figura 5 - Limite de detecção do Kinect



Fonte: Microsoft (2012f).

Além do *Skeletal tracking*, o *stream* de profundidade pode ser utilizado em conjunto com outros *streams* pela possibilidade de sincronizar a captura dos mesmos. Juntamente com a sincronização dos *streams*, o motor de inclinação pode ser utilizado para movimentar verticalmente o sensor, buscando posicionar as câmeras de uma forma que auxilie a detecção das pessoas.

2.3 ALGORTIMO DE RAMER-DOUGLAS-PEUCKER

O algoritmo Ramer-Douglas-Peucker é comumente utilizado para simplificar segmentos de reta, criado por Urs Ramer, David Douglas e Thomas Peucker (RAMER, 1972). Este algoritmo pode ser reconhecido por diversos nomes: “o algoritmo de Douglas-Peucker”, “algoritmo iterativo de ajuste de ponto final” e “algoritmo de divisão e união” (DOUGLAS; PEUCKER, 1973).

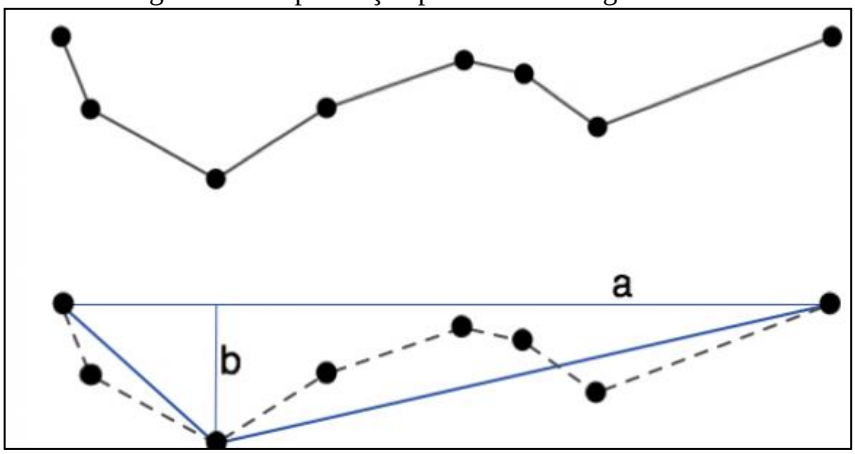
O funcionamento do algoritmo procede da seguinte forma: o primeiro ponto da linha é definido como a "âncora" e o último ponto como um "flutuador". Estes dois pontos estão ligados por um segmento de reta e distâncias perpendiculares, onde a partir deste segmento todos os pontos intermediários são calculados. Se nenhuma destas distâncias perpendiculares exceder uma tolerância especificada, o segmento de reta é considerado adequado para representar toda a linha de forma simplificada.

Se esta condição não for satisfeita, então o ponto de maior distância do segmento de reta perpendicular é selecionado, tornando-se o novo ponto flutuante. O ciclo é repetido e o novo segmento de reta é então definido pela âncora e o novo ponto flutuador. As distâncias para os pontos intermediários são então recalculados, perpendiculares a este novo segmento. Este processo continua até que os critérios de tolerância sejam atendidos. Uma vez que os critérios de tolerância forem alcançados, o ponto âncora torna-se o ponto flutuador mais

recentemente selecionado, e o novo ponto flutuante é selecionado a partir do topo da pilha de flutuadores.

Na Figura 6 é possível visualizar um segmento de reta simplificado pelo algoritmo de Ramer-Douglas-Peucker.

Figura 6 - Simplificação por Ramer-Douglas-Peucker



Fonte: Ramer (1972).

O processo de seleção é repetido até que, eventualmente, o ponto âncora atinja o último ponto da linha e o processo de simplificação está completo. Pontos previamente designados como âncoras são ligados por segmentos de reta de modo a formar a linha simplificada (WHYATT; WADE, 1988).

2.4 TRABALHOS CORRELATOS

A seguir serão apresentadas quatro ferramentas relacionadas ao ensino e transcrição de LIBRAS. Na seção 2.4.1, um protótipo de software para ensino de LIBRAS, chamado SignWriting (SOUZA; PINTO, 2003). Na seção 2.4.2, 2.4.3 e 2.4.4, três projetos que utilizam Kinect, sendo Kinect SDK Dynamic Time Warping Gesture Recognition (RHEMYST; RYMIX, 2011), KinectFusion (IZADI et al., 2011) e KinectCAD (LÜBKE, 2012) respectivamente.

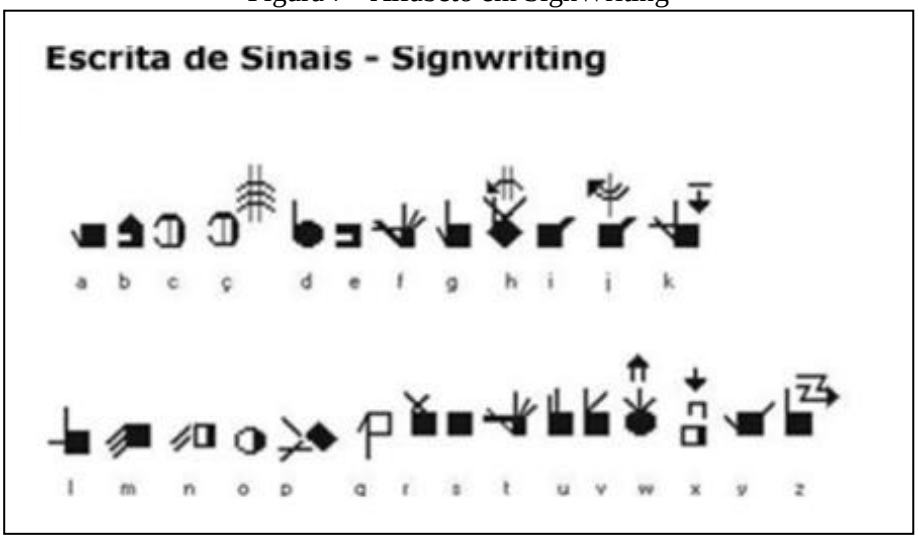
2.4.1 SignWriting

A ferramenta *SignWriting* foi desenvolvida com o objetivo de utilizar a linguagem de sinais através de representações escrita-visual. Esta ferramenta, que utiliza símbolos visuais, foi desenvolvida em 1974 por Valeerie Sutton na Califórnia, para auxílio na alfabetização de deficientes auditivos (SOUZA; PINTO, 2003).

Para utilizar a escrita gráfica *SignWriting* como mostra a Figura 7, foi desenvolvido o software *Sign Webmessage* com o intuito de tornar mais simples a comunicação entre os surdos e ouvintes devido à interação entre ambas línguas, portuguesa e Libras (SOUZA;

PINTO, 2003).

Figura 7 - Alfabeto em SignWriting

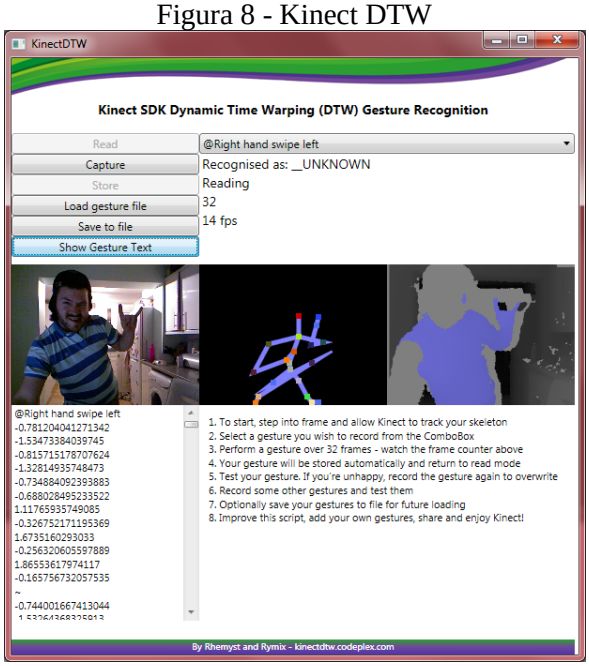


Fonte: Souza e Pinto (2003).

2.4.2 Kinect SDK *Dynamic Time Warping*

"Kinect SDK Dynamic Time Warping Gesture Recognition" (RHEMYST; RYMIX, 2011) é uma aplicação que grava movimentos de uma pessoa para seu posterior reconhecimento (Figura 8).

Figura 8 - Kinect DTW



Fonte: Rhemyst e Rymix (2011).

Para gravar os gestos, a aplicação utiliza os *joints* que formam o esqueleto e grava suas respectivas coordenadas. Para o posterior reconhecimento do gesto, é realizada a comparação do movimento gerado com as coordenadas gravadas anteriormente através de um algoritmo de verificação de distância entre pontos. O usuário pode gravar as coordenadas em um arquivo texto e importar o arquivo posteriormente. Como a aplicação só grava as coordenadas

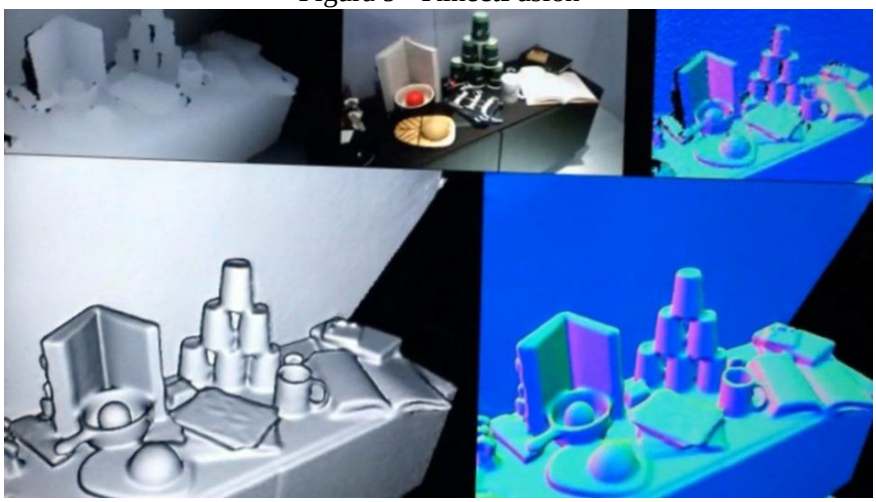
x e y, descartando a coordenada z, ela não reconhece movimentos tridimensionais.

2.4.3 KinectFusion

KinectFusion (IZADI et al., 2011) é uma aplicação que permite a reconstrução de objetos e ambientes através da captura de dados da câmera de profundidade do Kinect. Para executar o rastreamento da cena reconstruída, apenas os dados do sensor de profundidade do Kinect são utilizados, dando precisão geométrica e uma renderização em tempo real.

Na Figura 9, é perceptível a semelhança do objeto real com o objeto reconstruído, mas a reconstrução não é perfeita. É possível notar que os detalhes internos do objeto contêm imperfeições na reconstrução das superfícies.

Figura 9 - KinectFusion



Fonte: Izadi et al. (2011).

KinectFusion também simula física no ambiente captado. Na Figura 10 é demonstrada através de bolinhas amarelas a colisão com objetos na cena, como sofás, mesas, entre outros.

Figura 10 - Simulação de física

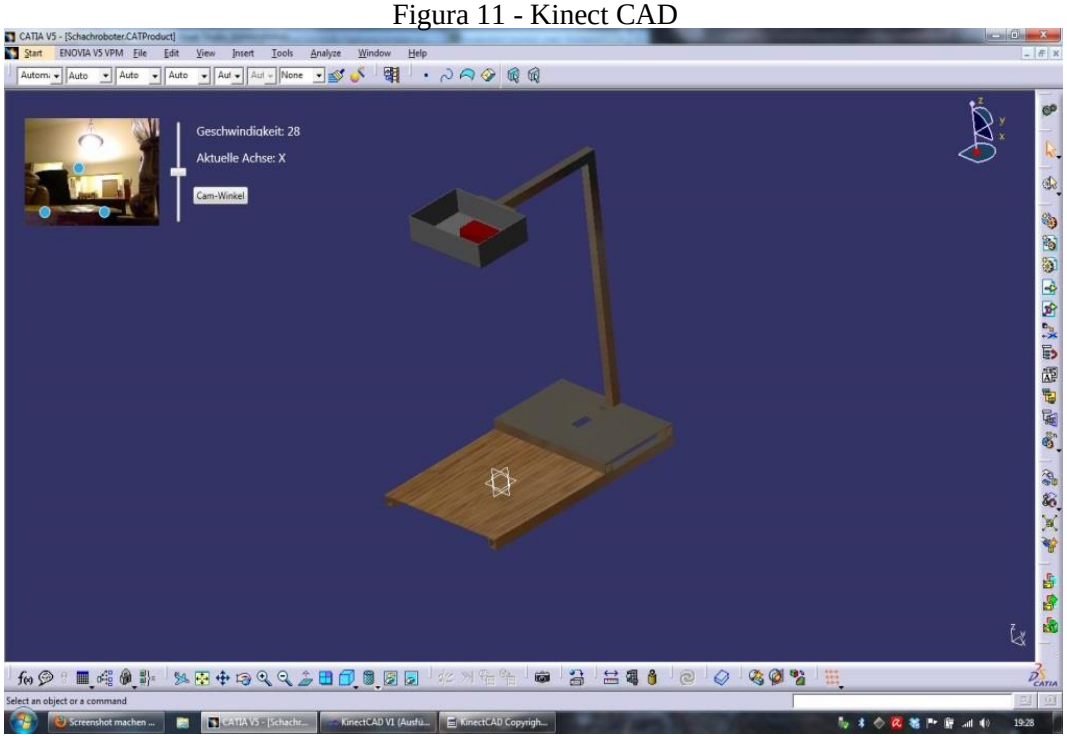
Fonte: Izadi et al. (2011).

