Trabalho de Conclusão de Curso (2012) 2: 1–17
DOI 10.1007/s00002-012-0002-5

GENERIC ARTICLE

Reconstrução de Modelos 3-D em tempo real utilizando *Kinect* e GPU

Márcio Cerqueira de Farias Macedo · Antonio Carlos dos Santos Souza

Recebido: 10/07/2012 / Aceito: 03/09/2012
© Springer-Verlag 2012

Resumo A digitalização de modelos 3-D tem crescido rapidamente nos últimos anos devido principalmente aos dispositivos de captura 3-D, como o *Kinect*. A digitalização desses modelos tem aplicação em várias áreas como engenharia civil, medicina e artes. Este trabalho descreve um sistema que permite a reconstrução de objetos 3-D a partir de mapas de profundidade adquiridos do *Kinect*. O sistema executa em tempo real com o auxílio das GPUs (*Graphics Processing Unit* — Unidade de Processamento Gráfico). Neste trabalho, o sistema será descrito de acordo com suas principais funcionalidades (aquisição dos mapas de profundidade, alinhamento das nuvens de pontos, integração volumétrica e *raycasting* sobre o modelo). Além disso, a modelagem do sistema e os resultados obtidos serão descritos e analisados.

1 Introdução

Pesquisas no campo da computação gráfica tradicionalmente são baseadas em três objetivos: representação das formas tridimensionais (modelagem), descrição do movimento dessas formas (animação) e simulação da luminosidade no ambiente produzindo imagens foto-realísticas (renderização). A produção de imagens e vídeos de formas mais complexas exige um maior detalhamento em cada uma dessas fases, o que também exige maior quantidade de processamento e uma maior dificuldade na produção dos mesmos, sendo ainda necessária a interferência humana para a melhoria das aplicações durante a fase de produção [17].

Márcio Cerqueira de Farias Macedo
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
E-mail: marciomacedo@ifba.edu.br

Antonio Carlos dos Santos Souza
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
E-mail: antoniocarlos@ifba.edu.br

Com o avanço das pesquisas na área, a necessidade de interferência humana no processo de produção de imagens e vídeos mais realísticos tem diminuído, pois, com a utilização de dispositivos de captura 3-D, a forma tridimensional dos objetos pode ser adquirida, permitindo a estimativa de seus movimentos associados, além da medição de luminosidade do ambiente [17].

Os dispositivos de captura 3-D fornecem mapas de profundidade capturados da cena instantaneamente visualizada. Os mapas de profundidade são imagens que contém valores de profundidade da cena associados para cada *pixel*. Para algumas aplicações, não é interessante obter somente o formato tridimensional visualizado de um ângulo pela câmera: reconstrução de órgãos capturados por CT (*Computed Tomography* — Tomografia Computadorizada), de esculturas, cidades e prédios. No caso das cidades e prédios, um modelo parcial/completo permitiria que fossem realizados *walkthroughs* ou visitas técnicas. Já para os órgãos capturados por CT e para as esculturas, uma reconstrução tridimensional mais completa permitiria uma análise mais eficiente e precisa, seja para algum exame, ou para análise dos detalhes da obra. Para esses casos, faz-se necessário o uso de técnicas para reconstrução 3-D (parcial ou completa) da forma desejada.

Outra área que pode utilizar a reconstrução tridimensional para suas aplicações é a Realidade Aumentada. Ela consiste em um conjunto de técnicas para integração dos objetos do mundo real com os objetos virtuais (gerados computacionalmente) no mesmo espaço do mundo real. Uma das suas propriedades é a necessidade de execução interativa e em tempo real [1]. Com o avanço dos dispositivos gráficos, e com o surgimento da GPU, tornou-se possível a realização da

reconstrução tridimensional em tempo real, reconstruindo-se modelos de alta qualidade e muito precisos.

O presente trabalho tem como objetivo a reconstrução tridimensional de um modelo a partir do alinhamento dos diversos mapas de profundidade fornecidos pelo *Kinect*, em tempo real, utilizando GPU. O alinhamento dos mapas de profundidade, representados num ambiente 3-D por nuvens de pontos, permite uma integração de visões de diferentes ângulos de uma mesma cena.

O sistema será analisado a partir de três versões:

– A primeira versão foi feita somente em CPU;
– A segunda versão foi feita utilizando CPU e GPU utilizando a linguagem *OpenCL* [22], sendo que foram utilizados mapas de profundidade sintéticos para os testes de reconstrução;
– A terceira versão consiste numa adaptação do sistema de reconstrução 3-D implementado como um projeto do PCL [18], que utiliza a linguagem *CUDA* [19] e funciona em tempo real com os mapas de profundidade adquiridos diretamente do *Kinect*.

As duas primeiras versões foram utilizadas para os testes com modelos sintéticos, permitindo também testes comparando o tempo de processamento em CPU e GPU dos algoritmos utilizados. Já a adaptação feita na terceira versão do sistema teve como objetivo melhorar o encapsulamento e reduzir o acoplamento entre os módulos implementados pelo sistema de reconstrução 3-D do PCL.

O presente trabalho está estruturado da seguinte forma: A seção 2 apresenta os principais trabalhos relacionados à área de reconstrução tridimensional em tempo real. A seção 3 apresenta a fundamentação teórica relativa à reconstrução tridimensional. Nessa seção, os algoritmos são apresentados baseados em CPU. A seção 4 apresenta a fundamentação teórica sobre a Unidade de Processamento Gráfico (GPU), além das adaptações e otimizações feitas nos algoritmos mostrados na seção 3 para a execução em GPU nas duas últimas versões do sistema. A seção 5 apresenta a documentação do sistema, com os diagramas de classes e sequência para as duas primeiras versões do sistema; casos de uso e arquitetura para todas as três versões do sistema. Já a seção 6 apresenta uma análise sobre o tempo de processamento do sistema para cada *frame* obtido, bem como uma discussão sobre a qualidade do modelo reconstruído e outros testes realizados sobre os modelos em reconstrução.

2 Trabalhos Relacionados

Como exemplos de trabalhos que realizam a reconstrução tridimensional em **tempo real**, temos:

Sistema de aquisição e renderização em tempo real de modelos 3-D de Szymon Rusinkiewicz [17]: sistema que faz a reconstrução completa de um modelo 3-D em tempo real utilizando apenas o processamento da CPU. Enquanto algoritmos bastante velozes são utilizados para a reconstrução tridimensional, o sistema apresenta baixa qualidade nos modelos reconstruídos.

KinectFusion de Izadi et al. [9]: sistema que utiliza a paralelização proporcionada pela GPU e o *Kinect* como dispositivo de captura para realizar reconstrução de cenas em 3-D. Além disso, o *KinectFusion* possui a funcionalidade da realidade aumentada sem o uso de marcadores. Durante a reconstrução tridimensional, o *KinectFusion* utiliza o algoritmo ICP implementado em GPU para o alinhamento das nuvens de pontos. A diferença entre o nosso trabalho e o *KinectFusion* está nos testes realizados e no algoritmo do cálculo do SDF para a *grid* volumétrica.

Sistema de animação facial de Weise et al. [27]: sistema que utiliza *blendshapes* para representar o modelo facial reconstruído. Diferente dos outros trabalhos citados, este segmenta a face em regiões rígidas e não-rígidas para o alinhamento com ICP e, a partir de um modelo probabilístico, identifica qual *blendshape* representa a expressão do usuário. A desvantagem desta abordagem está na perda de precisão associada à utilização do *blendshape*, visto que ele representa um modelo facial padrão, não sendo específico para cada tipo de usuário.

Sistema de reconstrução de objetos baseado em *surfels* de Weise et al. [28]: Diferente dos outros sistemas mencionados, este sistema representa um objeto como um *surfel* [8]. A partir disso, um algoritmo ICP não-rígido é aplicado para o alinhamento dos *frames* sucessivos com o modelo em reconstrução, e uma abordagem similar ao *KinectFusion* é utilizada para a integração volumétrica. Esse sistema funciona em tempo real e consegue reconstruir objetos com qualidade superior a outras abordagens que não funcionam em tempo real.

3 Reconstrução tridimensional

Nesta seção serão examinadas algumas das técnicas e algoritmos utilizados para cada passo da reconstrução tridimensional.

3.1 Aquisição dos mapas de profundidade e geração das nuvens de pontos

Os dispositivos de captura de profundidade da cena podem ser classificados em ativos e passivos. Os ativos são os que fornecem e controlam a sua própria iluminação; os passivos apenas absorvem as radiações do ambiente, e a partir delas buscam extrair informações de profundidade da cena [31].

Um dos métodos mais utilizados com sensores passivos é a visão estéreo. Um exemplo de aplicação em tempo real

pode ser visto em [24], em que os autores utilizam um sistema com visão estéreo para a reconstrução tridimensional de alta precisão do ambiente, permitindo interação virtual entre os usuários.

Para os sensores ativos, os métodos mais conhecidos são: ToF (*Time-of-Flight* — Tempo de voo) e triangulação de luz estruturada projetada. Nas câmeras ToF, a profundidade da cena é obtida através do tempo de duração entre a emissão da luz pela unidade de iluminação e o retorno dela ao sensor (um *survey* sobre o assunto pode ser visto em [11]). Para a triangulação de luz estruturada projetada [3, 15], o mapa de profundidade é obtido a partir da triangulação feita entre a câmera receptora e o emissor IR do equipamento. Essa técnica consiste na iluminação da cena feita pelo emissor, com uma faixa de luz conhecida, e na observação da mesma cena pela câmera, posicionada em um ângulo conhecido em relação ao emissor IR. A posição tridimensional dos pontos da cena é determinada a partir da interseção entre a direção da visão da câmera e a direção da luz produzida pelo emissor (Figura 1) [23]. Câmeras de luz estruturada tem a vantagem de prover mapas de profundidade densos com alta precisão em relação às câmeras ToF e às utilizadas para a visão estéreo [28].

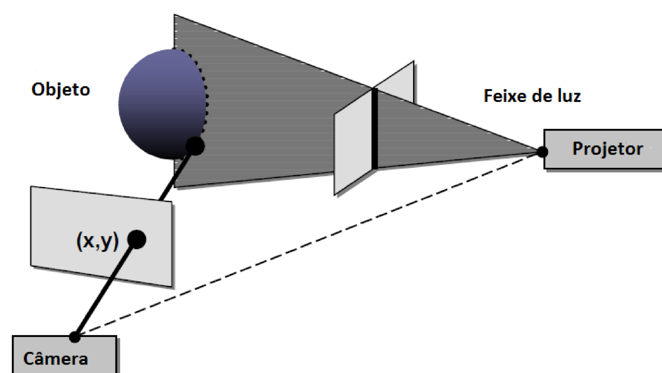

Figura 1 Princípio da triangulação de luz estruturada. A localização 3-D é determinada a partir da interseção entre a direção da visão da câmera e a direção da luz produzida pelo emissor [23](Figura modificada de [17]).

Neste trabalho, a aquisição dos mapas de profundidade é feita a partir do *Kinect*. Este equipamento contém uma câmera RGB, uma câmera e um emissor IR, sendo que o mapa de profundidade é obtido a partir da triangulação de luz estruturada projetada.

O *Kinect* é um sensor de baixo custo que fornece mapas de profundidade ruidosos e com muitos buracos. Para a redução dessas características, filtros muito conhecidos como o gaussiano [29] e/ou o bilateral [21] podem ser aplicados sobre os mapas de profundidade. Assim como feito em [9], o filtro bilateral foi o escolhido, pois preserva descontinuidades no mapa de profundidade, reduzindo a quantidade de ruídos presentes no mesmo.

A cada instante i, um mapa de profundidade D_i é gerado pelo *Kinect*. Dada a matriz de calibração intrínseca K da câmera IR do *Kinect*, cada *pixel* u com profundidade associada no mapa de profundidade é reprojetado como um vértice 3-D no sistema de coordenadas da câmera a partir da seguinte equação: $v_i(u) = D_i(u)K^{-1}[u, 1]$. A partir deste processo de calibração, o mapa de profundidade D_i pode ser convertido para o mapa de vértices/nuvem de pontos V_i.

Uma vez com a nuvem de pontos V_i, pode-se calcular os vetores normais associados a cada ponto v_i. Diferente do que foi feito em [9], o cálculo utilizado no sistema proposto é baseado em estatísticas locais de covariância: o vetor normal pode ser obtido a partir do autovetor do menor autovalor da matriz de covariância calculada a partir da nuvem de pontos. Essa abordagem reduziu o erro obtido no modelo reconstruído.

Para cada nuvem de pontos V, é atribuída uma matriz de transformação de corpo rígido $T = [R|t]$, contendo uma matriz de rotação 3×3 (R) e um vetor de translação 3-D (t). A partir dessa transformação, um ponto v com vetor normal n pode ser convertido do sistema de coordenadas da câmera para o sistema de coordenadas global: $v_g(u) = Tv(u)$ e $n_g(u) = Rn(u)$.

Nas próximas seções, as notações v_m e n_m serão utilizadas para identificar um ponto e vetor normal associado ao modelo em reconstrução.

3.2 Alinhamento das nuvens de pontos

Para a reconstrução tridimensional, é necessário obter visões de diferentes ângulos de um objeto, alinhando os mapas de profundidade adquiridos em um sistema de coordenadas comum, que é denominado sistema de coordenadas global. Todos os mapas de profundidade, quando não estão alinhados, estão no sistema de coordenadas da câmera (Figura 2).

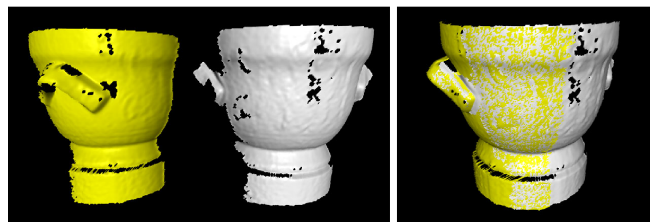

Figura 2 (Esquerda) Nuvens de pontos no sistema de coordenadas da câmera. (Direita) Nuvens de pontos alinhadas e integradas volumetricamente [26].

Para o alinhamento, é necessária a obtenção da transformação rígida que define a relação entre as nuvens de pontos. Esse processo de obtenção da transformação é chamado de

registro [25]. A transformação é rígida porque possivelmente o modelo não sofrerá deformações (mudanças de forma) durante o processo de aquisição.

O registro pode ser classificado em dois tipos: *frame-frame* e *frame-model*. O *frame-frame* consiste no registro das diversas nuvens de pontos sequenciais e apresenta baixa qualidade de reconstrução para suas aplicações. O *frame-model* consiste no registro entre a nuvem de pontos capturada pelo dispositivo de captura 3-D com uma nuvem de pontos que representa o modelo reconstruído. Esse tipo de registro apresenta menor acúmulo de erros durante a reconstrução [9].

Para o alinhamento das diversas nuvens de pontos, o algoritmo ICP (*Iterative Closest Point* - Ponto mais próximo iterativo) [2,5] tem sido o método mais utilizado para a aproximação de alta precisão de duas nuvens de pontos, dada uma boa estimativa inicial de alinhamento. No sistema de reconstrução 3-D, o ICP é utilizado para estimar a pose da câmera para cada mapa de profundidade gerado. Isto é feito baseado no registro *frame-model*, tendo em vista a sua melhor qualidade de reconstrução final [9]. Uma vez com a pose da câmera estimada, podemos converter o sistema de coordenadas da câmera para o sistema de coordenadas global e vice-versa, como mencionado na seção anterior.

O algoritmo ICP pode ser sumarizado em 6 passos:

- Seleção de pontos de uma ou das duas nuvens de pontos;
- Correspondência dos pontos selecionados entre as nuvens de pontos;
- Atribuição de peso para os pares com correspondência;
- Rejeição de alguns pares com correspondência;
- Associação de um erro baseado nas correspondências;
- Minimização do erro.

Foram desenvolvidos muitos variantes do ICP. O variante descrito por [9] é conhecido por ser veloz e eficiente para alinhar nuvens de pontos adquiridas em tempo real. Esse variante pode ser descrito a partir do seguinte algoritmo:

- Seleção de pontos: São selecionados todos os pontos com $D_i(u) > 0$. Isso significa que são selecionados todos os pontos visíveis na cena, que representam o modelo.
- As correspondências são feitas a partir da associação projetiva de dados [16]. Nesse algoritmo, cada ponto v_i é transformado em coordenadas da câmera e é projetado em um plano 2-D, assumindo como seu correspondente o ponto do modelo v_m de mesma coordenada, se existir.
- Atribuição de peso para as correspondências: É atribuído peso constante para todas as correspondências feitas.
- Rejeição de correspondências: A rejeição é feita a partir da distância euclidiana dos pontos selecionados e do ângulo entre os seus vetores normais. Tanto os pontos v_i e v_m quanto os vetores normais n_i e n_m são convertidos do sistema de coordenadas da câmera para o sistema de coordenadas global, e a rejeição é feita para aqueles pares que possuírem os parâmetros supracitados acima de um valor limite pré-definido.
- Associação do erro entre as correspondências: A medição de erro é feita a partir da métrica [5]:

$$\sigma^2 = \sum_{u=1}^{n} ||(p_m(u) - Tp_i(u)) \cdot n_m(u)||^2 \quad (1)$$

  sendo definida como a soma das distâncias quadráticas entre cada ponto e o plano tangente do seu correspondente. Para esta métrica, os únicos *pixels* u válidos são aqueles que estão associados a um par de pontos correspondentes.
- Minimização de erro: Para a minimização de erro, é utilizada a solução descrita em [7], este assumindo que a rotação incremental, num registro em tempo real, geralmente é muito pequena. Linearizando a rotação, aproximando $\cos\theta$ por 1 e $\sin\theta$ por 0, podemos reescrever a rotação completa (em todos os eixos) R da seguinte forma [1]:

Tabela 1 Tabela dos símbolos utilizados na descrição da minimização do erro

Símbolo	Descrição
R	Matriz de Rotação 3-D
t	Vetor de Translação 3-D
α	Ângulo de rotação no eixo X
β	Ângulo de rotação no eixo Y
γ	Ângulo de rotação no eixo Z
σ	Somatório para a métrica ponto-plano [5]
c'	Produto vetorial entre um ponto p e um vetor normal n_m
tr	Indicação de transposição em um vetor/matriz
r	$[\alpha\, \beta\, \gamma]^{tr}$
$\phi(w)$	$p_i(w) - p_m(w)) \cdot n_m(w)$

$$R = \begin{pmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ \beta & \alpha & 1 \end{pmatrix} \quad (2)$$

Com isso, podemos reescrever (1) da seguinte forma:

$$\begin{aligned} \sigma^2 &= \sum_{w=1}^{n} ||(p_i(w) - p_m(w)) \cdot n_m(w) \\ &+ = t \cdot n_m(w) + r \cdot c'(w)||^2 \end{aligned} \quad (3)$$

A partir da equação de minimização (3), um sistema linear na forma $Cx = b$ pode ser calculado:

$$C = \sum_{w=1}^{n} \begin{pmatrix} A_{cc}(w) & A_{cn}(w) \\ A_{nc}(w) & A_{nn}(w) \end{pmatrix} \quad (4)$$

sendo A definida por:

$$A_{ab}(w) = \begin{pmatrix} a_{w,x}b_{w,x} & a_{w,x}b_{w,y} & a_{w,x}b_{w,z} \\ a_{w,y}b_{w,x} & a_{w,y}b_{w,y} & a_{w,y}b_{w,z} \\ a_{w,z}b_{w,x} & a_{w,z}b_{w,y} & a_{w,z}b_{w,z} \end{pmatrix} \quad (5)$$

[1] A Tabela 1 apresenta o significado de cada símbolo utilizado durante a descrição do algoritmo de minimização de erro

$$x = [\alpha\, \beta\, \gamma\, t_x\, t_y\, t_z]^{tr} \tag{6}$$

$$b = -\sum_{w=1}^{n} \begin{pmatrix} c_x(w)\phi(w) \\ c_y(w)\phi(w) \\ c_z(w)\phi(w) \\ n_{i-1,x}(w)\phi(w) \\ n_{i-1,y}(w)\phi(w) \\ n_{i-1,z}(w)\phi(w) \end{pmatrix} \tag{7}$$

Aplicando a decomposição de Cholesky sobre a equação $Cx = -b$, pode-se encontrar a transformação T que define a relação entre as nuvens de pontos.

Após a utilização do ICP e o consequente alinhamento entre a nuvem de pontos adquirida no instante i e o modelo em reconstrução, obtêm-se a transformação T_i que define a pose da câmera no instante i, permitindo que os dois modelos sejam relacionados, e a nuvem de pontos seja integrada ao modelo reconstruído final.

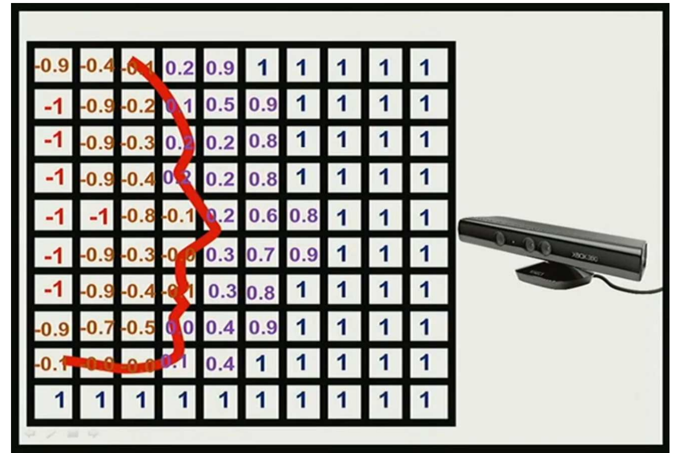

Figura 3 Visão lateral do modelo inserido na *grid* volumétrica. A SDF especifica a distância do *voxel* em relação ao modelo, sendo positiva se estiver posicionado à frente do modelo, e negativa na situação oposta. O modelo (cor vermelha) é definido a partir das posições onde ocorre o *zero-crossing*.

3.3 Reconstrução

Após a obtenção da matriz de transformação que define a pose da câmera, pode-se converter a nuvem de pontos V_i do sistema de coordenadas da câmera para o sistema de coordenadas global e vice-versa. Uma vez com a pose global, podemos integrar as diversas nuvens de pontos obtidas em um único modelo final.