2.4.4 KinectCAD

Existem diversas tecnologias utilizadas para manipulação de objetos em softwares CAD, sejam elas ambientes virtuais, óculos ou até mesmo luvas. Com o advento do Kinect, é possível manipular estes mesmos objetos utilizando as próprias mãos (DUNCAN, 2012).

KinectCAD (LÜBKE, 2012) é uma aplicação que permite a manipulação de objetos no sistema *Computer Aided Three-dimensional Interactive Application* (CATIA) (Figura 11) através do esqueleto gerado pelo SDK. Dentro dela é possível executar diversas ações como rotacionar, mover e dar zoom em objetos. Além disso, é possível utilizar recursos de reconhecimento de voz para alterar configurações da aplicação.

Figura 11 - Kinect CAD



Fonte: Lübke (2012).

2.4.5 Comparação dos trabalhos

É possível analisar o trabalho SignWriting (SOUZA; PINTO, 2003) como protótipo para ensino da língua de sinais, que busca representar visualmente o alfabeto através de símbolos, com o intuito de auxiliar o aprendizado da língua de sinais para pessoas com deficiência auditiva.

Além do protótipo acima mencionado, os softwares Kinect SDK Dynamic Time Warping Gesture Recognition (RHEMYST; RYMIX, 2011), KinectFusion (IZADI et al., 2011) e KinectCAD (LÜBKE, 2012) demonstram o uso do sensor Kinect como ferramenta para captura de movimentos possibilitando o uso em diversos ambientes, como reconstrução de superfícies, transcrição de gestos e manipulação de imagens e documentos.

Kinect SDK Dynamic Time Warping Gesture Recognition (RHEMYST; RYMIX, 2011), introduz a transcrição de movimentos dando foco em gestos mais amplos, como movimentos de braços e outros membros, visando a execução de comandos pré-selecionados. Por outro lado, o software KinectCAD (LÜBKE, 2012) captura pequenos gestos executados

pelas mãos do usuário, para manipular imagens e objetos em diferentes softwares existentes. Por fim, o software KinectFusion (IZADI et al., 2011) possibilita visualizar a reconstrução da cena capturada pelo sensor Kinect, disponibilizando um meio para transcrever o ambiente real no qual o usuário está inserido.

O Quadro 1 mostra as principais características dos trabalhos correlatos baseando-se em critérios extraídos no decorrer desta seção.

Quadro 1 - Comparação dos trabalhos correlatos

características	RHEMYST e RYMIX (2011)	IZADI et al. (2011)	LÜBKE (2012)
utiliza *skeletal tracking*	X		X
utiliza *stream de vídeo*	X	X	X
utiliza *stream de profundidade*		X	
utiliza Kinect *for* Windows SDK	X	X	X
faz reconhecimento de gestos	X		X
principal funcionalidade utilizada da Kinect *for* Windows SDK	*skeletal tracking*	*stream* de profundidade	*stream* de profundidade

Os trabalhos acima analisados demonstram a utilidade da captura dos movimentos executados por um ou mais usuários. Além disso, fica evidente a importância da transcrição dos gestos como ferramenta para inclusão e aprimoramento da qualidade de vida, acesso e participação do público alvo.

3 DESENVOLVIMENTO DO SOFTWARE

Neste capítulo são abordadas as etapas envolvidas no desenvolvimento do software proposto. Na seção 3.1 são apresentados os requisitos principais. A seção 3.2 descreve a especificação. A seção 3.3 apresenta a implementação e, por fim, na seção 3.4 são listados os resultados, discussões e sugestões de melhorias.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O software de reconhecimento de gestos proposto tem como requisitos:

a) permitir ao usuário informar qualquer letra e/o número contidos no alfabeto datilológico (Requisito Funcional – RF);
b) transcrever a entrada do usuário para texto na tela do computador (RF);
c) utilizar um arquivo xml contendo dados sobre as formas geométricas para comparações com os sinais de entrada (RF);
d) ser implementado utilizando o ambiente de desenvolvimento Microsoft Visual Studio (Requisito Não-Funcional – RNF);
e) ser implementado utilizando a linguagem de programação C# (RNF);
f) ser compatível com sistemas operacionais que ofereçam suporte ao .NET Framework (RNF);
g) utilizar o sensor de movimentos Microsoft Kinect (RNF);
h) utilizar o Kinect *for* Windows SDK (RNF).

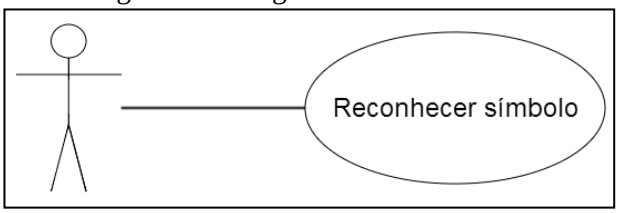
3.2 ESPECIFICAÇÃO

A especificação do software foi desenvolvida seguindo a análise orientada a objetos, utilizando a *Unified Modeling Language* (UML) em conjunto com a ferramenta Visual Studio 2012, responsável pela modelagem dos diagramas de classes. Para geração do diagrama de caso de uso e de atividade, foi utilizada a ferramenta Enterprise Architect 9.3.930.

3.2.1 Casos de uso

Nesta seção será descrito o caso de uso do software, ilustrado na Figura 12. Identificou-se apenas um autor, denominado `Usuário`, o qual utiliza todas as funcionalidades do software.

Figura 12 - Diagrama de caso de uso



O caso de uso descreve a relação entre o usuário e a funcionalidade que permite o reconhecimento de símbolos. Detalhes deste caso de uso estão descritos no Quadro 2.

Quadro 2 - Caso de uso `UC01 – reconhecer símbolo`

Número	01
Caso de Uso	Reconhecer símbolo
Descrição	Este caso de uso tem por objetivo reconhecer o movimento do usuário para traduzir em um símbolo presente no alfabeto datilológico.
Ator	`Usuário`
Pré-condições	Estar com o software aberto
Cenário Principal	1. O `Usuário` executa o movimento desejado; 2. O software identifica o movimento, traduzindo para um símbolo do alfabeto datilológico; 3. O símbolo reconhecido é demonstrado visualmente ao `Usuário`.
Fluxo Alternativo	4. No passo 2, caso o software não reconheça o movimento, um sinal de ‘?’ (interrogação) será demonstrado visualmente ao `Usuário`.

3.2.2 Diagramas de classes

Nesta seção são descritas as estrutura e classes do software desenvolvido. Os diagramas de classe foram definidos conforme os módulos implementados, sendo a seção 3.2.2.1 contendo os diagramas das classes e estruturas de captura dos dados vindos do Kinect, posteriormente utilizadas para armazenar os dados dos símbolos, e a seção 3.2.2.2 contendo os diagramas da parte do processamento do software.

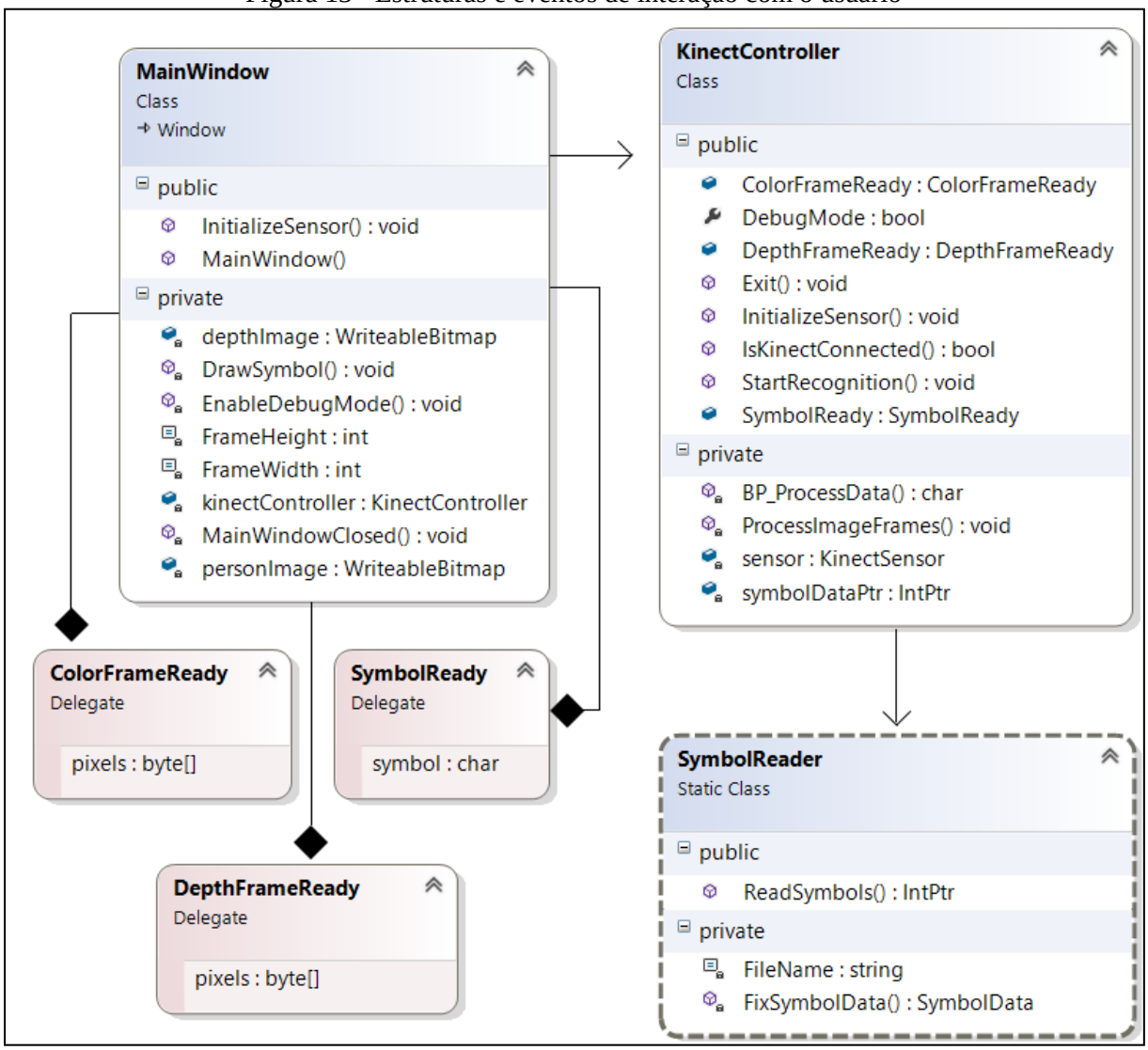
3.2.3 Estruturas e classes utilizadas para armazenar dados dos símbolos

Nas seções a seguir, 3.2.2.1.1 e 3.2.2.1.2 respectivamente, estão descritas as classes e estruturas que servirão para capturar os dados do Kinect, e as classes e estruturas utilizadas para armazenar os dados pertinentes aos símbolos.

3.2.4 Diagrama das classes de interação com o usuário

As classes e eventos responsáveis pela interação com o usuário possuem a responsabilidade de receber os dados vindos do Kinect, enviá-los para a parte de processamento e por fim, exibir na tela o resultado obtido (Figura 13).

Figura 13 - Estruturas e eventos de interação com o usuário



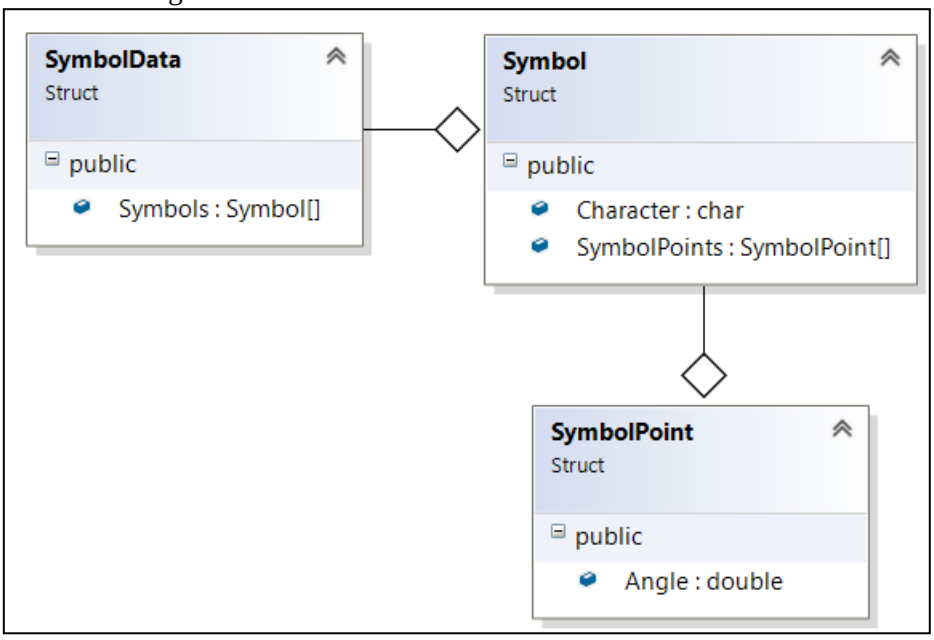
A classe `MainWindow` é responsável por exibir os dados processados, mostrando um feedback visual da captura dos movimentos do usuário. Além disso, ela demonstra se o usuário está em uma distância aceitável do sensor Kinect. A classe `SymbolReader` executa a leitura do arquivo `XML` contendo os dados para reconhecimento de um símbolo.

Já a classe `KinectController` é responsável por ligar e desligar o sensor, buscar o usuário, regulando o ângulo de visão do sensor, e executar a chamada para o processamento dos dados capturados, utilizando os dados vindos do Kinect. Além destas 3 classes, existem 3 eventos: `ColorFrameReady`, `SymbolReady` e `DepthFrameReady` que são disparados quando o processamento do feedback visual for executado, o símbolo for detectado e a distância entre o usuário e o sensor forem calculadas, respectivamente.

3.2.5 Estruturas de armazenamento

As estruturas `SymbolData`, `Symbol` e `SymbolPoint` (Figura 14) são utilizadas para enviar os dados obtidos através da leitura do arquivo XML para o processamento do software.

Figura 14 - Estruturas de armazenamento dos símbolos



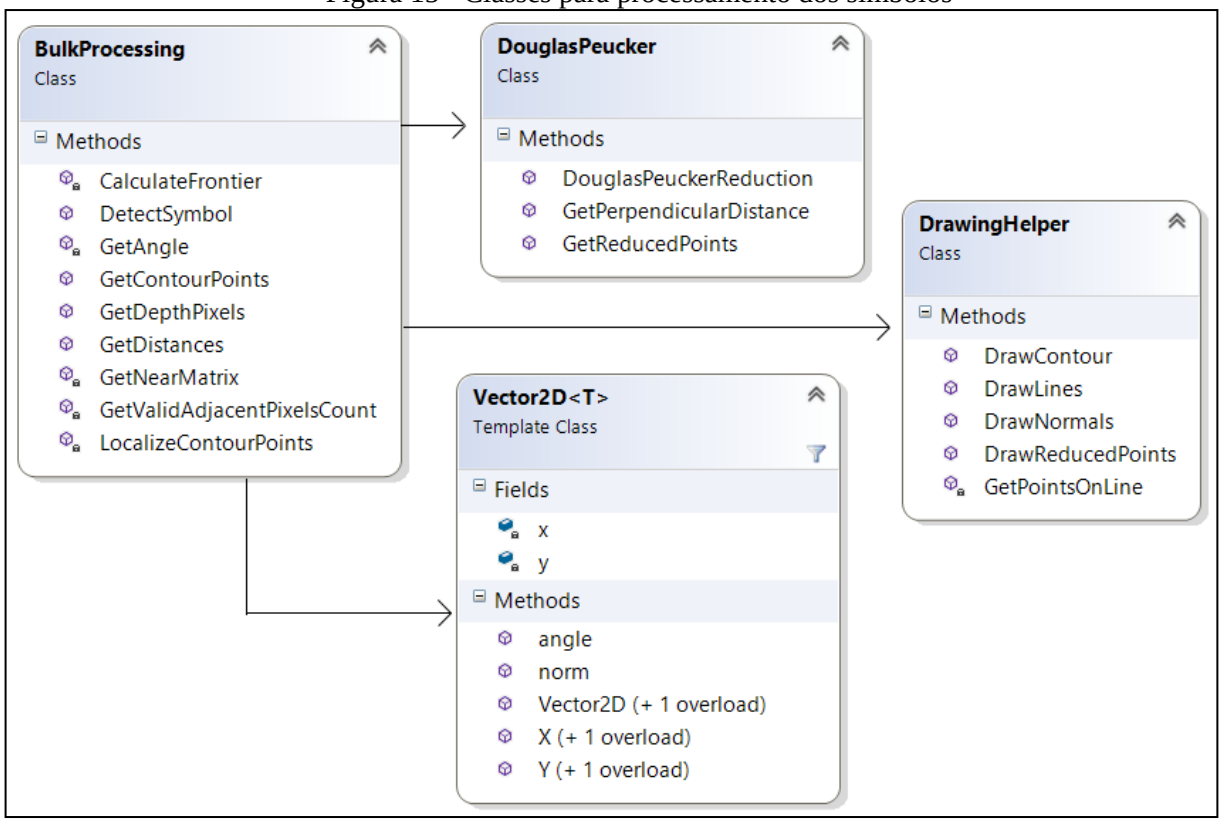
A estrutura `SymbolPoint` possui apenas o valor: `Angle`, que é utilizado para demarcar o ângulo relacionado à um ponto específico do contorno obtido. A estrutura `Symbol` possui uma lista de `SymbolPoints` e um caractere que está diretamente relacionado com a lista de pontos. Já a estrutura `SymbolData` possui uma lista de `Symbol`, e é utilizada como parte dos dados entrada para o processamento.

3.2.6 Diagramas de classes para processamento do software

As classes `BulkProcessing`, `DouglasPeucker`, `DrawingHelper` e `Vector2D` (Figura 15) representam, respectivamente, o principal ponto de processamento do software, o algoritmo para definição da forma geométrica gerada pela interação do usuário, a escrita dos pixels calculados para exibição, e a estrutura utilizada para guardar dados de pontos e posterior cálculo dos ângulos.

A classe `BulkProcessing` possui os métodos necessários para calcular as distâncias, matriz de profundidade, pontos de contorno da mão do usuário e o processamento dos ângulos juntamente com a classe Vector2D, que se torna responsável por guardar os valores X e Y de um ponto e executar os cálculos necessários para obtenção dos ângulos utilizados.

Figura 15 - Classes para processamento dos símbolos

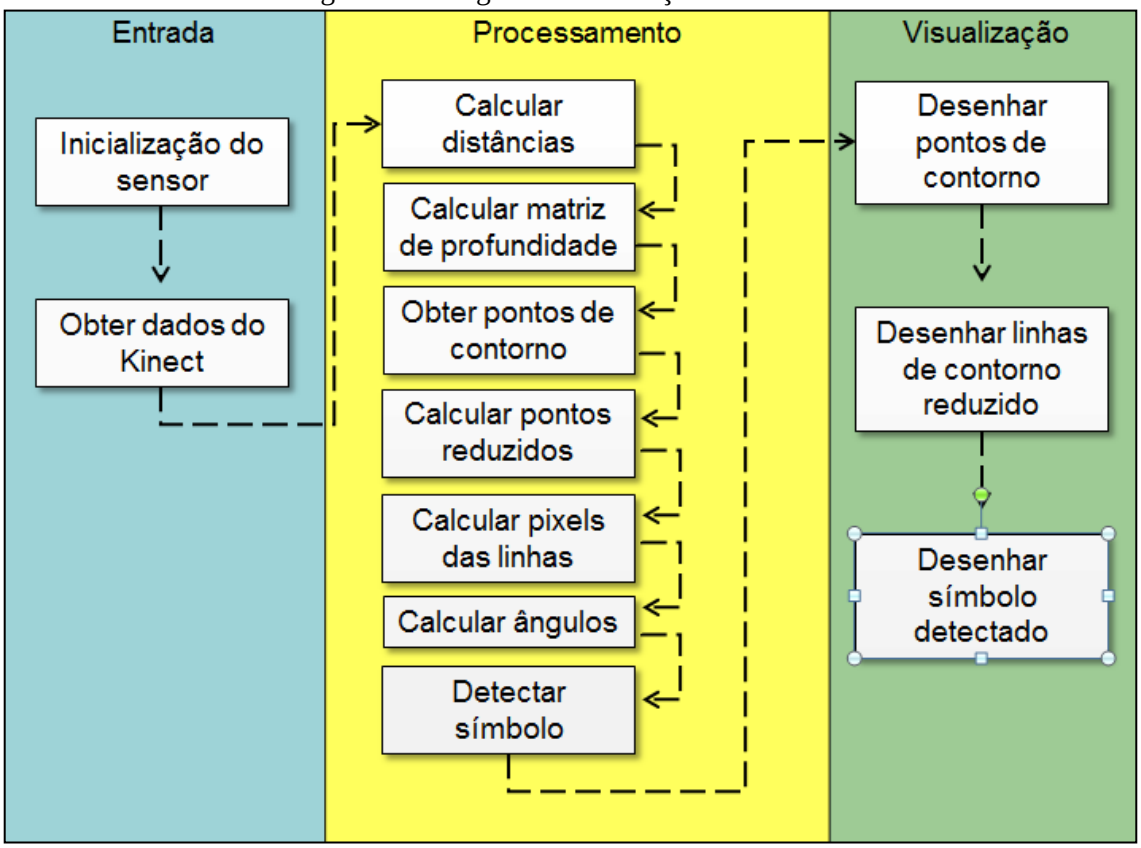


A classe `DouglasPeucker` é, unicamente, utilizada para executar a redução de pontos em uma curva que no caso deste software, é demonstrada pelo contorno da mão do usuário. E por fim, a classe `DrawingHelper` é utilizada para atribuir valores de cor aos pixels recebidos pelo Kinect, demonstrando o feedback necessário ao usuário.

3.2.7 Diagrama de atividade

O diagrama de atividades é responsável por representar os estados de uma computação. A Figura 16 apresenta os passos sequenciais para obtenção da imagem do usuário e processamento da mesma, tendo como resultado um caractere que representa o símbolo reconhecido.

Figura 16 - Diagrama de execução do software



Na Figura 16, pode-se observar que a primeira operação realizada é inicializar o sensor. Como o Kinect possui mais de uma forma para utilização de suas câmeras, é necessário habilitar, via código, quais serão utilizadas. Após isso, é executada uma cópia dos dados obtidos do Kinect e então, os mesmo são enviados para processamento.

Com os dados recebidos, são calculadas as distâncias que serão utilizadas para calcular a matriz de profundidade. Esta matriz será utilizada para determinar a viabilidade da execução do cálculo do contorno da mão do usuário, avaliando a distância entre o sensor e a mesma.

Baseando-se nesta matriz de profundidade, os pontos de contorno são armazenados e desenhados na tela para demonstrar a área capturada pelo software. Após isto, é executado o algoritmo de Ramer-Douglas-Peucker, descrito na seção 3.3.1.3, para reduzir a quantidade de ponto, restringindo a área para obtenção da forma geométrica que será trabalhada posteriormente. Com os pontos reduzidos devidamente calculados, é feita uma nova passagem na interação com o usuário, desenhando as linhas e pontos que demarcam a forma obtida.

Por fim, são executadas diversas iterações sobre os pontos reduzidos, a fim de calcular os ângulos formados entre os mesmos obtendo assim, um meio para detectar o símbolo relacionado à intenção do usuário. Após a detecção, um novo *feedback* é demonstrado em tela, desenhando o símbolo detectado pelo software.

3.3 IMPLEMENTAÇÃO

Nesta seção são descritas as técnicas e ferramentas utilizadas e a operacionalidade da implementação. Na seção 3.3.1 e subseções são apresentadas as técnicas utilizadas para desenvolvimento do software.

3.3.1 Técnicas e ferramentas utilizadas

A estrutura do software desenvolvido foi separada em dois módulos: interface e algoritmos de processamento (cálculos de profundidade, distâncias, pontos de contorno, algoritmos de redução e cálculo de ângulos). O módulo de interface foi desenvolvido utilizando a linguagem de programação C#, com auxílio da ferramenta Visual Studio 2012. Neste módulo, as seguintes bibliotecas foram utilizadas:

a) Microsoft.Kinect;
b) Windows.Media;
c) Xml.Serialization;
d) Runtime.InteropServices.

O módulo de processamento foi desenvolvido na linguagem de programação C++, com o auxílio da ferramenta Visual Studio 2012. Neste módulo, as seguintes bibliotecas foram utilizadas:

a) math.h;
b) algorithm.

No desenvolvimento, foi utilizado um sensor Microsoft Kinect *for* Xbox 360.

As seções a seguir demonstram a implementação da interface e do algoritmo de processamento, bem como suas principais características e amostras de código fonte.

3.3.2 Inicialização e leitura dos dados do sensor

O processamento para detecção de um símbolo está baseado nos dados provenientes do sensor Kinect. Para que seja possível trabalhar com estes dados, é necessário executar a inicialização do sensor para posteriormente obter os dados capturados por suas câmeras.

Com o sensor inicializado, a leitura dos dados capturados pelo Kinect é feita em dois passos:

a) leitura dos dados provenientes do sensor de profundidade, conforme Quadro 3;
b) leitura dos dados provenientes da câmera *Red Green Blue* (RGB), conforme Quadro 4.

Quadro 3 - Leitura dos dados do sensor de profundidade

```
01. using (var depthFrame = args.OpenDepthImageFrame())
02. {
03.     if (depthFrame == null)
04.         return;
05.
06.     var depthData = new short[depthFrame.PixelDataLength];
07.     depthFrame.CopyPixelDataTo(depthData);
```

Analisando o Quadro 3 é possível observar o acesso aos dados provenientes do sensor de profundidade na linha 01. Já na linha 03 e 04, é executada a validação dos dados vindos do Kinect. Como os dados são baseados em *frames*, é possível que o sensor não capture algum *frame* específico, obrigando a validação executada. Nas linhas 06 e 07 são executados os procedimentos necessários para copiar os dados vindos do sensor.