3.3.1 Representação volumétrica

Ao invés de concatenarmos todas as nuvens de pontos com coordenadas globais em um único modelo, o que resultaria em uma nuvem com uma grande quantidade de pontos muito próximos entre si, é utilizada uma representação volumétrica baseada em [6]. Essa representação volumétrica, além de lidar com o problema da falta de decimação da nuvem de pontos, permite que seja realizada, de forma mais fácil, a triangulação do modelo reconstruído. Isso pode ser feito a partir de algoritmos como *Marching Cubes* [12].

Nessa representação, uma *grid* com *voxels* 3-D de tamanho pré-definido é utilizada para mapear as nuvens de pontos obtidas em uma dimensão específica. Cada *voxel* armazena a média ponderada de um variante da SDF [14] (*Signed Distance Function* - Função de distância com sinal), medido para cada nova nuvem de pontos a ser integrada.

A SDF especifica a distância do *voxel* em relação ao modelo. Esses valores são positivos, se o *voxel* estiver posicionado à frente do modelo, e negativos, na situação oposta. O modelo é definido a partir das posições onde ocorre o *zero-crossing*, em que os sinais das distâncias mudam (Figura 3).

Assim como em [9], nós usamos um variante da SDF, a TSDF (*Truncated SDF* - SDF Truncada), que apenas armazena o valor da distância para uma região próxima ao modelo atual [6]. Além da menor quantidade de processamento exigida por este variante, ele reduz a incerteza nos dados do modelo final reconstruído, tratando de forma eficiente as múltiplas medições feitas pelo dispositivo de captura [9]. Além disso, o mapa de profundidade D_i utilizado nesta fase, é um mapa de profundidade sem aplicação de filtro, uma vez que deseja-se integrar os pontos reais do modelo adquirido [13].

3.3.2 Integração volumétrica

A integração das diferentes nuvens de pontos adquiridas pelo *Kinect* pode ser feita a partir do Algoritmo 1.

O algoritmo de integração volumétrica pode ser descrito da seguinte forma:

Dada uma posição (x, y), o eixo Z é percorrido. *gridVolumeSize* é uma variável interna do algoritmo (linhas 1 até 3).

Localiza-se o *voxel* associado a posição atualmente percorrida (linha 4) e converte-se este do sistema de coordenadas da *grid* para o sistema de coordenadas global. g, v^g e o vetor *voxel* são variáveis internas do algoritmo (linha 5).

Converte-se então o ponto de coordenadas globais para coordenadas da câmera (linha 6) e o ponto é projetado num plano 2-D, permitindo a obtenção do *pixel* correspondente ao ponto v. As variáveis v e $pixel$ são criadas internamente, enquanto a variável T^{-1} representa um dado de entrada do algoritmo (linha 8).

Se existir um ponto real do modelo no *pixel* obtido (linha 9), calcula-se a distância entre um *voxel* de posição (x, y) e o ponto do mapa de profundidade D_i de mesma coordenada (cálculo do SDF). A condição feita na linha 12 verifica se o ponto do mapa de profundidade possui grau de incerteza abaixo de um valor pré-definido *trunc*. Caso esse ponto possua menor grau de incerteza do que *trunc*, efetua-se o cálculo do TSDF. D_i representa um dado de entrada do algoritmo,

Algoritmo 1 Integração volumétrica

1: **for** $x \leftarrow 1, gridVolumeSize.x$ **do**
2: **for** $y \leftarrow 1, gridVolumeSize.y$ **do**
3: **for** $z \leftarrow 1, gridVolumeSize.z$ **do**
4: $g \leftarrow voxel[x][y][z]$
5: $v^g \leftarrow$ converta g do sistema de coordenadas da $grid$ para o sistema de coordenadas global
6: $v \leftarrow T_i^{-1} v^g$
7: **if** v na visão da câmera **then**
8: $pixel \leftarrow$ projete v num plano 2-D
9: **if** $D_i(pixel).z > 0$ **then**
10: $t_i \leftarrow$ adquira o vetor de translação associado a nuvem de pontos do mapa D_i
11: $sdf_i \leftarrow D_i(pixel).z - v^g.z$
12: **if** $sdf < trunc$ **then**
13: **if** $sdf_i > 0$ **then**
14: $tsdf_i \leftarrow MIN(1, sdf_i/trunc)$
15: **else**
16: $tsdf_i \leftarrow MAX(-1, sdf_i/trunc)$
17: **end if**
18: $peso_i \leftarrow MIN(maxpeso, g.peso + 1)$
19: $tsdf^{avg} \leftarrow \frac{tsdf_{i-1}peso_{i-1} + tsdf_i peso_i}{peso_{i+1} + peso_i}$
20: $g.peso \leftarrow peso$
21: $g.tsdf \leftarrow tsdf$
22: **end if**
23: **else**
24: **break**
25: **end if**
26: **end if**
27: **end for**
28: **end for**
29: **end for**

sdf e t representam variáveis criadas internamente e $trunc$ é um parâmetro pré-definido.

O cálculo do SDF é feito a partir da distância, no eixo Z, entre o ponto $D_i(pixel)$ e o *voxel* v^g (linha 11). Diferente de [9], nós calculamos o SDF considerando apenas o eixo Z, uma abordagem mais simples e com resultados similares do que foi feito em [13], em que o SDF é calculado considerando os 3 eixos (X, Y, Z) e em seguida é dividido por $||\lambda||$, sendo λ um vetor com valores $(v^g.x, v^g.y, 1)$ para os eixos X, Y e Z respectivamente.

Normaliza-se o SDF para um TSDF (linhas 13 até 17) e atribui-se um peso ao TSDF calculado (linha 18).

Pondera-se o TSDF obtido com o TSDF calculado na iteração anterior (linha 19). $peso_i$ é uma variável criada internamente.

O peso e o TSDF calculados são armazenados no *voxel* g (linhas 20 e 21) e caso a *grid* não tenha sido percorrida totalmente, retorna-se ao passo 1.

Esse algoritmo não possui nenhum dado de saída.

3.3.3 Raycasting

O *raycasting* é feito sobre a *grid* volumétrica para extrair uma imagem de alta qualidade representando o modelo inserido na *grid* [9]. O Algoritmo 2 realiza o *raycasting* sobre a *grid* e é baseado em [14].

Algoritmo 2 Raycasting sobre a *grid* volumétrica

1: **for** $pixel \leftarrow 0, 640 * 480$ **do**
2: $x \leftarrow pixel \,\%\, 640$
3: $y \leftarrow pixel/640$
4: $ray^{inicio} \leftarrow$ converta *voxel* $[x][y][0]$ do sistema de coordenadas globais em coordenadas da *grid*.
5: $ray^{proximo} \leftarrow$ converta *voxel* $[x][y][1]$ do sistema de coordenadas globais em coordenadas da *grid*.
6: $ray^{direcao} \leftarrow$ normalize $(ray^{proximo} - ray^{inicio})$
7: $ray^{tamanho} \leftarrow 0$
8: $RI_i(pixel) \leftarrow (x, y, 0)$
9: $g \leftarrow$ extraia o primeiro *voxel* na direção $ray^{direcao}$
10: $tsdf \leftarrow g.tsdf$
11: **while** *voxel* g contido na *grid* **do**
12: $ray^{tamanho} \leftarrow raycaster^{tamanho} + 1$
13: $g \leftarrow$ extraia o próximo *voxel* na direção $ray^{direcao}$
14: **if** g não está contido na *grid* **then**
15: **break**
16: **end if**
17: $tsdfanterior \leftarrow tsdf$
18: $tsdf \leftarrow g.tsdf$
19: **if** $tsdfanterior < 0$ e $tsdf > 0$ **then**
20: **break**
21: **end if**
22: **if** $tsdfanterior > 0$ e $tsdf < 0$ **then**
23: $g \leftarrow$ converta *voxel* $(g.x - 0.5, g.y - 0.5, g.z - 0.5)$ do sistema de coordenadas da *grid* em coordenadas globais.
24: $RI_i(pixel) \leftarrow g$
25: **end if**
26: **end while**
27: **end for**

O algoritmo de *raycasting* pode ser descrito da seguinte forma:

Seleciona-se um novo *pixel* para o *raycasting* sobre a *grid* volumétrica (linhas 1 até 3). Os valores 640 e 480 representam as dimensões da imagem de saída.

Inicializa-se o raio de acordo com as coordenadas do *pixel* selecionado, sendo que ele percorrerá a *grid* no eixo Z (linhas 4 até 7). O *pixel* é convertido do sistema de coordenadas globais para o sistema de coordenadas da *grid*. A variável *ray* é interna ao algoritmo.

Atribui-se o valor mínimo para a imagem RI_i, que representa a imagem após o *raycasting*. Isso é feito como precaução para o caso de não haver posição *zero-crossing* durante a passagem do raio *ray* (linha 8). RI_i é o único dado de saída do algoritmo.

Extrai-se o primeiro *voxel* g na direção do raio *ray* (linha 9) e armazena-se o TSDF associado ao *voxel* g (linha 10).

Se o *voxel* g estiver contido na *grid*, incrementa-se o tamanho do raio e captura-se o novo *voxel* g percorrido pelo raio de novo tamanho. Se o novo *voxel* não estiver contido na *grid*, encerra-se a passagem do raio pelo *pixel* selecionado (linhas 11 até 16).

Se localizada uma posição de *zero-crossing*, interpola-se trilinearmente o *voxel* g, convertendo-o do sistema de coordenadas da *grid* para o sistema de coordenadas global, e g é armazenado na imagem RI_i (linhas 17 até 24). A subtração do *voxel* g por 0.5 permite acesso ao centro do *voxel*.

O algoritmo de *raycasting* também é utilizado para extração da nuvem de pontos do modelo V_m em uma projeção perspectiva. Essa nuvem de pontos é utilizada como nuvem base para o registro *frame-model*.

3.3.4 Extração da nuvem de pontos parcial/completa

Para a extração completa do modelo tridimensional reconstruído, faz-se necessária a utilização de um algoritmo baseado somente em CPU. A exibição da composição no modelo **não** é feita em tempo real. A exibição da imagem extraída pelo *raycasting* representará o modelo em reconstrução.

O algoritmo pode ser resumido da seguinte forma: Para cada coordenada (x, y) da *grid* volumétrica, é lançado um raio ao longo do eixo Z, e, para cada posição em que houver um *zero-crossing*, será extraído um ponto correspondente a interpolação trilinear daquela posição.

4 Unidade de Processamento Gráfico - GPU

Os microprocessadores das CPUs tiveram rápido crescimento, em termos de desempenho, por mais de 2 décadas. Contudo, a partir de 2003, esse crescimento vem reduzindo cada vez mais devido às limitações da própria tecnologia do equipamento. A alternativa para essa redução de evolução da CPU foi realizar a construção de múltiplos processadores, ao invés de somente um único, como era feito até então [10].

A área de multiprocessamento pode ser dividida a partir de dois equipamentos: CPUs *multi-core* e GPUs. A diferença principal entre essas duas tecnologias é que, enquanto a primeira tem como objetivo tornar mais veloz a execução de programas/algoritmos sequenciais, a segunda tem como objetivo tornar mais veloz a execução de aplicações/algoritmos paralelos [10].