Da mesma forma que é possível visualizar no Quadro 3, as linhas 01 e 07 do Quadro 4, demonstram a leitura dos dados provenientes da câmera do sensor Kinect.

Quadro 4 - Leitura dos dados da câmera RGB

```
01. using (var colorFrame = args.OpenColorImageFrame())
02. {
03.     if (colorFrame == null)
04.         return;
05.
06.     var colorPixels = new byte[colorFrame.PixelDataLength];
07.     colorFrame.CopyPixelDataTo(colorPixels);
```

A partir da obtenção destes dados, é possível enfim, enviá-los para o algoritmo de processamento para que seja possível identificar o símbolo.

3.3.3 Cálculo de distâncias e representação visual

O cálculo de distâncias consiste em traduzir os dados recebidos do sensor de profundidade em um valor que representa a distância, em milímetros, entre o sensor e objetos próximos. Estes dados recebidos possuem informações referentes à distância e índice do jogador detectado.

Tendo como exemplo os bits recebidos do sensor sendo *0100 1011 1001 0010*, ao mover três bits à direita obtém-se o resultado *0000 1001 0111 0010*, que pode ser convertido para o decimal 2148, representando a distância em milímetros. Essa conversão pode ser observada no Quadro 5, onde é feita uma iteração nos dados recebidos pelo sensor resultando em uma lista contendo as distâncias capturadas.

Quadro 5 - Cálculo das distâncias

```
01. int* BulkProcessing::GetDistances(short* depthData, int dataLength)
02. {
03.   int* distances = new int[dataLength];
04.
05.   for (int i = 0; i < dataLength; ++i)
06.         distances[i] = depthData[i] >> 3;
07.
08.   return distances;
09. }
```

As linhas 05 e 06 podem ser destacadas por executar o acima citado. Após a execução do cálculo das distâncias, é realizada uma iteração nos dados obtidos para preencher os pixels da imagem de acordo com a distância. No Quadro 6, pode-se observar a atribuição de valores de cor para cada camada presente no RGB.

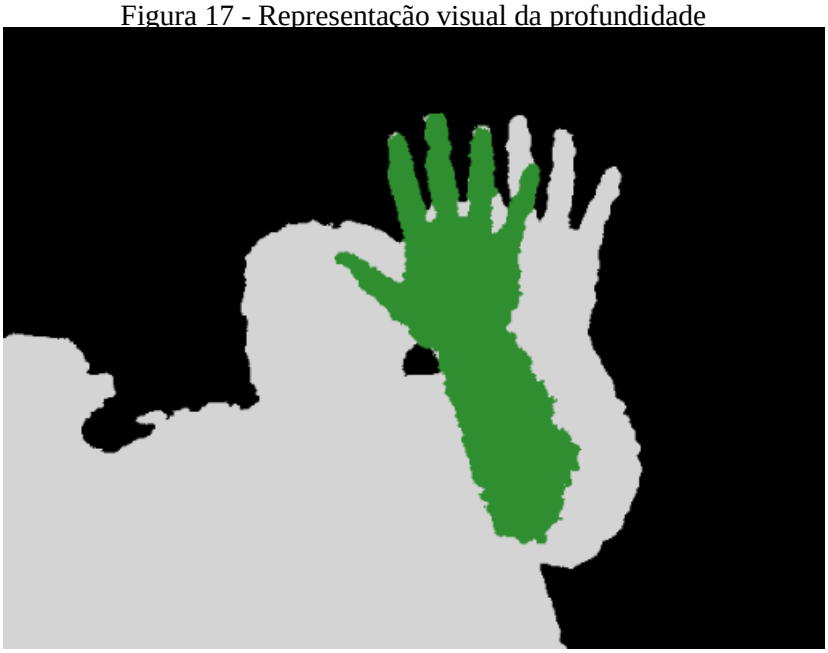
Quadro 6 - Atribuição de cores para representação visual

```
01. for (int i = 0, colorIndex = 0; i < length &&
            colorIndex < pixelLength; ++i, colorIndex += 4)
02. {
03.   ...
04.   else if (distances[i] == 0)
05.   {
06.         depthPixels[colorIndex] = 47;         // Camada do azul
07.         depthPixels[colorIndex + 1] = 142;    // Camada do verde
08.         depthPixels[colorIndex + 2] = 47;     // Camada do vermelho
09.   }
10.   ...
11. }
```

Nas linhas 06, 07 e 08, é possível identificar a atribuição dos valores: 47 para a camada ‘B’, camada do azul; 142 para a camada ‘G’, camada do verde; e 47 para a camada ‘R’, camada do vermelho.

Na Figura 17 é possível notar a representação visual dos dados trabalhados. Para distâncias com valores iguais à zero, um tom de verde foi atribuído com o objetivo de demonstrar que a região está sendo capturada pelo sensor, e esta é a região que será utilizada para processamento no software.

Figura 17 - Representação visual da profundidade



Além das distâncias com valor zero, distâncias com valor igual à -1, demonstrando que o sensor não consegue trabalhar com os objetos muito distantes, recebem a cor preta, e distâncias com um valor máximo de 2000 mm recebem a cor cinza, demonstrando que são capturadas pelo sensor, porém serão descartadas para processamento no software.

Após a atribuição de valores, estes pontos serão demonstrados visualmente ao usuário como um *feedback* para aprimorar o posicionamento do mesmo em relação ao sensor.

3.3.4 Matriz de profundidade e pontos de contorno

Para obter a área que será utilizada para processamento do gesto do usuário, uma matriz é criada com o objetivo de filtrar apenas os dados desejados, demonstrados aqui através de valores do tipo `bool`, indicando se um ponto específico será utilizado ou não no processamento.

No Quadro 7, é possível verificar na linha 11 que a avaliação dos pontos para processamento leva em conta apenas distâncias de valor zero, que conforme explicado previamente, é a distância trabalhada pelo software.

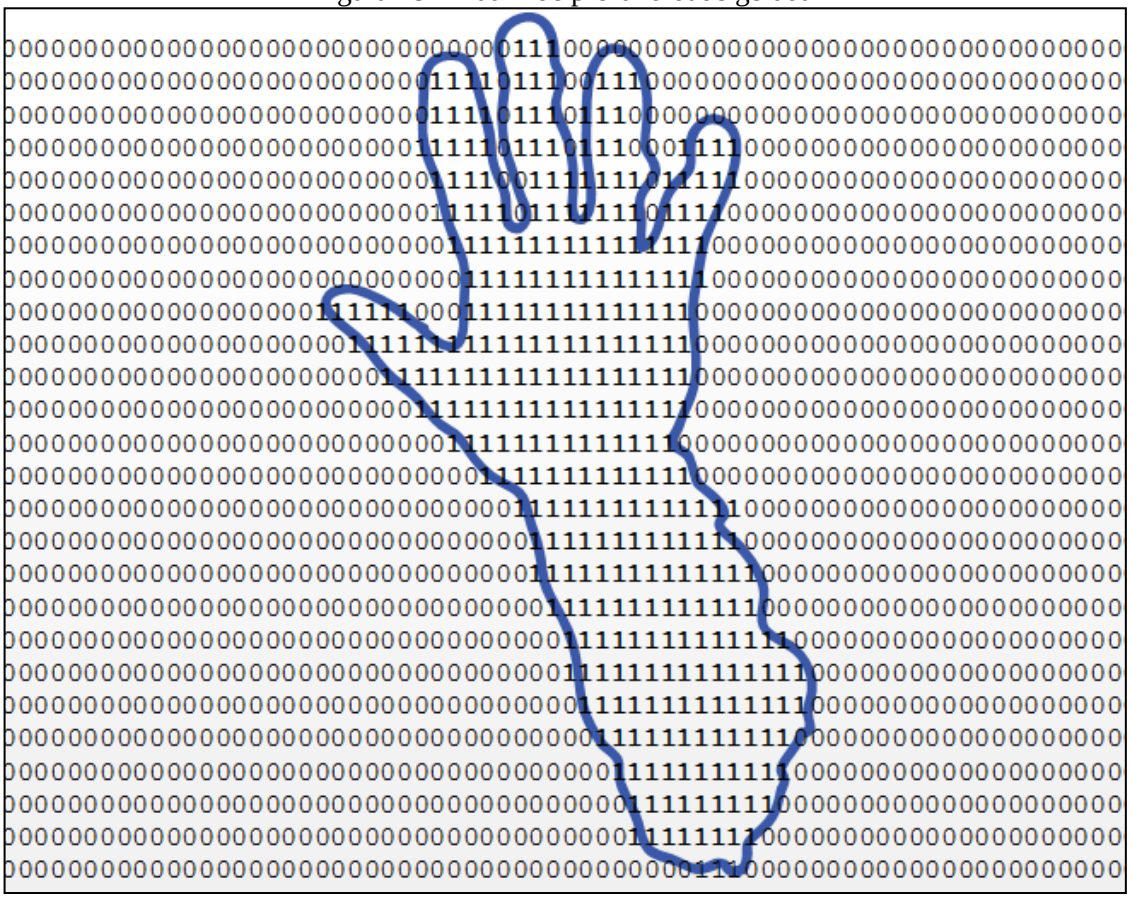
Quadro 7 - Construção da matriz de profundidade

```
01. bool* BulkProcessing::GetNearMatrix(int* distances)
02. {
03.     bool* nearMatrix = new bool[WIDTH*HEIGHT];
04.     memset(nearMatrix, 0, sizeof(bool)*WIDTH*HEIGHT);
05.
06.     for (int i = 0; i < HEIGHT; ++i)
07.     {
08.         for (int j = 0; j < WIDTH; ++j)
09.         {
10.             int index = i * WIDTH + j;
11.             nearMatrix[index] = distances[index] == 0;
12.         }
13.     }
14.
15.     return nearMatrix;
16. }
```

Ainda no Quadro 6 destacam-se as linhas 03 e 04 pela criação da matriz utilizada para guardar os valores pertinentes à profundidade. Já nas linhas 06 e 08, é possível visualizar a iteração executada no vetor `distances`, utilizando as constantes `WIDTH` e `HEIGHT` que simbolizam uma iteração em todos os *pixels* da imagem capturada.

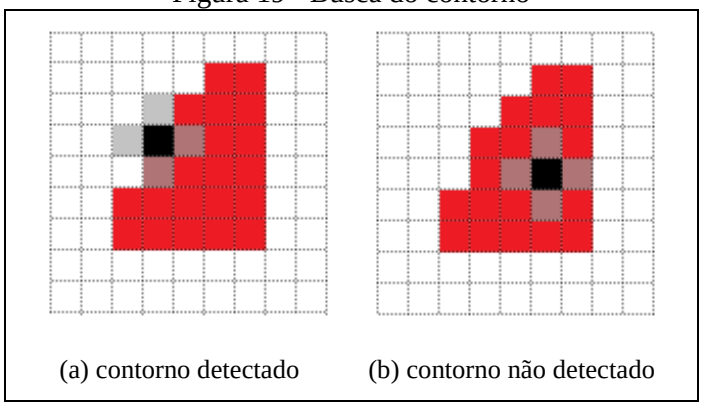
Na Figura 18 é possível visualizar um exemplo da matriz de profundidade, resultado da execução do código presente no Quadro 7.

Figura 18 - Matriz de profundidade gerada



Após a geração da matriz de profundidade é realizada a detecção do contorno da mão. Tendo como ponto de partida a matriz de profundidade, é possível definir os pontos de contorno baseando-se na quantidade de pontos adjacentes ao mesmo. Ao analisar a Figura 19, é possível visualizar como a implementação funciona.

Figura 19 - Busca do contorno



(a) contorno detectado (b) contorno não detectado

Na imagem da esquerda, observa-se que o pixel destacado em preto é um ponto de contorno por possuir ao menos um pixel adjacente não pertencente à matriz de profundidade. Já na imagem da direita, todos os quatro pixels comparados pelo algoritmo pertencem à matriz, caracterizando-o como não participante do contorno. O Quadro 8 possui o código que é executado para determinar o contorno.

Quadro 8 - Algoritmo para determinar o contorno

```
01. for (int i = 1; i < HEIGHT - 1; ++i)
02. {
03.     for (int j = 1; j < WIDTH - 1; ++j)
04.     {
05.         int index = WIDTH * i + j;
06.         if (nearMatrix[index])
07.         {
08.             int count = GetValidAdjacentPixelsCount(nearMatrix,
                                i, j);
09.
10.             Vector2D<double> point(i, j);
11.
12.             if (count != 4)
13.                 contourPoints.push_back(point);
14.         }
15.     }
16. }
```

Nas linhas 01 e 03 é possível observar novamente a iteração utilizando as constantes `WIDTH` e `HEIGHT`, com o intuito de simular uma iteração em todos pixels da imagem. A linha 06 é responsável por definir, baseando-se na matriz de profundidade, se o *pixel* atual será contado ou não para construção do contorno. Nas linhas 08 e 12 é executado o cálculo dos *pixels* adjacentes e a validação baseada na quantidade de *pixels* retornados pela chamada do método `GetValidAdjacentPixelsCount`.

Na linha 13 do Quadro 8 é possível visualizar que cada ponto é adicionado à lista de pontos válidos do contorno. Porém, ainda é necessário ordenar a lista de pontos para poder processar a redução do contorno, com objetivo de encontrar uma forma geométrica.

No Quadro 9, os pontos de contorno são ordenados utilizando o algoritmo da tartaruga, para uso posterior do algoritmo de Ramer-Douglas-Peucker. Caso os pontos não fossem ordenados seria impossível encontrar uma forma geométrica pelo modo como os mesmos são adicionados ao contorno.

Além de ordenar os pontos e criar uma forma “fechada”, o algoritmo da tartaruga também valida se o objeto detectado é suficientemente grande para ser uma mão, validando a quantidade de pontos encontrados.

Quadro 9 - Ordenação dos pontos de contorno

```
01.   std::vector< Vector2D<double> > points;
02.   Vector2D<double> last(-1, -1);
03.
04.   Vector2D<double> current(start.X(), start.Y());
05.
06.   int dir = 0;
07.
08.   do
09.   {
10.         int index = current.X() * WIDTH + current.Y();
11.         if (nearMatrix[index])
12.         {

13.               dir = (dir + 1) % 4;
14.               if (current != last)
15.               {
16.                     Vector2D<double> newPoint(current.X(),
                                                  current.Y());

17.
18.                     points.push_back(newPoint);
19.
20.                     Vector2D<double> newLastPoint(current.X(),
                                                      current.Y());
21.
22.                     last = newLastPoint;
23.                     contour[index] = false;
24.               }
25.         }
26.         else
27.               dir = (dir + 3) % 4;
28.
29.         switch(dir)
30.         {
31.         case 0:
32.               current.X(current.X() + 1);
33.               break;
34.         case 1:
35.               current.Y(current.Y() + 1);
36.               break;
37.         case 2:
38.               current.X(current.X() - 1);
39.               break;
40.         case 3:
41.               current.Y(current.Y() - 1);
42.               break;
43.         }
44.   } while (current != start);
```

Na linha 08, finalizando na linha 44, é feita a iteração do ponto inicial do contorno até o último, que no caso do software é o próprio ponto inicial. Nas linhas 11 e 13, é possível verificar a validação executada objetivando inserir o ponto validado na iteração de forma ordenada, conforme linha 18. Para continuar a validação dos pontos de contorno é necessário decidir na direção de qual eixo será feita a iteração, conforme é possível visualizar entre as linhas 29 e 42.

3.3.5 Representação visual do contorno e cálculo dos pontos reduzidos

Para representar visualmente o contorno que está sendo trabalhado optou-se por atribuir a cor vermelha aos pixels que compõem o mesmo. No quadro abaixo (Quadro 10), é possível visualizar a iteração sobre os pontos para representação do contorno na imagem trabalhada.

Quadro 10 - Representação visual do contorno

```
01. for (size_t i = 0; i < contourPoints.size(); ++i)
02. {
03.     Vector2D<double>* currentPoint = &contourPoints.at(i);
04.     int position = ((currentPoint->X() - 1) * WIDTH * 4) +
                                    ((currentPoint->Y() - 1) * 4);

05.     colorPixels[position] = 0;       // Camada do azul
06.     colorPixels[position+1] = 0;     // Camada do verde
07.     colorPixels[position+2] = 255;   // Camada do vermelho
08. }
```

Após finalizar a obtenção dos pontos do contorno, o algoritmo Ramer-Douglas-Peucker é executado para simplicar as linhas, e obter uma forma geométrica que será utilizada para obtenção de um símbolo.

O código responsável pela redução dos pontos de contorno em linhas encontra-se no Quadro 11. Destaca-se o cálculo da distância perpendicular entre os pontos na linha 03, a adição de um ponto chave para a criação da forma geométrica na linha 13, e as chamadas, que neste caso são recursivas, à própria função de redução de pontos nas linhas 15 e 16.

Quadro 11 - Redução dos pontos de contorno

```
01. for (int index = firstPoint; index < lastPoint; ++index)
02. {
03.   double distance = GetPerpendicularDistance(
                              contourPoints[firstPoint],
                              contourPoints[lastPoint],
                              contourPoints[index]
                         );
04.   if (distance > maxDistance)
05.   {
06.     maxDistance = distance;
07.     indexFarthest = index;
08.   }
09. }
10.
11. if (maxDistance > tolerance && indexFarthest != 0)
12. {
13.   pointsIndexes.push_back(indexFarthest);
14.
15.   DouglasPeuckerReduction(contourPoints, firstPoint,
            indexFarthest, tolerance, pointsIndexes);
16.   DouglasPeuckerReduction(contourPoints, indexFarthest,
            lastPoint, tolerance, pointsIndexes);
17. }
```

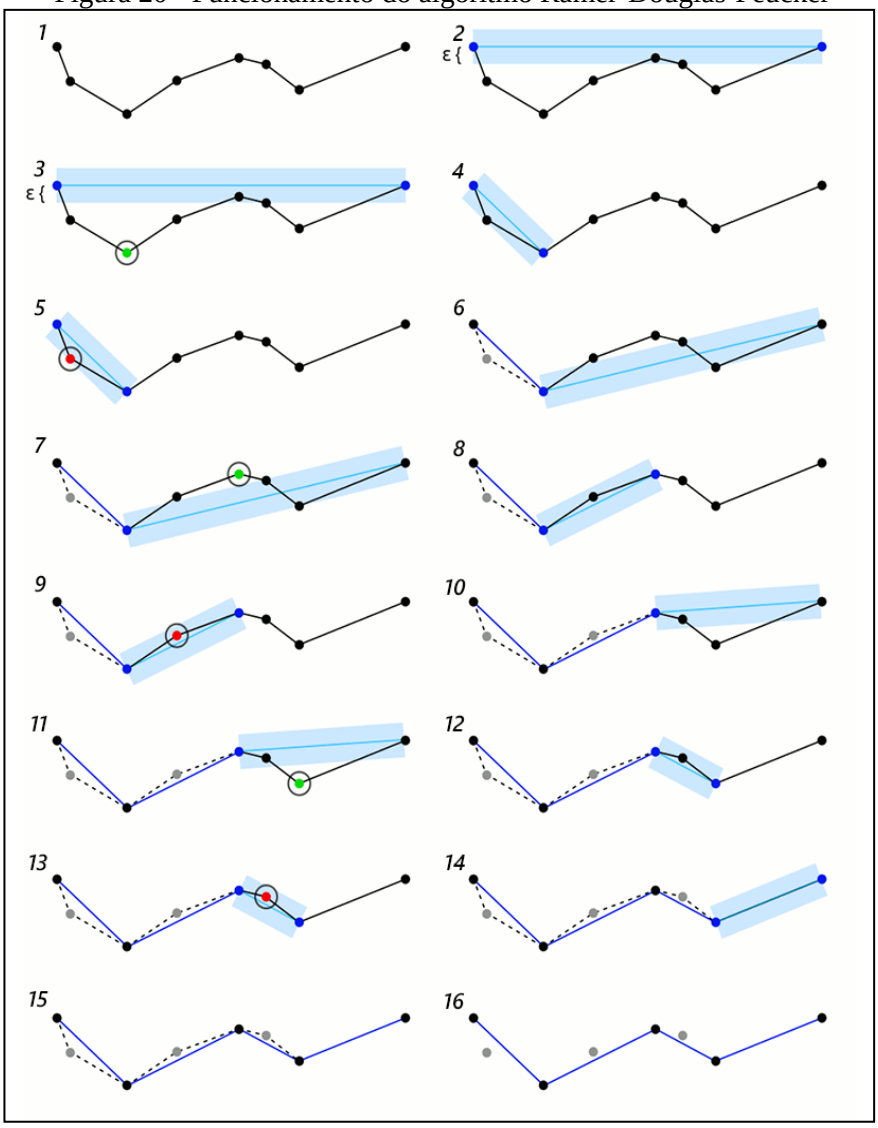
Após o cálculo das linhas para forma geométrica, os últimos passos são constituídos em dar um feedback visual ao usuário do contorno do objeto trabalhado e por fim, identificar o símbolo desejado.

3.3.6 Desenho das linhas

Para representar as linhas do contorno para o usuário, é realizada uma iteração sobre os pontos resultantes da execução do algoritmo de Ramer-Douglas-Peucker conforme seção 2.3.

Na figura abaixo (Figura 20), é possível interpretar o funcionamento do algoritmo de Ramer-Douglas-Peucker. O primeiro passo executado, que é em forma de loop, sendo consecutivamente executado com todos os pontos, é calcular qual o ponto mais distante da reta formada entre o ponto inicial e o último ponto comparado (passo 1).

Figura 20 - Funcionamento do algoritmo Ramer-Douglas-Peucker



Para o cálculo dos pontos pertencentes, um valor chamado de tolerância será utilizado para delimitar a distância máxima entre a reta formada pelos pontos, e o ponto mais distante desta mesma reta. Caso o ponto mais distante possua uma distância maior que a tolerância,

este ponto deve ser considerado como ponto final, e a validação deve ser executada novamente com o ponto inicial e o encontrado, respectivamente (passo 4).

Ao executar o algoritmo novamente, caso a distância encontrada seja menor ou igual à tolerância este ponto deverá ser excluído, reduzindo a quantidade de pontos no contorno, porém mantendo a forma geométrica. Conforme explicado acima, o algoritmo é executado recursivamente para todos pontos. Após a execução completa, o resultado final será utilizado no reconhecimento dos símbolos.

A partir do quadro abaixo (Quadro 12), é possível visualizar que na linha 12 é feito o cálculo dos pontos do segmento de reta que liga o `currentPoint` com o `nextPoint`, que representam o ponto atual e o próximo ponto respectivamente.

Quadro 12 - Representação visual dos pontos reduzidos

```
01. for (int i = 0; i < reducedPoints.size(); ++i)
02. {
03.   Vector2D<double>* currentPoint = &reducedPoints.at(i);
04.   Vector2D<double>* nextPoint;
05.
06.   if (i + 1 < reducedPoints.size())
07.     nextPoint = &reducedPoints.at(i+1);
08.   else
10.     nextPoint = &reducedPoints.at(0);
11.
12.   std::vector< Vector2D<double> > linePoints =
            GetPointsOnLine(currentPoint->X(),
            currentPoint->Y(), nextPoint->X(),
            nextPoint->Y());
13.
```

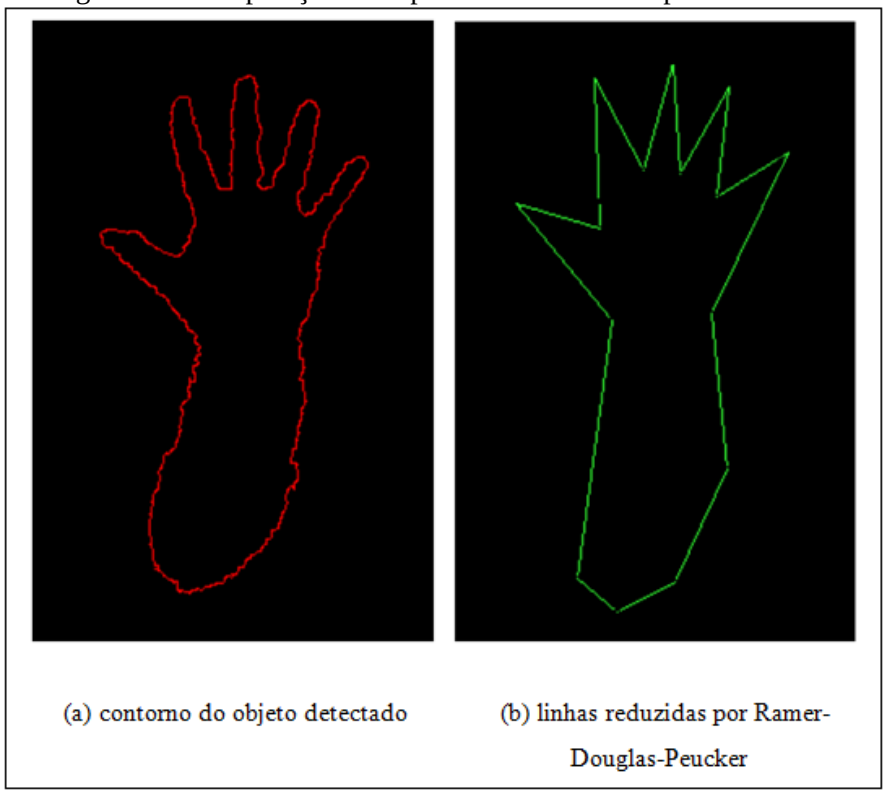
Para calcular os pontos desde segmento foi utilizado o algoritmo de Bresenham, que se utiliza da equação da linha, representada no Quadro 13, para calcular os pontos pertencentes à mesma.

Quadro 13 - Equação da linha

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0.$$

Analisando a Figura 21, torna-se possível visualizar o objetivo do algoritmo Ramer-Douglas-Peucker em conjunto com o algoritmo de Bresenham. À esquerda, pode-se visualizar o contorno da mão e braço do usuário, e à direita o resultado da aplicação de ambos os algoritmos.