Existem duas linguagens de programação bastante utilizadas para manipulação das *threads* disponibilizadas pela GPU: CUDA [19] e OpenCL [22]. A segunda versão do sistema foi desenvolvida com a linguagem OpenCL, enquanto a terceira versão do sistema foi desenvolvida com a linguagem CUDA, já utilizada para outros algoritmos pelo PCL.

A paralelização feita pelas versões do sistema com GPU pode ser descrita da seguinte forma:

- Na fase de aplicação do filtro bilateral, cada *thread* foi responsável pelo cálculo do novo valor de profundidade para cada *pixel* do mapa de profundidade D_i.
- Na fase de alinhamento, cada *thread* foi responsável pela seleção, associação e rejeição de pares correspondentes para cada *pixel* do mapa de profundidade D_i;
- Na fase de reconstrução, cada *thread* foi responsável por calcular o TSDF e o seu peso associado para cada *voxel* da *grid*;
- Para o algoritmo de *raycasting*, cada *thread* foi responsável por calcular cada *pixel* de saída da imagem RI_i;

Como a maioria dos algoritmos em CPU foram desenvolvidos pensando-se na paralelização, tornou-se muito mais fácil a adaptação para execução em GPU.

5 Modelagem do Sistema

Nesta seção é apresentada a documentação da primeira e segunda versões do sistema a partir dos seguintes artefatos: arquitetura do sistema, diagrama de classe, casos de uso e diagrama de sequência, respectivamente. A maioria dos aspectos de modelagem da terceira versão do sistema estão presentes nas duas primeiras versões. As diferenças entre as três versões são descritas ao longo desta seção.

5.1 Arquitetura do Sistema

O sistema foi subdividido de forma que cada grupo de funcionalidades possuísse um modulo específico que as implementasse e disponibilizasse. A Figura 4 mostra o diagrama componente-conector do sistema.

O componente *ImageProcessing* é responsável por lidar com todas as operações que são feitas com os mapas de profundidade. Esse componente carrega um mapa de profundidade a partir do componente *OpenCV* [4], que também é responsável por aplicar o filtro bilateral sobre a imagem.

O componente *CloudProcessing* é responsável por processar e manipular a nuvem de pontos. Este componente é responsável pela conversão da imagem para uma representação tridimensional, alinhamento das nuvens sequenciais, agrupamento iterativo dessas nuvens para a obtenção de um único modelo tridimensional da superfície e renderização do modelo acumulado. Este componente utiliza o componente *Eigen* [20] que é responsável pelas funções referentes às operações de cálculos de matrizes.

O componente *View* permite visualizar a superfície tridimensional capturada pelo *Cloud Processing* e os mapas de profundidade. A interface provida *QGLWidget* somente está presente nas duas primeiras versões do sistema. Na terceira versão, a interface provida é *PCLVisualizer*.

O conector *signal and slot* é um conector que permite a comunicação entre dois componentes sem invocação explícita de método entre eles.

5.2 Diagrama de classe

Assim como foi mencionado na Introdução deste trabalho, o sistema de reconstrução tridimensional foi desenvolvido

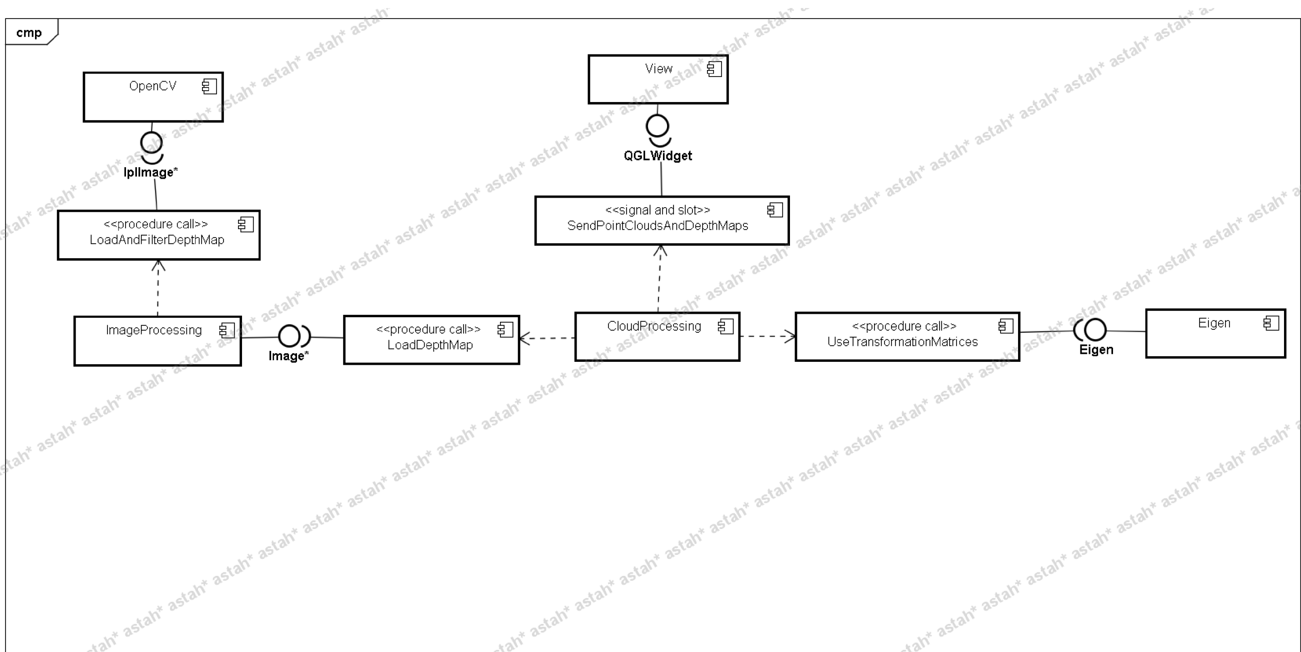

Figura 4 Visão Geral da Arquitetura do sistema.

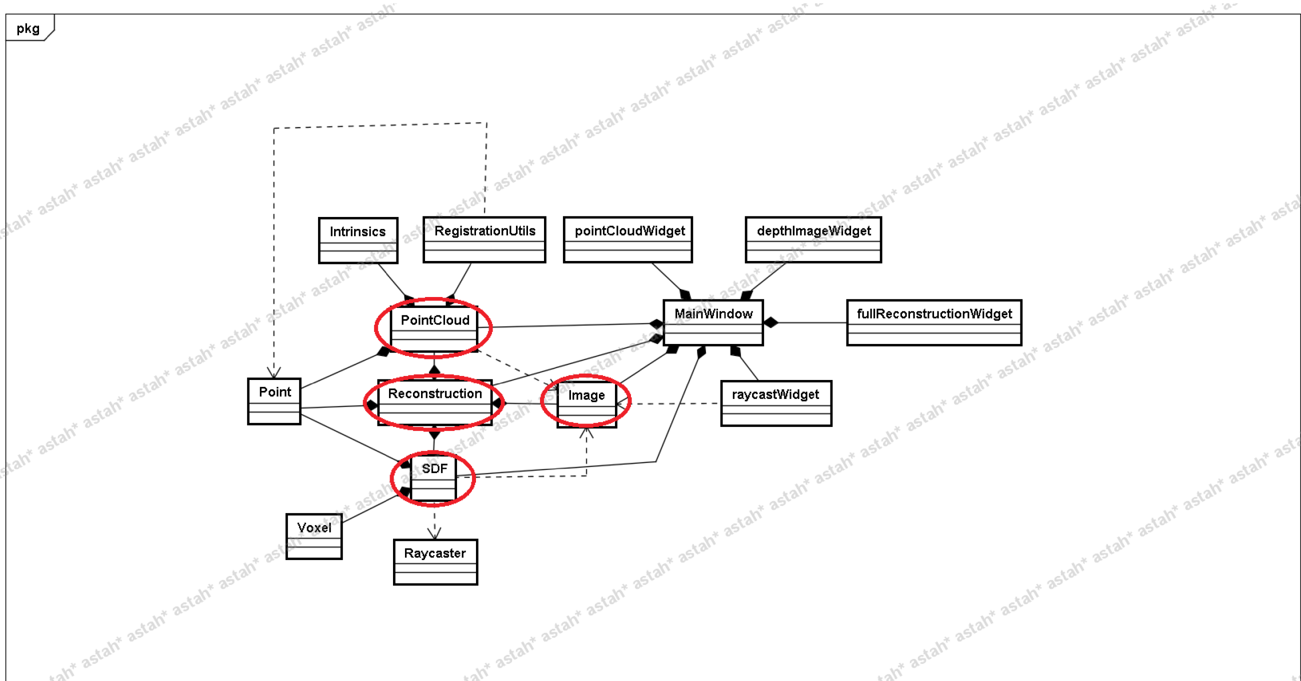

Figura 5 Diagrama de Classes Simplificado (numa visão sem métodos e atributos). As classes localizadas nas áreas destacadas são compartilhadas entre todas as versões do sistema

em três versões. As duas primeiras versões foram desenvolvidas baseadas no mesmo diagrama de classes, enquanto a terceira versão foi desenvolvida baseada no projeto da biblioteca PCL, contendo somente as classes *PointCloud*, *Reconstruction*, *SDF* e *Image* em comum com as duas outras versões.

As classes das duas primeiras versões podem ser vistas na Figura 5 e serão apresentadas a partir de seus principais métodos e atributos:

– As classes *Point*, *Intrinsics*, *Voxel* e *Raycaster* não possuem nenhum método. A classe *Point* representa um ponto 3-D. A classe *Intrinsics* representa o conjunto de parâmetros intrínsecos do *Kinect*. A classe *Voxel* representa

um *voxel* inserido na *grid*. A classe *Raycaster* representa o raio que atravessará a *grid* na fase de *raycasting*.

- A classe *MainWindow* é responsável pela inicialização do sistema. Esta classe serve como ponte entre a interface gráfica e as classes de negócio do sistema.
- As classes *DepthImageWidget* e *RaycastWidget* são responsáveis pela visualização do mapa de profundidade sendo processado e da imagem obtida via *raycast*.
- A classe *PointCloudWidget* é responsável por exibir as nuvens de pontos base e alvo processadas durante a fase de alinhamento. Essa exibição é feita a partir do *OpenGL* [30] integrado ao *Qt*.
- A classe *FullReconstructionWidget* é responsável por exibir a nuvem de pontos extraída da *grid* no sistema de coordenadas globais, permitindo que seja possível acompanhar, em tempo real, o alinhamento entre os *frames* sucessivos. Assim como na classe *PointCloudWidget*, essa exibição também é feita a partir do *OpenGL* [30] integrado ao *Qt*.
- A classe *Image* armazena o mapa de profundidade obtido, ou mesmo cria um novo mapa de profundidade, sendo também responsável pela aplicação do filtro sobre o mesmo. O método *convert* sem parâmetros converte um mapa de profundidade obtido para uma representação interna, aplicando o filtro (quando necessário) sobre o mapa. O método *convert* com parâmetros permite a criação de uma imagem a partir de dados externos.
- A classe *Reconstruction* é a classe principal do sistema, sendo responsável por estruturar todo o processo de reconstrução tridimensional. Essa classe, uma vez criada pela classe *MainWindow*, recebe como parâmetros todos os objetos que serão necessários para a reconstrução tridimensional, e, através do método *staticMethod()*, faz as iterações do sistema. Essa classe também é responsável por enviar os dados pós-processados do sistema para as classes de interface gráfica através das primitivas *signal* e *slot* do *Qt*.
- A classe *SDF* é responsável pelas fases de integração volumétrica e *raycasting* (em conjunto com a classe *Raycaster*), tendo como principal atributo a própria *grid* (conjunto de *voxels*). O método *integrateTsdfVolumeBasedOnKinectFusion* faz a integração volumétrica baseada em [9], onde se dá prioridade à reconstrução dinâmica: para cada *pixel* da *grid*, mantém-se no máximo um ponto do modelo real; enquanto o método *integrateTsdfVolumeBasedOnFullReconstruction* faz a integração volumétrica baseada na nossa adaptação de [13], em que se dá prioridade à reconstrução completa: para cada *pixel* da *grid*, a quantidade máxima de pontos do modelo real que pode ser inserido na *grid* é definida pelo usuário. Além disso, temos os métodos *raycastTsdfVolume* que faz o algoritmo de *raycasting* sobre a *grid*, e o método *extractPointCloudFromGrid*, que funciona apenas em CPU e faz a extração completa da superfície implícita na *grid* volumétrica.
- A classe *RegistrationUtils* é responsável por disponibilizar uma série de implementações relativas à fase de alinhamento. Esta classe armazena principalmente os algoritmos que permitem a associação de correspondências e os algoritmos de estimativa de transformação, no nosso caso, o algoritmo baseado na decomposição de Cholesky.
- A classe *PointCloud* representa uma nuvem de pontos no sistema. Ela armazena dados como: conjunto de pontos da nuvem, vetores normais de cada ponto, matriz de transformação 3-D que permite conversão de coordenada da câmera para global e vice-versa. Além disso, a classe *PointCloud*, através do método *alignPointClouds* realiza o registro entre duas nuvens de pontos. Isso é feito em conjunto com a classe *RegistrationUtils* descrita acima.

5.3 Casos de Uso

Existem apenas dois casos de uso para o sistema: O usuário pode adicionar uma nuvem de pontos ao sistema, o que corresponderia à inicialização do sistema com o *Kinect*, ou então, de fato, a adição de uma nuvem de pontos ao sistema, caso o sistema esteja realizando a reconstrução de um objeto a partir de mapas de profundidade gravados em disco. O usuário também pode optar por visualizar a nuvem de pontos reconstruída completa, ou somente sua visão perspectiva, como mencionado na seção 3.3.4.

5.4 Diagrama de Sequência

O diagrama de sequência do sistema de reconstrução tridimensional para o primeiro caso de uso pode ser visto a partir da Figura 7. A partir da ação do usuário de adicionar uma nova nuvem de pontos ao sistema, uma instância da classe *MainWindow* invoca o método *staticMethod* do objeto *reconstruction*. Uma vez em funcionamento, o objeto *reconstruction* carrega o mapa de profundidade atual com e sem filtro, a partir dos objetos *image* e *imageWithoutFilter* respectivamente. A partir disso, temos as seguintes condições:

- Se o mapa de profundidade for o primeiro adquirido, carregamos a nuvem de pontos alvo a partir dele, pois queremos alinhar o mapa de profundidade adquirido no instante i com o mapa adquirido no instante $i-1$. Após isso, inserimos a nuvem de pontos na *grid*. Isso tudo está encapsulado no método *renderBasedOnKinectFusion* da classe *SDF*.
- Se o mapa de profundidade for o segundo adquirido pelo sistema, carregamos a nuvem de pontos base a partir dele. Uma vez com as nuvens de pontos V_i e V_{i-1}, chamamos o método *alignPointClouds* do objeto *basePC* passando

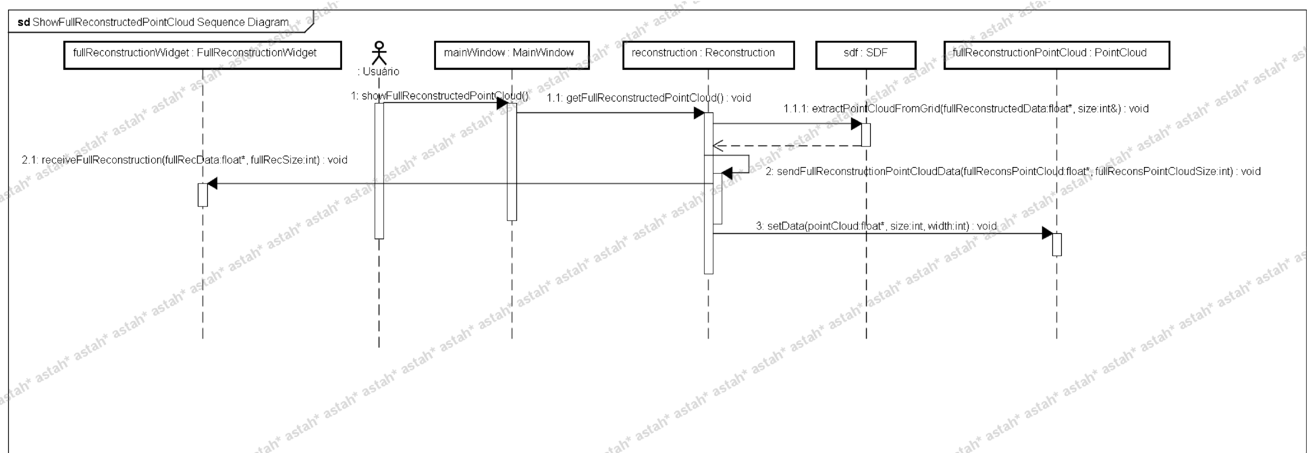

Figura 6 Diagrama de sequência para o caso de uso "visualizar nuvem de pontos completa".

como parâmetro a nuvem alvo. Esse método que encapsula o algoritmo de alinhamento (ICP), tenta encontrar a matriz de transformação que melhor define a relação entre as duas nuvens de pontos. Se o algoritmo não conseguir reduzir o erro inicial entre as duas nuvens de pontos, consideramos que houve um erro durante o processamento do alinhamento, e não integramos a nuvem de pontos base na *grid*. Caso contrário, integramos a nova nuvem de pontos na *grid* e realizamos novamente o *raycasting* sobre a *grid*.

- Em um caso diferente dos anteriores, nós carregamos a nuvem de pontos alvo a partir da base ($V_{i-1} = V_i$), e carregamos uma nova nuvem de pontos base a partir do mapa de profundidade atual carregado ($V_i =$ nova D_i). Neste caso, nós procedemos como no caso anterior, em que a função *alignPointClouds* é invocada para realizar o alinhamento, e se o algoritmo for bem sucedido, a nuvem de pontos é inserida na *grid*.

O diagrama de sequência referente ao segundo caso de uso pode ser visto a partir da Figura 6, em que o usuário pede ao objeto *mainWindow* para visualizar a nuvem de pontos completa, e este objeto, por sua vez, pede ao objeto *reconstruction* que extraía do objeto *sdf* a nuvem de pontos completa. Por fim, essa nuvem é enviada para o objeto *fullReconstructionWidget*, para posterior visualização na tela a partir do *OpenGL*.

6 Resultados e Discussões

Nessa seção, serão descritos e avaliados todos os testes realizados sobre os três sistemas.

6.1 Ambiente de Testes

Todos os testes nas duas primeiras versões do sistema foram feitos com modelos sintéticos, enquanto os testes para a terceira versão do sistema foram feitos com mapas de profundidade adquiridos diretamente do *Kinect*. Os testes realizados nas duas primeiras versões do sistema foram feitas em um *notebook* com sistema operacional *Windows* 7, *Intel Core* i5-2430M CPU @ 2.40GHZ, 4GB de memória RAM, placa de vídeo NVIDIA GTX 525M, enquanto os testes realizados com o *Kinect* foram feitos em um computador *desktop* com sistema operacional *Windows* XP, *Intel Pentium III Xeon* CPU @ 2.33GHZ, 3.49GB de memória RAM, com placa de vídeo NVIDIA GeForce GTX 450.

Os testes com diferentes ambientes não prejudicaram nenhuma das considerações feitas neste trabalho, tendo em vista que nenhum teste foi realizado com as três versões do sistema.

Nos testes feitos com o *Kinect*, o dispositivo de captura foi rotacionado em torno do objeto a ser reconstruído e o tempo total de processamento por *frame* foi em torno de 90 ms.

Os resultados e discussões para cada teste serão descritos mais detalhadamente nas próximas subseções.

6.2 Primeiro Teste: Comparação por tempo de processamento entre CPU e GPU

Este teste teve como objetivo avaliar a otimização do tempo de processamento com a GPU. Nós comparamos o desempenho de alguns algoritmos paralelizáveis em CPU e GPU, além do tempo total de processamento por *frame* do sistema. A primeira e segunda versões do sistema foram utilizadas para esse teste.

O modelo utilizado para o teste corresponde a um modelo poligonal de um carro (Figura 8), sendo que a densidade da