Figura 21 - Comparação entre pontos de contorno e pontos reduzidos

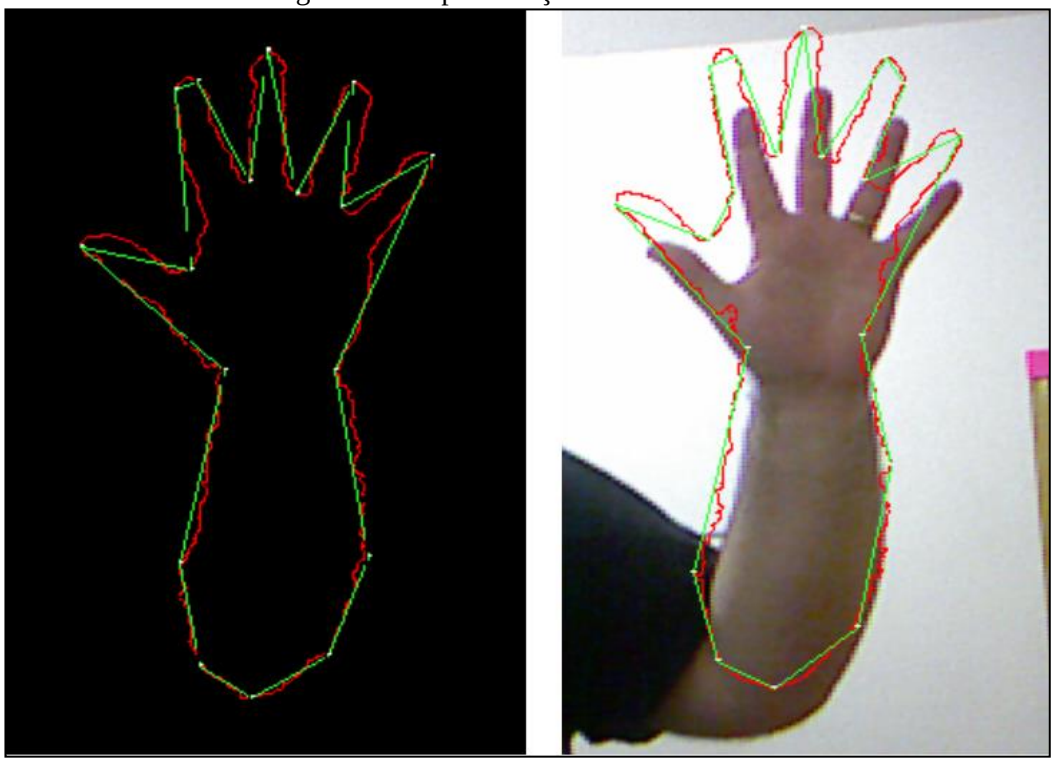


(a) contorno do objeto detectado

(b) linhas reduzidas por Ramer-Douglas-Peucker

Ainda, na Figura 22 é possível visualizar o *feedback* final dado ao usuário. À esquerda encontra-se a sobreposição de contornos, e à direita o resultado demonstrado.

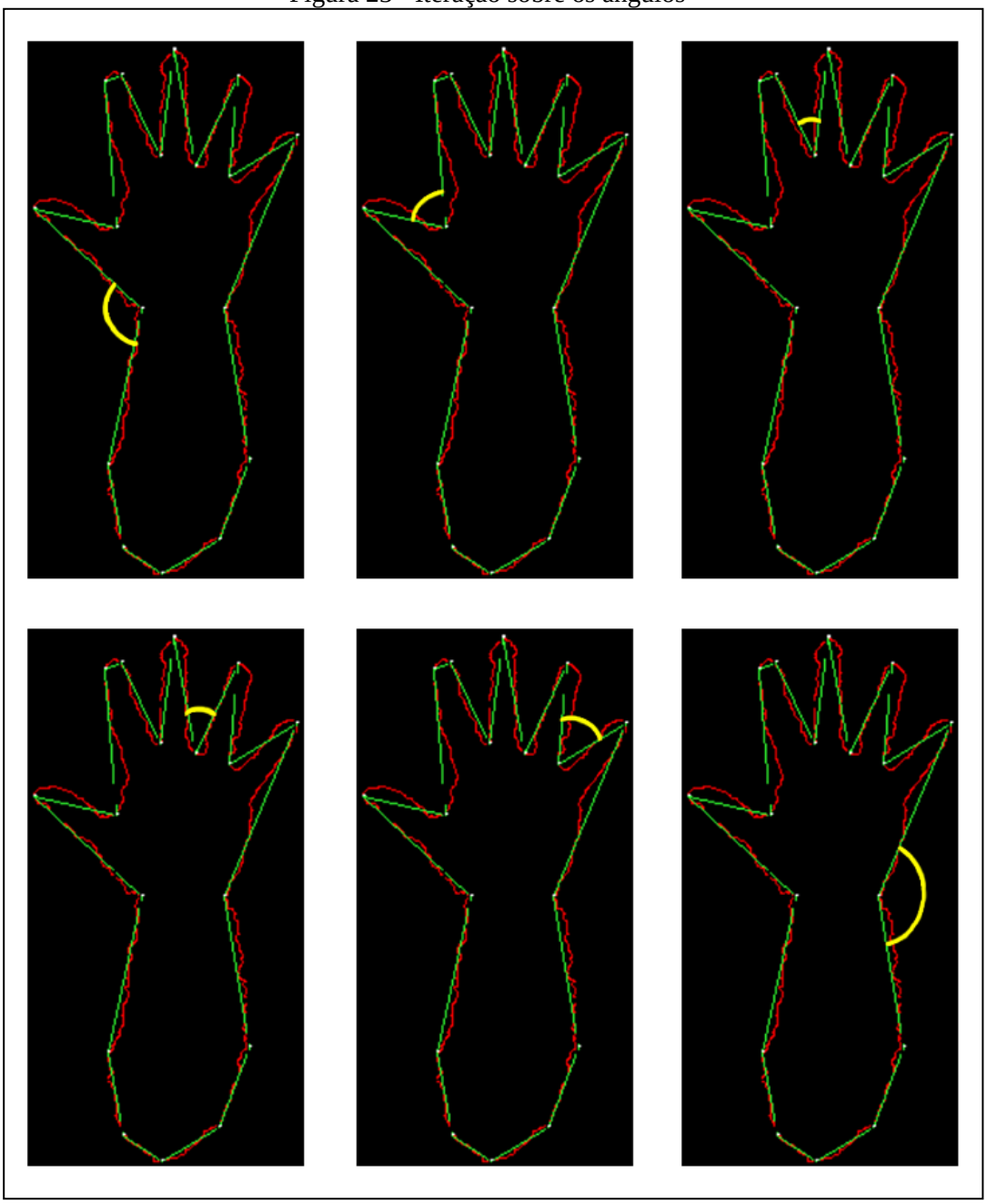
Figura 22 - Representação final ao usuário



3.3.7 Reconhecimento do símbolo

Para reconhecer o símbolo representado pelo usuário, optou-se por uma comparação dos ângulos existentes na forma geométrica gerada pelo algoritmo de Ramer-Douglas-Peucker. Utilizando-se da iteração do algoritmo, a Figura 23 representa, de forma hipotética, a sequência de análise dos ângulos.

Figura 23 - Iteração sobre os ângulos



Para possibilitar o entendimento sobre o funcionamento do algoritmo, é possível visualizar a Figura 24 abaixo contendo os ângulos representados no quadro abaixo (Quadro 14).

Figura 24 - Ilustração dos ângulos cadastrados

Quadro 14 - XML para configuração de símbolos

```
<SymbolData xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Symbols>
    <Symbol character="53">
      <SymbolPoints>
        <SymbolPoint angle="55" />
        <SymbolPoint angle="40" />
        <SymbolPoint angle="15" />
        <SymbolPoint angle="15" />
        <SymbolPoint angle="25" />
        <SymbolPoint angle="150" />
      </SymbolPoints>
    </Symbol>
  </Symbols>
</SymbolData>
```

Após a obtenção dos ângulos existentes no símbolo, é feita uma comparação utilizando os dados cadastrados no XML. Esta comparação executa uma validação, onde o valor cadastrado deve ser o mais próximo possível do valor detectado na forma capturada. No Quadro 14 é possível visualizar a estrutura dos símbolos cadastrados:

a) o caractere correspondente ao símbolo representado pelo seu código da tabela ASCII. É descrito utilizando a sintaxe `<Symbol character="53">`, onde o número 53 representa o caractere ‘5’;
b) a lista de ângulos que constituem a definição do símbolo, representados pela sintaxe `<SymbolPoint angle="55" />`, onde o número 55 representa o valor do ângulo.

No Quadro 15, é possível visualizar a iteração nos ângulos existentes na forma, e a comparação com os valores previamente cadastrados no arquivo XML.

Quadro 15 - Código para reconhecimento de um símbolo

```
01. for (int i = 0; i < SYMBOL_COUNT; ++i)
02. {
03.   Symbol currentSymbol = symbols[i];
04.   int pointCount = GetPointCount(&currentSymbol);
05.
06.   int matchedAngleCount = 0;
07.   for (int j = 0; j < pointCount; ++j)
08.   {
09.         SymbolPoint currentPoint = currentSymbol.SymbolPoints[j];

10.
11.         angleComparer comparer(currentPoint);
12.         std::vector<float>::iterator result =
std::find_if(angles.begin(), angles.end(), comparer);
13.
14.         if (result != angles.end()) {
15.               ++matchedAngleCount;
16.         }
17.   }
18.
19.   if (matchedAngleCount == pointCount)
20.         identifiedSymbol = currentSymbol.Character;
21. }
```

Observando o quadro acima (Quadro 15), é possível destacar as linhas 01, 04 e 09 pelo uso da variável `SYMBOL_COUNT`. A variável `SYMBOL_COUNT` é utilizada para demonstrar a quantidade símbolos cadastrados no arquivo `XML`, juntamente com a variável `pointCount`, que é utilizada para demonstrar quantos ângulos foram cadastrados no símbolo atual.

Ainda no Quadro 15, as linhas 11 e 12 são responsáveis por determinar se um ângulo específico está contido ou não na forma geométrica atual. Na linha 12, é feita a iteração em todos os ângulos capturados da forma utilizando o método `std::find_if` para comparação com o ponto atual.

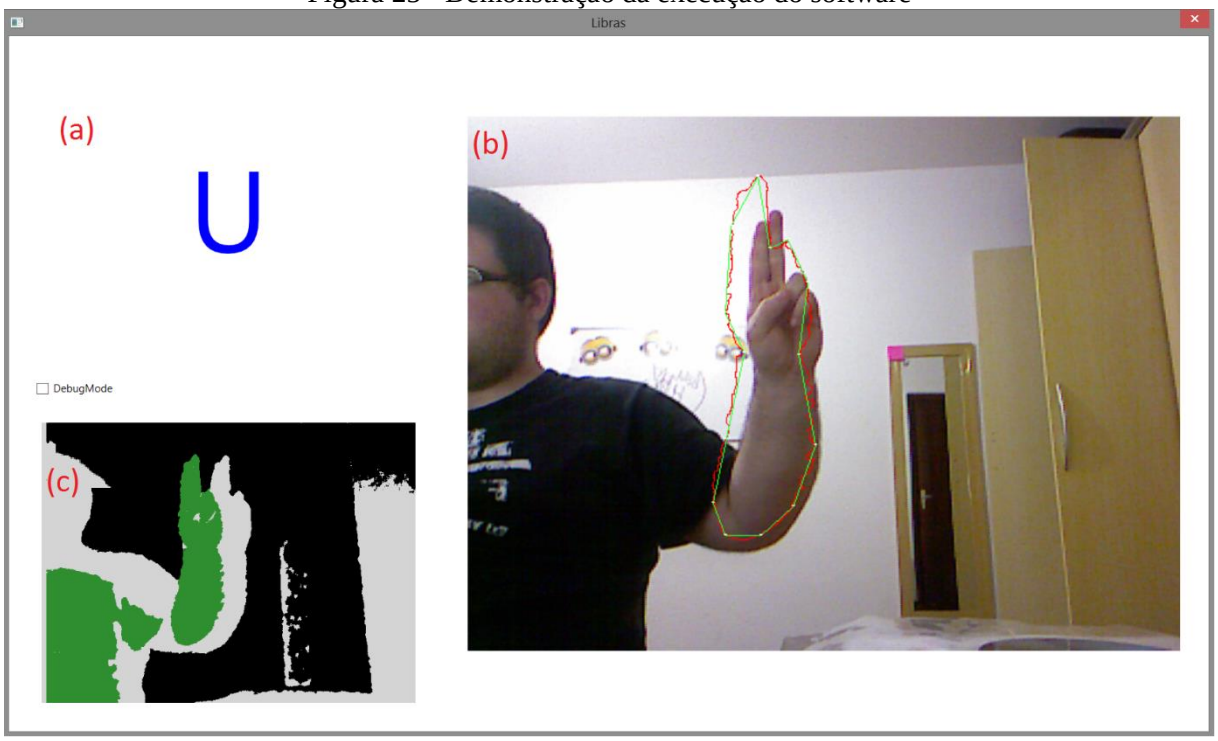
Após a detecção do símbolo, o caractere representado pelo mesmo é demonstrado visualmente para o usuário na interface do software. Caso nenhum símbolo seja encontrado, o caractere ‘?’ é demonstrado no lugar.