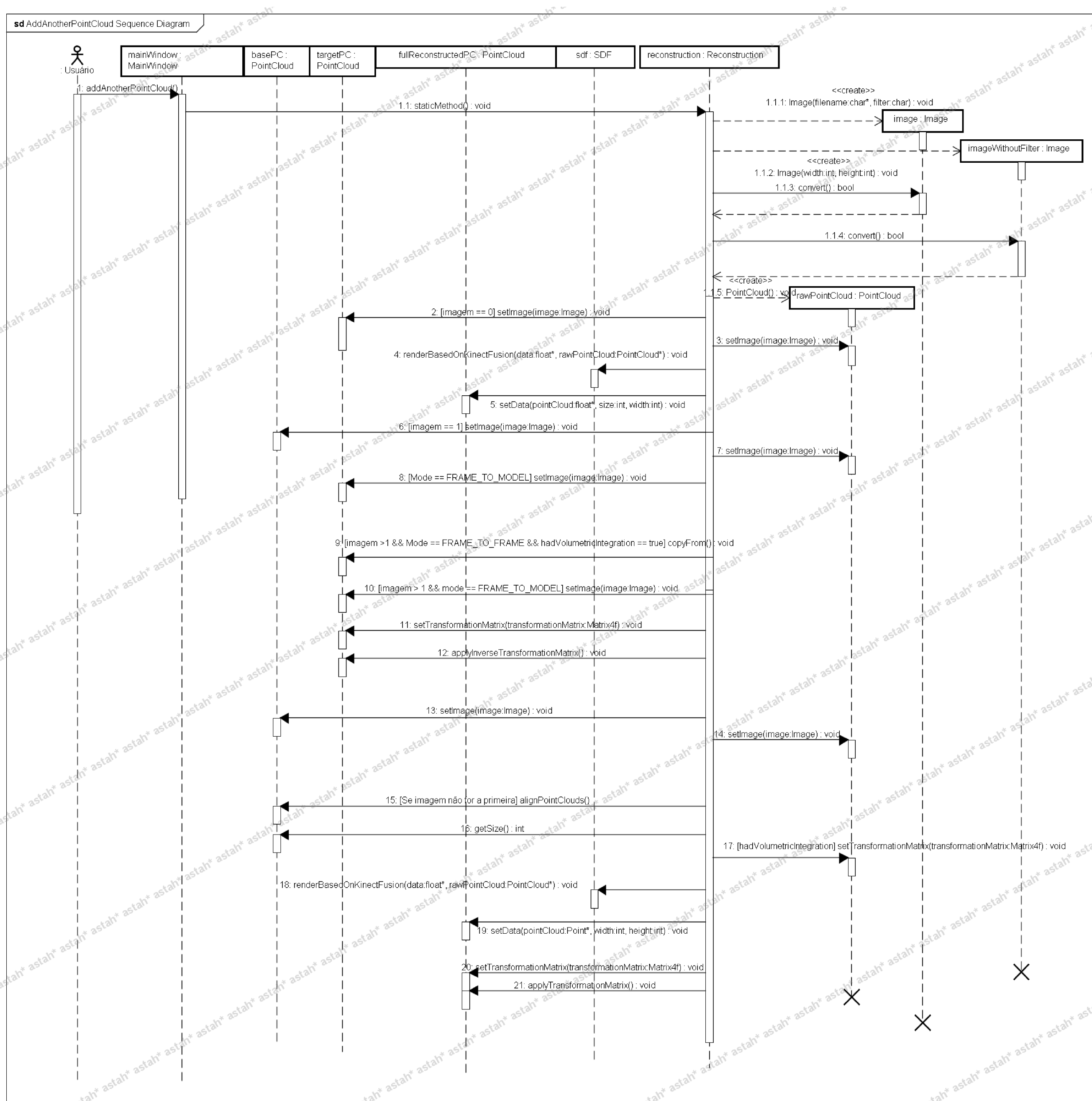

Figura 7 Diagrama de sequência para o caso de uso "adicionar nuvem de pontos".

nuvem de pontos foi reduzida a partir de sucessivas filtragens dos pontos vizinhos.

Para esse teste, CPU e GPU foram comparados a partir dos algoritmos: *raycasting* (Figura 9), integração volumétrica (Figura 10), associação projetiva de dados (Figura 11), além do tempo total de processamento por *frame* (Figura 12), que engloba tempo de pré-processamento e atualização das interfaces de visualização do sistema.

Dentre todos os três algoritmos testados, o algoritmo de *raycasting* foi o que apresentou maior tempo de execução, e também foi o único que, à medida em que a densidade da nuvem de pontos decresceu, o tempo de processamento aumentou. Isso aconteceu pois, quanto menor a quantidade de posições *zero-crossing* na *grid*, maior a quantidade de raios que percorreram todo o eixo Z para sua posição (x,y) atribuída. Lembrando-se de que, se um raio passa por uma posição *zero-crossing*, ele extraí o ponto do modelo real correspondente a esta posição e não percorre mais a *grid*. A redução média de processamento da CPU para a GPU ficou em torno de 98,3% para este algoritmo.

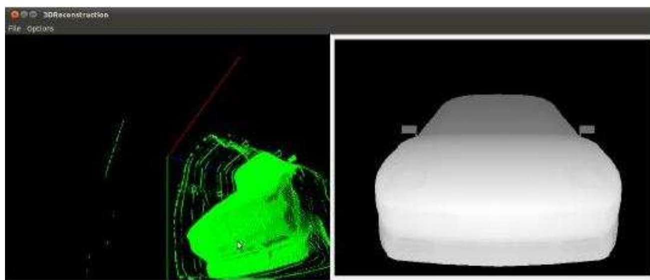

Figura 8 Modelo poligonal utilizado para o teste de comparação por tempo de processamento entre CPU e GPU.

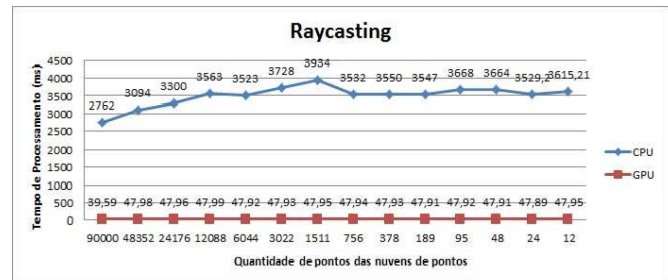

Figura 9 Desempenho do algoritmo *raycasting* em CPU e GPU.

O algoritmo de integração volumétrica teve um comportamento diferente dos outros algoritmos testados, pois, quanto menor a densidade da nuvem de pontos, muito menor foi seu tempo de execução em CPU, de tal forma que o tempo de execução a partir de uma determinada densidade foi menor do que em GPU. Uma *thread* da CPU possui menor tempo de execução em relação a uma *thread* da GPU, uma vez que ela executa rapidamente sobre todo o sistema, e não somente em um trecho de código, como no caso das *threads* da GPU. O que faz a execução da GPU ser mais rápida do que a execução em CPU é o fato de que a GPU possui uma quantidade de *threads* muito maior do que a CPU, e que só executam um trecho bem específico do sistema. Contudo, para esse algoritmo, mesmo com a grande quantidade de *threads* utilizadas em GPU paralelamente, a única *thread* em execução na CPU foi mais veloz para uma nuvem de pontos com densidade baixa.

O algoritmo de associação projetiva de dados, por ser altamente paralelizável, apresentou grande redução do tempo de processamento quando executado com o auxílio da GPU. A redução variou de 83% a 99,36%, relativas respectivamente a menor e maior densidade testadas.

Já para o tempo total de processamento por *frame*, a execução em CPU com a GPU conseguiu reduzir em torno de 90% o tempo de execução feito somente pela CPU. Vale lembrar que, mesmo com a redução de 90%, o tempo de execução com GPU ainda estava muito alto, chegando a mais de 700 *ms* para a nuvem com 90000 pontos. Isso se deve ao fato de que: as imagens sintéticas eram carregadas diretamente do disco rígido; algumas nuvens de pontos foram mantidas e processadas, sendo que estas não existiriam na versão final (terceira versão) do sistema e o algoritmo de estimativa da transformação baseado na decomposição de Cholesky não foi paralelizado.

6.3 Segundo Teste: Influência do filtro bilateral para a reconstrução do modelo

Este teste teve como objetivo verificar a influência do filtro bilateral sobre o modelo reconstruído. Para isso, a terceira versão do sistema foi utilizada.

Figura 10 Desempenho do algoritmo de integração volumétrica em CPU e GPU.

Figura 11 Desempenho do algoritmo de associação projetiva de dados em CPU e GPU.

Apesar de a reconstrução ser realizada sobre o mapa de profundidade não-filtrado, o registro é feito sobre o mapa de profundidade pós-filtro bilateral. A aplicação do filtro bilateral permitiu melhor qualidade para a reconstrução do modelo, sendo que o ICP falhou poucas vezes para pequenas rotações. Já para as situações em que o filtro não foi aplicado, o ICP falhou várias vezes mesmo para pequenas rotações feitas com o *Kinect*. Contudo, se o objetivo fosse reconstrução instantânea para superresolução do modelo, poderia-se utilizar o modelo sem filtro (Figura 13), ainda que este mantivesse uma quantidade maior de buracos e pontos com descontinuidade.

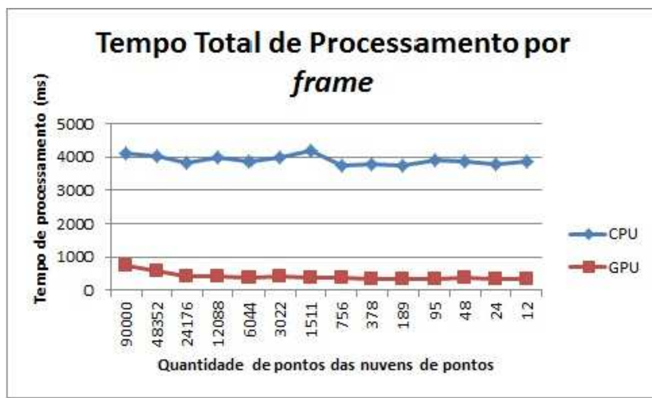

Figura 12 Tempo total de processamento por *frame* em CPU e GPU.

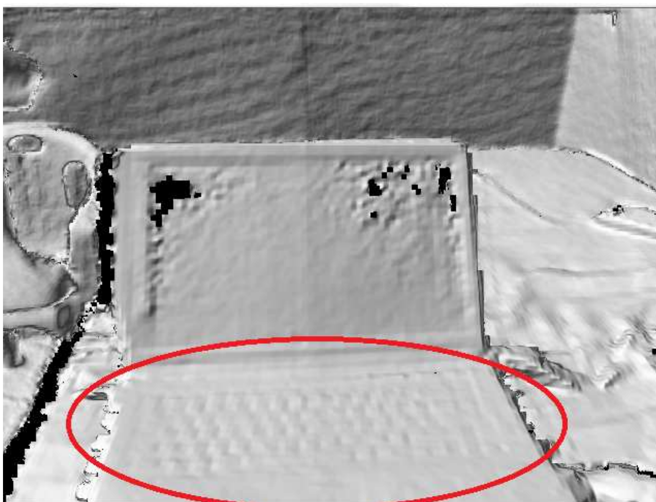

Figura 13 Modelo com alta qualidade reconstruído a partir de somente uma visão, sem aplicação do filtro bilateral. O teclado do *notebook* não possui nenhum buraco, ao contrário da tela do mesmo.

6.4 Terceiro Teste: Eficiência do algoritmo de alinhamento x Ângulo de variação entre *frames*

Este teste teve como objetivo avaliar a eficiência do algoritmo de alinhamento utilizado na medida em que o ângulo de variação entre os *frames* foi aumentado. Somente a primeira versão do sistema foi utilizada para esse teste.

Para esse teste, foram utilizados três modelos sintéticos para avaliar o variante do algoritmo ICP utilizado para aplicações que funcionam em tempo real. O objetivo do teste é avaliar, para cada um dos modelos, qual o ângulo máximo de variação em que o ICP consegue estimar a transformação rígida corretamente.

Assim como feito em [16], foram escolhidos três modelos que possuem diferentes características: O *Bunny* (Figura 14) representa um modelo com altas curvaturas, geometria relativamente suave e consiste em modelo fácil para alinhamento para muitos variantes do ICP. O modelo *fractal*, semelhante a uma árvore em que os nós são semelhantes a árvore original (Figura 15), representa um modelo com muitos níveis de detalhe, apesar de possuir poucos pontos (em torno de 4000 pontos). O plano (Figura 16) foi extraído de um cubo e não possui quase nenhum detalhe. O plano é considerado um modelo difícil para alinhamento, e muitos variantes não conseguem reduzir o erro inicial medido para esse modelo, mesmo quando o ângulo de variação é muito pequeno [16]. Apesar de esses modelos não incluírem todos os tipos existentes, eles representam uma grande classe de objetos.

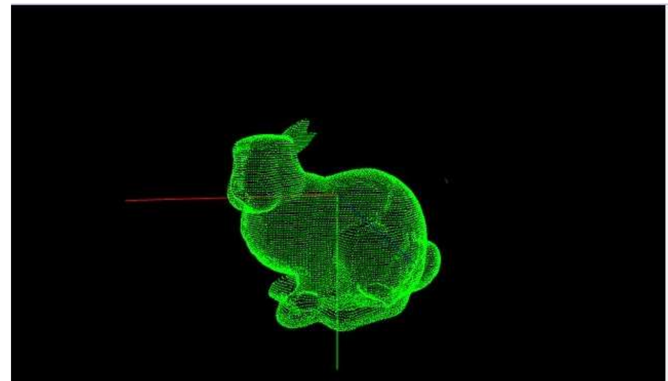

Figura 14 Modelo *Bunny* utilizado no terceiro teste.

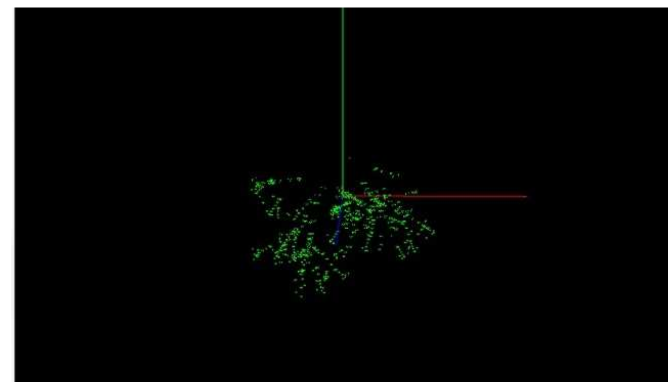

Figura 15 Modelo *Fractal* utilizado no terceiro teste.

O motivo da utilização de modelos sintéticos nesse teste é que nós conhecemos a transformação correta que permite o alinhamento das nuvens de pontos. Isso permite que o algoritmo de alinhamento seja avaliado tanto em relação à redução do erro inicial medido quanto em relação ao ângulo de rotação encontrado pelo algoritmo. Nesse teste, os modelos foram rotacionados em relação ao eixo Y, e o erro médio foi calculado a partir do RMS (*Root Mean Square* - Valor Quadrático Médio), definido por (8):

$$RMS = \sqrt{\frac{1}{n} * \sigma} \qquad (8)$$

sendo σ o somatório definido na Equação 1 e n a quantidade de pontos utilizada.

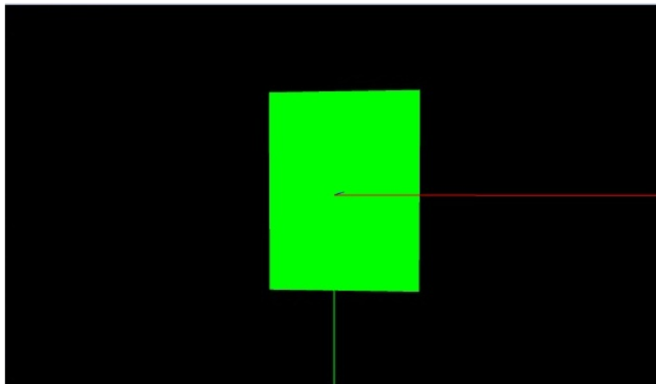

Figura 16 Modelo de um cubo utilizado no terceiro teste.

Figura 18 Ângulo de variação medido pelo algoritmo de alinhamento para o modelo plano da Figura 16. O variante do ICP conseguiu estimar o ângulo de variação corretamente até 8 graus de variação.

6.4.1 Análise para o modelo plano

O variante do ICP conseguiu identificar o ângulo correto até a variação de 8 graus entre as nuvens de pontos base e alvo (Figura 18). Isto também pode ser visto na Figura 17, em que a diferença entre o erro inicial e final decresceu muito a partir de 8 graus. Contudo, mesmo não identificando corretamente o ângulo de variação, o algoritmo de alinhamento conseguiu reduzir mais de 50% do erro até a variação de 13 graus. Diferente de outros variantes, o variante utilizado realiza a fase de associação de correspondências considerando como pares correspondentes, pontos de mesma coordenada (x, y). Como pode-se ver numa visão superior do modelo plano, na Figura 19, a associação de correspondências no mesmo *pixel* associa corretamente os pares correspondentes, permitindo que o ângulo de rotação seja corretamente identificado. A partir da diferença de 8 graus, a quantidade de correspondências associadas incorretamente aumenta, de forma que o algoritmo não consegue estimar a transformação corretamente (Figura 20).

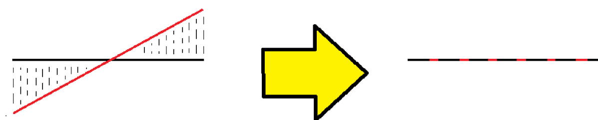

Figura 19 Visão superior das nuvens de pontos base (vermelha) e alvo (preta) para 7 graus de variação. As retas tracejadas indicam a associação dos pontos correspondentes.

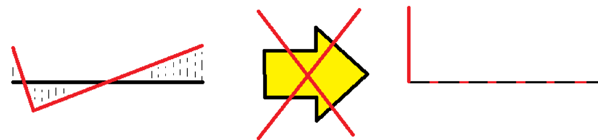

Figura 20 Visão superior das nuvens de pontos base (vermelha) e alvo (preta) para 15 graus de variação. As retas tracejadas indicam a associação dos pontos correspondentes.

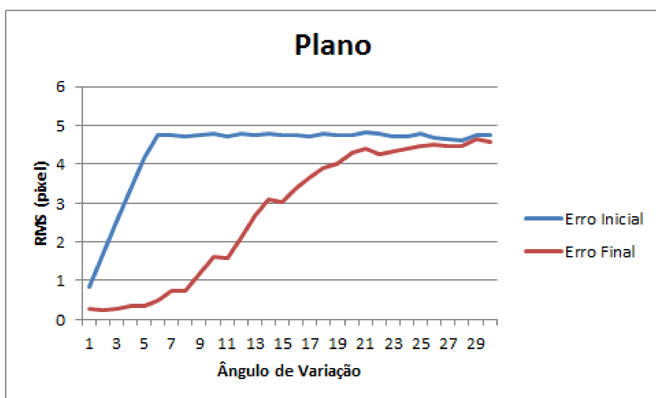

Figura 17 RMS medido para o modelo plano da Figura 16.

6.4.2 Análise para o Bunny

Para o modelo *Bunny*, o variante do ICP conseguiu identificar o ângulo correto apenas para pequenas rota-ções (menores do que 2 graus) (Figura 22). Já o gráfico que exibe a redução do erro médio mostra que o algoritmo conseguiu reduzir mais de 50% do erro médio até a variação de 3 graus entre os modelos (Figura 21).

A associação projetiva de pontos possui comportamento mais crítico em modelos com alta curvatura, como pode ser visto na Figura 23. Mesmo para pequenas variações, a quantidade de correspondências incorretas aumenta de forma considerável, impedindo um alinhamento correto por parte do ICP.

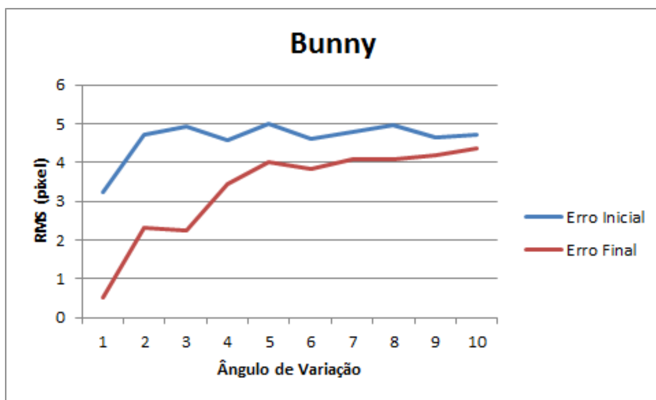

Figura 21 RMS medido para o *Bunny* da Figura 14.

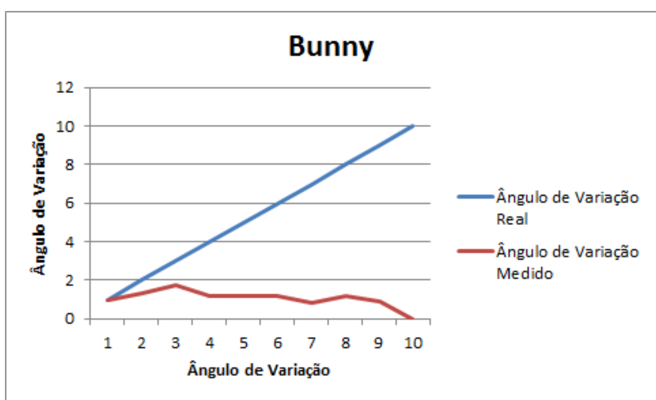

Figura 22 Ângulo de variação medido pelo algoritmo de alinhamento para o modelo *Bunny* da Figura 14. O variante do ICP conseguiu estimar o ângulo de variação correto somente para ângulos de variação abaixo de 1 grau.

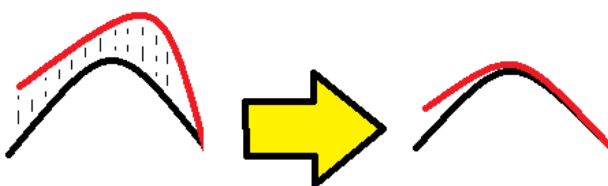

Figura 23 Visão superior das nuvens de pontos base (preta) e alvo (vermelha) para uma pequena variação de grau entre os modelos. As retas tracejadas indicam a associação dos pontos correspondentes.

6.4.3 Análise para o modelo Fractal

Para o modelo *Fractal*, o variante do ICP obteve um comportamento diferente em relação aos outros modelos. O algoritmo de alinhamento foi robusto até a variação de 2 graus (Figuras 24 e 25), contudo, para uma variação acima de 7 graus, o variante do ICP teve um comportamento diferenciado, aumentando o erro final em relação ao erro inicial medido, e, para a variação de 8 graus, o ICP não conseguiu estimar uma transformação rígida. Isso se deve ao fato de que o modelo *Fractal*, dentre todos os modelos, é o que possui a menor quantidade de pontos, e eles estão bem distribuídos (e espacialmente distantes) ao longo do modelo. A partir do momento em que há uma grande diferença de rotação entre as nuvens de pontos base e alvo, a quantidade de correspondências realizadas pela associação projetiva de dados reduz bastante, sendo que a maioria das correspondências realizadas sejam incorretas e façam com que o ICP estime uma transformação rígida incorreta.

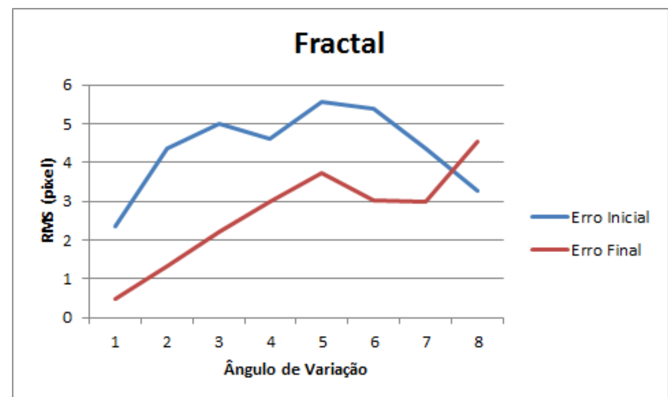

Figura 24 RMS medido para o *Fractal* da Figura 15.