3.3.8 Operacionalidade da implementação

Para utilizar o software o usuário deve iniciá-lo com o Kinect devidamente conectado ao computador. O software vai detectar o Kinect e iniciar a captura de dados e apresentar a transcrição do símbolo detectado, a representação da distância entre o usuário e o sensor, e o vídeo obtido pela câmera do Kinect. Caso nenhum sensor esteja conectado a aplicação será encerrada.

O software possui uma janela que apresenta todos os dados trabalhados. Na Figura 25 é possível visualizar todas estas informações, sendo a parte demarcada como item (a) representa a transcrição do símbolo capturado, como item (b) os dados obtidos pela câmera do Kinect, e como item (c) a representação da distância entre o sensor e o usuário.

Figura 25 - Demonstração da execução do software



Pelo fato do software executar a transcrição de símbolos indefinidamente, o fim de sua execução dar-se-á, exclusivamente, fechando o software.

3.4 RESULTADOS E DISCUSSÃO

Para validar a implementação, foram efetuados testes utilizando um arquivo XML contendo os dados de cada símbolo existente no alfabeto datilológico. A seção 3.4.1 descreve os testes efetuados com a linguagem C# e técnicas para detecção do símbolo e comparação. A seção 3.4.2 apresenta resultados de identificação de alguns símbolos, enquanto a seção 3.4.3 demonstra, brevemente, um quadro com a comparação do software desenvolvida com trabalhos correlatos.

3.4.1 Experimentos descartados

Durante a implementação, a primeira linguagem utilizada para o processamento foi o C#. A escolha desta linguagem baseou-se na disponibilidade da mesma como ferramenta para manipulação dos dados provenientes do Kinect. Além disto, a maior parte dos exemplos existentes na internet foram, e continuam sendo, desenvolvidos nesta linguagem.

Após a total implementação dos algoritmos para processamento utilizando a linguagem

C#, percebeu-se que a performance foi abaixo da esperada, onde ao executar o movimento representando qualquer símbolo contido no alfabeto o software travava, impossibilitando e inviabilizando a execução do mesmo. Após uma revisão do código escrito, optou-se por mudar para a linguagem de programação C++.

Outro experimento descartado consiste no algoritmo de detecção do símbolo. A ideia geral era trabalhar a visibilidade e oclusão dos dedos, e identificação da palma da mão do usuário. Conforme o algoritmo foi implementado, percebeu-se uma dificuldade para o discernimento dos dedos do usuário, já que uma grande quantidade de símbolos possuía uma forma que obrigava os dedos a ficarem unidos, impossibilitando uma identificação precisa. Para determinar esta dificuldade, testes simples, como a tentativa de discernir os símbolos 'I' e 'U', demonstraram a impossibilidade de detectar a diferença entre um, dois, ou mais dedos juntos.

Perante isso, a abordagem inicial, de trabalhar a oclusão dos dedos, foi descartada, e passou a ser utilizado o algoritmo para comparação dos ângulos existentes na forma geométrica.

3.4.2 Resultados da transcrição

Para testar o funcionamento do software, foram executados diversos testes com vários símbolos diferentes. Os resultados obtidos foram divididos em duas seções, demonstrando a execução dos testes para símbolos cadastrados separadamente conforme seção 3.4.2.1, e execução dos testes para símbolos cadastrados simultaneamente, conforme seção 3.4.2.2.

3.4.3 Símbolos cadastrados separadamente

A primeira sequência de testes executada consistiu em cadastrar apenas um símbolo por vez no XML. Para testar o funcionamento do software, três símbolos foram aleatoriamente escolhidos para demonstração, onde o primeiro símbolo escolhido foi a letra 'Y'. Para testar a detecção deste símbolo, e de todos outros a seguir, o software foi iniciado com o Kinect devidamente conectado, e o gesto correspondente ao símbolo foi executado em frente ao sensor. Os resultados podem ser vistos nas figuras abaixo (Figura 26, Figura 27 e Figura 28) demonstrando os símbolos 'Y', 'F' e 'U', respectivamente, sendo identificados pelo software.

Figura 26 - Detecção do símbolo Y, cadastrado separadamente

Figura 27 - Detecção do símbolo F, cadastrado separadamente

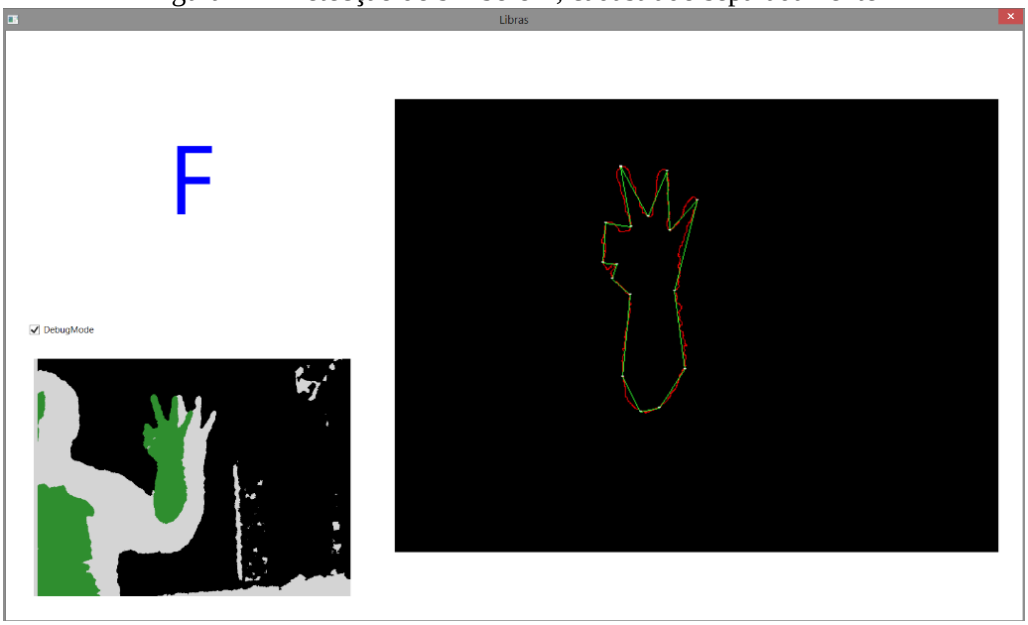
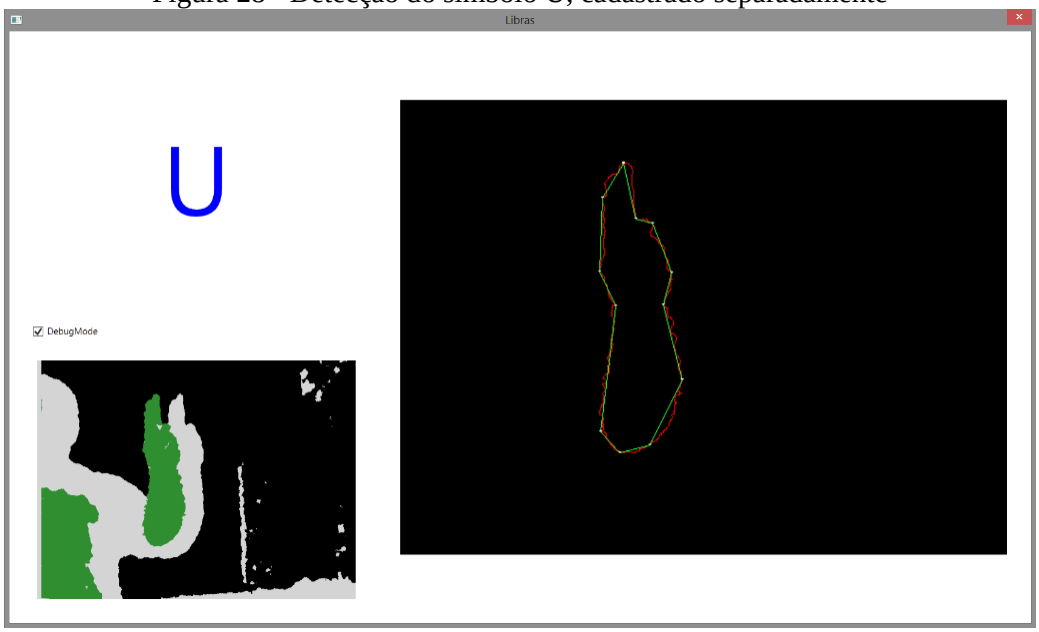


Figura 28 - Detecção do símbolo U, cadastrado separadamente



No quadro abaixo (Quadro 16) é possível observar que todos os símbolos cadastrados foram detectados com sucesso. Os resultados denominados de OK e S-OK especificam que o símbolo foi detectado com sucesso ou com dificuldade, respectivamente. Para os símbolos descritos na tabela abaixo, o resultado final da transcrição ficou em cerca de 70% de acerto.

Quadro 16 - Resultado de transcrição

Símbolos cadastrados separadamente			
0	OK	G	OK
1	OK	I	OK
2	OK	L	OK
3	OK	M	S-OK
4	OK	N	S-OK
5	S-OK	P	OK
6	OK	Q	S-OK
7	OK	R	OK
9	S-OK	T	OK
A	OK	U	S-OK
B	S-OK	V	
C	S-OK	Y	
D	OK		
E	OK		
F	OK		

Os símbolos ‘H’, ‘J’, ‘K’, ‘W’, ‘X’ e ‘Z’ foram excluídos dos testes por necessitarem de movimento na execução, ou por possuírem ângulos muito semelhantes à outros símbolos.

Dentre os símbolos incluídos para detecção, sendo: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘I’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’ e ‘Y’, quando validados separadamente, o índice de detecção alcançou um valor muito próximo a 100%. Todos os símbolos descritos na Tabela 2 receberam OK no resultado de sua validação por terem sido detectados com sucesso, mesmo em casos que foram necessárias mais de uma tentativa para detecção.

3.4.4 Símbolos cadastrados simultaneamente

A segunda sequência de testes executada consistia em cadastrar todos os símbolos simultaneamente no XML. Ao cadastrar todos os símbolos simultaneamente o resultado mostra-se bem abaixo do índice acima citado, recebendo uma taxa de detecção de cerca de 12%.

As figuras seguintes (Figura 29 e Figura 30) demonstrando a falha na detecção dos símbolos ‘E’ e ‘I’ respectivamente, por suas semelhanças com outros símbolos e com a formação dos ângulos existentes nos mesmos.

Figura 29 - Falha na detecção do símbolo E

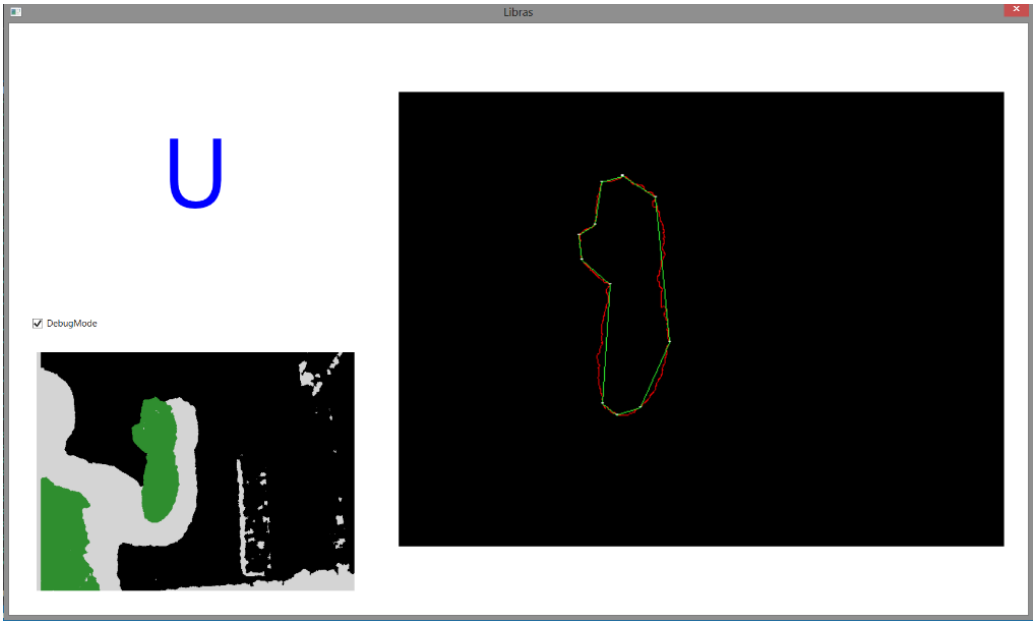
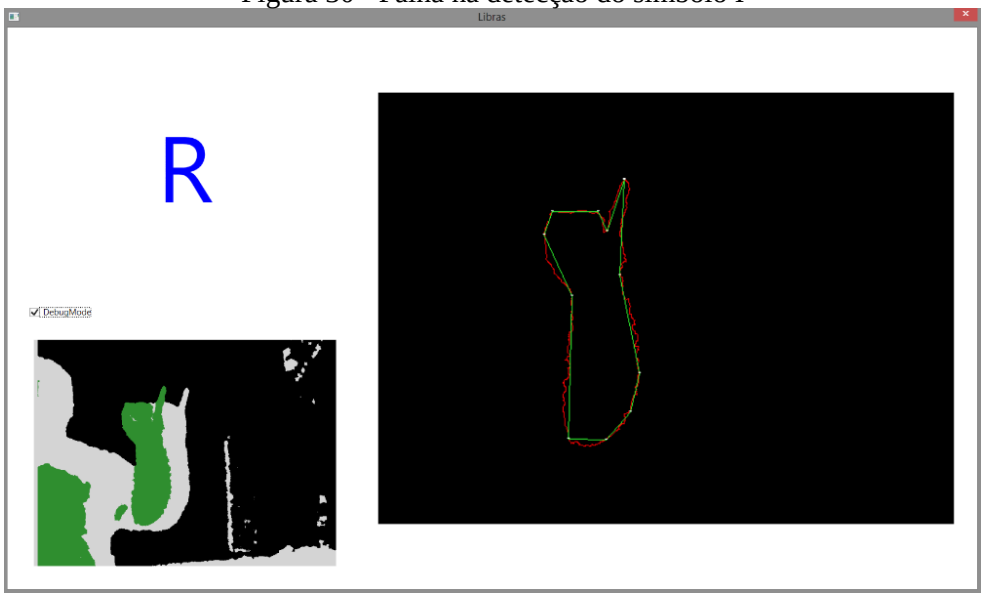


Figura 30 - Falha na detecção do símbolo I



No Quadro 17 é possível visualizar, no mesmo formato encontrado na Tabela 2, o resultado da detecção dos símbolos cadastrados simultaneamente. Os resultados denominados de NÃO OK e S-OK especificam que o símbolo não foi detectado com sucesso ou foi detectado com dificuldade, respectivamente. Para os símbolos descritos na tabela abaixo, o resultado final da transcrição ficou em cerca de 10% de acerto.

Quadro 17 - Resultado da detecção dos símbolos

Símbolos cadastrados em paralelo			
0	NÃO OK	G	NÃO OK
1	NÃO OK	I	NÃO OK
2	NÃO OK	L	NÃO OK
3	NÃO OK	M	NÃO OK
4	NÃO OK	N	NÃO OK
5	NÃO OK	P	NÃO OK
6	NÃO OK	Q	NÃO OK
7	NÃO OK	R	NÃO OK
9	NÃO OK	T	NÃO OK
A	NÃO OK	U	S-OK
B	NÃO OK	V	S-OK
C	NÃO OK	Y	S-OK
D	NÃO OK		
E	NÃO OK		
F	NÃO OK		

Quando a detecção é feita de um símbolo separado, a chance de existirem ângulos correspondentes torna-se muito alta, aumentando a chance de detecção do mesmo. Porém, a chance de existirem símbolos concorrentes, cresce conforme são cadastrados novos símbolos no arquivo XML. Para aprimorar a detecção é possível ajustar a tolerância utilizada para calcular e comparar os ângulos. Os valores utilizados nos testes variam entre 5 e 20 graus, porém nenhum mostrou uma melhora significativa na detecção de múltiplos símbolos simultaneamente.

3.4.5 Comparação com outros projetos que utilizam Kinect

O quadro a seguir (Quadro 18) demonstra uma comparação do software desenvolvido com as principais características de projetos relacionados.

Quadro 18 - Comparação final entre trabalhos

características	Software desenvolvido	RHEMYST e RYMIX (2011)	IZADI et al. (2011)	LÜBKE (2012)
utiliza *skeletal tracking*		X		X
utiliza *stream de vídeo*	X	X	X	X
utiliza *stream de profundidade*	X		X	
utiliza Kinect *for* Windows SDK	X	X	X	X
faz reconhecimento de gestos	X	X		X
transcrição de linguagens de sinais	X			
principal funcionalidade utilizada da Kinect *for* Windows SDK	*stream* de profundidade	*skeletal tracking*	*stream* de profundidade	*stream* de profundidade

A partir do Quadro 17, pode-se observar que o software desenvolvido, comparado com o KinectCAD (LÜBKE, 2012) e "Kinect SDK Dynamic Time Warping Gesture Recognition" (RHEMYST e RYMIX, 2011), que realizam o reconhecimento de gestos, não utiliza *skeletal tracking* para detecção da mão do usuário. Além disso, é possível observar que o software desenvolvido é o único a realizar a transcrição da linguagem de sinais. O trabalho correlato Kinect Fusion (IZADI et al., 2011) não possui o objetivo de reconhecer gestos, mas utiliza o Kinect *for* Windows SDK, sendo que a principal funcionalidade utilizada desse é o *stream* de profundidade, ou seja, a mesma funcionalidade base utilizada no software desenvolvido.

4 CONSIDERAÇÕES FINAIS

Observou-se que a área de reconhecimento de imagem já possui diversas abordagens e interpretações. Porém, uma percentagem muito pequena é voltada ao auxílio de deficientes. Existem diversas ideias para suprir esta necessidade, mas a abordagem deste trabalho torna-se diferente das outras por investir na interação de pessoas com deficiência.

Diversos dos trabalhos observados utilizaram-se do treinamento de gestos, onde o sistema é previamente treinado para aprimorar sua detecção. Diferentes técnicas que utilizam desde o rastreamento do esqueleto humano, até a captura dos dados provenientes do sensor de profundidade, marcaram o estudo dos trabalhos correlatos. Independente qual técnica foi utilizada, alguns dos trabalhos relacionados precisam de ambientes controlados por causa da influência da luz.

Em relação aos trabalhos correlatos, pode-se afirmar que o software proposto diferencia-se dos demais, pois foi construído um software que captura os movimentos do usuário e interpreta-os como sinais do alfabeto datilológico utilizando o sensor Kinect como hardware para captura. Além disto, o software criado utiliza de padrões, para que no momento da detecção da mão possa ser identificado qual padrão melhor se aproxima, aprimorando a precisão. A escolha do sensor Kinect foi baseada na facilidade do seu uso, e pelo fato do sensor já possuir reconhecimento do corpo humano e API disponíveis para lidar e interagir com o mesmo.

Os testes finais apresentaram resultados diferentes do esperado. O algoritmo para transcrição dos símbolos - desenvolvido neste trabalho - não obteve a eficiência de identificação desejada, falhando na diferenciação de símbolos com ângulos parecidos.

O método utilizado para identificação dos símbolos pode ser substituído por uma comparação de imagens, utilizando apenas o contorno obtido pelos algoritmos utilizados. Além disto, os símbolos são detectados em apenas uma posição, não aceitando que seja demonstrado com a mão em uma posição diferente da esperada.

Outro problema encontrado na solução apresentada consiste nos valores cadastrados no arquivo XML. Dependendo do tamanho da mão do usuário, os ângulos podem ter uma variação acima do tolerado, e mesmo que o símbolo seja demonstrado corretamente, a detecção falhará.

Apesar do resultado deste trabalho não ter sido conforme o esperado, ele demonstra um método viável para detecção de sinais, servindo como ponto de partida para trabalhos que possam aperfeiçoar o mecanismo de identificação de símbolos.

4.1 EXTENSÕES

Como proposta de continuidade sugere-se as seguintes melhorias:

a) aumentar a quantidade de ângulos descritos em cada símbolo e aprimorar a precisão dos já existentes;
b) implementar detecção dos símbolos que exijam movimentos, como: “H”, “J”, entre outros;
c) emitir um som correspondente ao símbolo detectado para aprimorar o treinamento do usuário;
d) armazenar todos caracteres transcritos com o objetivo de possibilitar a escrita de uma frase;
e) executar testes com um público alvo maior para estudar a viabilidade de implantação do software como um serviço público e gratuito;
f) rever técnica utilizada para detecção e transcrição do símbolo, incluindo análise para viabilidade do uso de técnicas para reconhecimento de padrões.

REFERÊNCIAS BIBLIOGRÁFICAS

ARAÚJO, Ana P. **Linguagem de Sinais Brasileiras (Libras)**; [S.l]. 2007. Disponivel em: <http://www.infoescola.com/portugues/lingua-brasileira-de-sinais-libras/>. Acesso em: 22 Abril 2013.

CUORE, Raul E. C. **A Importância de conhecer a estrutura linguística de libras para o educador**. Campo Grande, 2009. Disponível em: <http://www.artigonal.com/educacao-artigos/a-importancia-de-conhecer-a-estrutura-linguistica-da-libras-para-o-educador-945026.html>. Acesso em: 25 abr. 2013.

DOUGLAS, David; PEUCKER, Thomas. **Algorithms for the reduction of the number of points required to represent a digitized line or its caricature.** *Cartographica: The International Journal for Geographic Information and Geovisualization*. v. 10, p. 112-122, 1973.

DUNCAN, Greg. **KinectCAD for CATIA.** [S.l.], 2012. Disponível em: < http://channel9.msdn.com/coding4fun/kinect/KinectCAD-for-CATIA>. Acesso em: 15 maio. 2013.

EISLER, Craig. **Kinect for Windows is now available!** [S.l.], 2012. Disponível em: <http://blogs.msdn.com/b/kinectforwindows/archive/2012/01/31/kinect-for-windows-is-now-available.aspx>. Acesso em: 4 abr. 2013.

IZADI, Shahram et al. **KinectFusion:** real-time 3D reconstruction and interaction using a moving depth camera. United Kingdom, 2011. Disponível em: <http://research.microsoft.com/pubs/155416/kinectfusion-uist-comp.pdf>. Acesso em: 15 maio. 2013.

JOY, Kennet I. Bresenham's Algorithm. California, 1999. Disponível em: <http://www.idav.ucdavis.edu/education/GraphicsNotes/Bresenhams-Algorithm.pdf>. Acesso em: 12 jul. 2013.

LOCKTON, Raymond; FITZGIBBON, Andrew. W. **Real-time gesture recognition using deterministic boosting**. *British Machine Vision Conference*. v. 02, [S.l]. 2002.

LÜBKE, Florian. **KinectCAD**. [S.l.], 2012. Disponível em: <http://sourceforge.net/projects/kinectcad/>. Acesso em: 15 maio. 2013.

MICROSOFT. **Coordinate spaces.** [S.l.], 2012a. Disponível em: <http://msdn.microsoft.com/en-us/library/hh973078.aspx>. Acesso em: 04 maio. 2013.

_____. **Interaction space.** [S.l.], 2012b. Disponível em: <http://msdn.microsoft.com/en-us/library/hh973071.aspx>. Acesso em: 04 maio. 2013.

_____. **Kinect for Windows architecture.** [S.l.], 2012c. Disponível em: <http://msdn.microsoft.com/en-us/library/jj131023.aspx>. Acesso em: 04 maio. 2013.

_____. **Kinect for Windows sensor components and specifications.** [S.l.], 2012d. Disponível em: <http://msdn.microsoft.com/en-us/library/jj131033.aspx>. Acesso em: 04 maio. 2013.

_____. **Kinect Natural User Interface (NUI) overview.** [S.l.], 2012e. Disponível em: <http://msdn.microsoft.com/en-us/library/hh855348.aspx>. Acesso em: 05 abr. 2013.

_____. **Skeletal tracking.** [S.l.], 2012f. Disponível em: <http://msdn.microsoft.com/en-us/library/hh973074.aspx>. Acesso em: 04 maio. 2013.

MILES, Rob. **Using Kinect for Windows with XNA.** Kinect for Windows SDK. [S.l.], 2012. Disponível em: <http://channel9.msdn.com/coding4fun/kinect/The-Purple-Book-Using-Kinect-for-Windows-with-XNA>. Acesso em: 2 maio. 2013.

MONTEIRO, Myrna. S. História dos movimentos dos surdos e o reconhecimento da libras no brasil. ETD - Educação Temática Digital, Campinas, v. 7, n. 2, p. 292-302, jun. 2006.

OLIVEIRA, Paulo M. T. D. **Sobre surdos**. Site de Jonas Pacheco. [S.l]. 2011. Disponível em: <http://www.surdo.org.br>. Acesso em: 21 abr. 2013.

PACHECO, Jonas; ESTRUC, Eduardo. **Curso básico de libras**. [S.l.]. 2008. Disponível em: <www.surdo.org.br>. Aceso em: 21 abr. 2013.

QUEIROZ, Murilo. **Um cientista explica o Microsoft Kinect.** [S.l.], 2010. Disponível em: <http://blog.vettalabs.com/2010/10/29/um-cientista-explica-o-microsoft-kinetic/>. Acesso em: 4 maio. 2013.

RAMER, Urs. **An iterative procedure for the polygonal approximation of plane curves**. Computer Graphics and Image Processing. v. 1, p. 244-256. 1972.

RAMOS, Célia R. **LIBRAS:** A língua de sinais dos surdos brasileiros, Petrópolis, 2006. Disponível em: <http://www.editora-arara-azul.com.br/pdf/artigo2.pdf>. Acesso em: 21 abr. 2013.

RHEMYST, John; RYMIX, Kaly. **Kinect SDK dynamic time warping (DTW) gesture recognition.** [S.l.], 2011. Disponível em: <http://kinectdtw.codeplex.com/>. Acesso em: 15 maio. 2013.

SOUZA, Vinícius C. D.; PINTO, Sérgio C. C. D. S. Sign WebMessage: uma ferramenta para comunicação via web através da Língua Brasileira de Sinais – Libras. São Leopoldo, 2003. p. 395-404.

WHYATT, Duncan J.; WADE, Philip R. **The Douglas-Peucker line simplification algorithm:** an introduction with programs. [S.l.], 1988. Disponível em: <http://www2.dcs.hull.ac.uk/CISRG/publications/DPs/DP4/DP4.html/>. Acesso em: 13 maio. 2013.