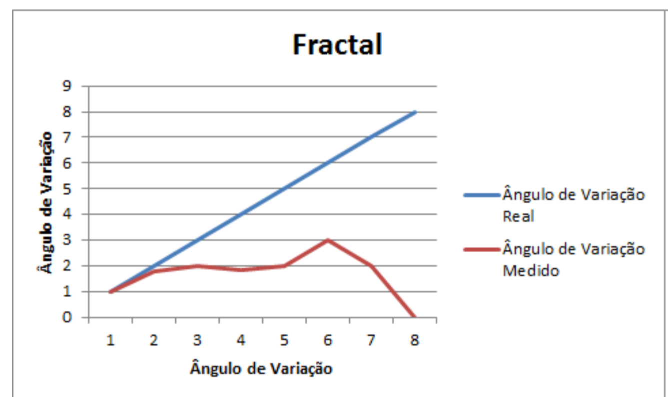

Figura 25 Ângulo de variação medido pelo algoritmo de alinhamento para o modelo *Fractal* da Figura 15. O variante do ICP conseguiu estimar o ângulo de variação correto somente para ângulos de variação abaixo de 2 graus.

6.5 Quarto Teste: Qualidade do Modelo Reconstruído

Assim como feito em [13] e [28], o sistema proposto será validado quanto a qualidade do modelo reconstruído. A área da imagem destacada na Figura 13 mostra que o sistema proposto reconstrói modelos de alta qualidade mesmo sem

a aplicação do filtro bilateral para facilitar o alinhamento e consequente integração das nuvens de pontos. Já a área da imagem destacada na Figura 26 mostra o nível de precisão do sistema de reconstrução, mesmo para o *Kinect* que provê mapas ruidosos e com buracos.

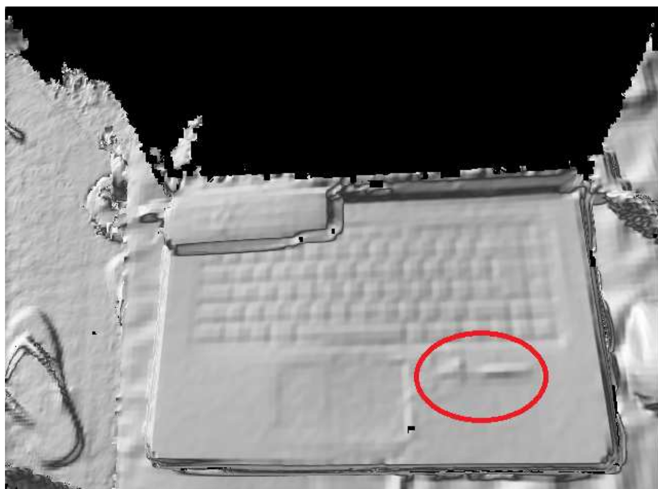

Figura 26 Modelo reconstruído a partir do sistema proposto. A tesoura em destaque na elipse mostra a precisão do sistema.

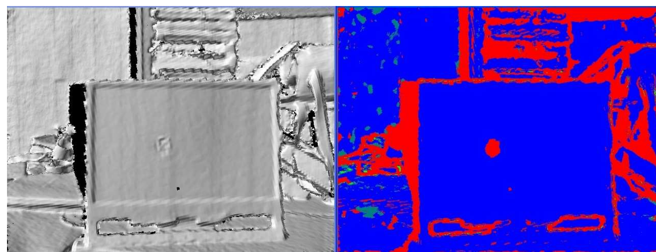

Figura 27 Distribuição de erro num modelo com muitas superfícies planas. (Legenda: Cor Azul = Erro $< 6mm$; Cor Vermelha $= 6mm <$ Erro $< 30mm$; Cor Verde = Erro $> 30mm$.)

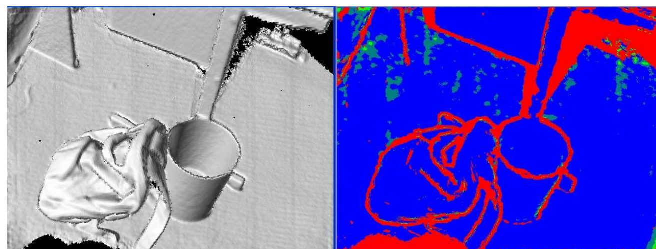

Figura 28 Distribuição de erro num modelo com altas curvaturas.(Legenda: Cor Azul = Erro $< 6mm$; Cor Vermelha $= 6mm <$ Erro $< 30mm$; Cor Verde = Erro $> 30mm$.)

6.6 Quinto Teste: Distribuição de Erro no Modelo Reconstruído

Este teste teve como objetivo verificar a distribuição de erro nos modelos reconstruídos. A terceira versão do sistema foi utilizada para esse teste.

Similarmente ao que foi feito na subseção 6.4, a distribuição de erro foi analisada a partir de dois tipos de modelos: um modelo com muitas superfícies planas e outro modelo circular com altas curvaturas, sendo que a cor azul indica erro menor do que $6mm$, a cor verde indica um erro entre $6mm$ e $30mm$, enquanto a cor vermelha indica um erro maior do que $30mm$.

Os dois modelos comportaram-se de forma similar em relação à distribuição de erro. A maior concentração de erro ocorreu nas regiões de incerteza e nas regiões onde estão localizados os buracos, o que nesse caso foram as bordas do modelo. Isso ocorreu pois essas regiões apresentam uma grande variação nos valores de profundidade entre *frames*.

7 Conclusão

O presente trabalho teve como objetivo descrever e avaliar um sistema de reconstrução tridimensional utilizando *Kinect* e GPU. Para tanto, o sistema proposto foi dividido em três versões: somente CPU, CPU com GPU (*OpenCL*), e CPU com GPU (CUDA-PCL). As duas primeiras versões foram utilizadas para testes de tempo de processamento e análise das fases do algoritmo de reconstrução. Nesses dois sistemas foram utilizados mapas de profundidade criados sinteticamente, simulando mapas de profundidade adquiridos do *Kinect* após a aplicação do filtro bilateral. Já para o terceiro sistema, que executou na média de 90 ms, foram testados aspectos de qualidade da reconstrução do modelo, sendo ela parcial ou completa. Vale lembrar que o tempo de processamento obtido, para o sistema em que foi utilizado, foi adequado, uma vez que permitiu interações em tempo real com o usuário.

Para cada fase da reconstrução tridimensional foram feitas as seguintes avaliações:

- Aquisição dos mapas de profundidade: Verificou-se que a aplicação do filtro bilateral sobre o mapa de profundidade tornou o sistema mais robusto, uma vez que reduziu a quantidade de falhas do ICP.
- Alinhamento das nuvens de pontos: Foi verificado que o variante veloz do ICP é muito robusto para modelos de "fácil"geometria, desde que a rotação incremental seja menor do que 10 graus. Enquanto que para os modelos de "difícil"geometria, a rotação incremental não pode ser maior do que 5 graus.
- Integração volumétrica e *raycasting*: Mostrou-se que o sistema proposto é bastante preciso, reconstruindo modelos de alta qualidade, e que a grande concentração de erro ocorre nas bordas dos modelos reconstruídos.

Além disso, assim como visto na seção 6.2, o processamento da GPU conseguiu reduzir mais de 90% do tempo de

processamento dos algoritmos paralelizáveis em comparação com a CPU.

7.1 Trabalhos Futuros

O sistema de reconstrução tridimensional apresentado neste trabalho é restrito à situação em que o modelo não tem seu formato alterado durante a sua reconstrução.

Como trabalho futuro, pretende-se integrar ao sistema de reconstrução em tempo real um algoritmo de registro deformável, ampliando a possibilidade de interações entre o usuário e o sistema, uma vez que este permitirá que haja reconstrução de modelos que, naturalmente, tem seu formato constantemente modificado (face, pessoas em movimento, etc..).

Além disso, pretende-se futuramente avaliar quão preciso é o modelo reconstruído por este sistema, de forma que se verifique se este pode ser aplicado na área de realidade aumentada aplicada à medicina, em que o nível de precisão do modelo reconstruído deve ser muito alto.

Referências

1. Azuma, R., Baillot, Y., Behringer, R., Feiner, S., Julier, S., MacIntyre, B.: Recent advances in augmented reality. IEEE Comput. Graph. Appl. **21**(6), 34–47 (2001)
2. Besl, P., McKay, N.: A method for registration of 3-d shapes. IEEE Transactions on Pattern Analysis and Machine Intelligence **14**, 239–256 (1992)
3. Besl, P.J.: Active, optical range imaging sensors. Mach. Vision Appl. **1**, 127–152 (1988)
4. Bradski, G., Kaehler, A.: Learning OpenCV: Computer Vision with the OpenCV Library. O'Reilly, Cambridge, MA (2008)
5. Chen, Y., Medioni, G.: Object modelling by registration of multiple range images. Image Vision Comput. **10**, 145–155 (1992)
6. Curless, B., Levoy, M.: A volumetric method for building complex models from range images. In: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, SIGGRAPH '96, pp. 303–312. ACM, New York, NY, USA (1996)
7. Gelfand, N., Ikemoto, L., Rusinkiewicz, S., Levoy, M.: Geometrically stable sampling for the ICP algorithm. In: Fourth International Conference on 3D Digital Imaging and Modeling (3DIM) (2003)
8. Habbecke, M., Kobbelt, L.: A Surface-Growing Approach to Multi-View Stereo Reconstruction. Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on pp. 1–8 (2007)
9. Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., et al.: Kinectfusion : Real-time 3d reconstruction and interaction using a moving depth camera. interactions p. 559–568 (2011)
10. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
11. Kolb, A., Barth, E., Koch, R., Larsen, R.: Time-of-flight cameras in computer graphics. Computer Graphics Forum **29**(1), 141–159 (2010)
12. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3d surface construction algorithm. In: Proceedings of the 14th annual conference on Computer graphics and interactive techniques, SIGGRAPH '87, pp. 163–169. ACM, New York, NY, USA (1987)
13. Newcombe, R.A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A.J., Kohli, P., Shotton, J., Hodges, S., Fitzgibbon, A.W.: Kinectfusion: Real-time dense surface mapping and tracking. In: ISMAR, pp. 127–136. IEEE (2011)
14. Osher, S.J., Fedkiw, R.P.: Level Set Methods and Dynamic Implicit Surfaces, 1 edn. Springer (2002)
15. Posdamer, J.L., Altschuler, M.D.: Surface measurement by space-encoded projected beam systems. Computer Graphics and Image Processing **18**(1), 1–17 (1982)
16. Rusinkiewicz, S., Levoy, M.: Efficient variants of the ICP algorithm. In: Third International Conference on 3D Digital Imaging and Modeling (3DIM) (2001)
17. Rusinkiewicz, S.M.: Real-time acquisition and rendering of large three-dimensional models. PhD Thesis, Stanford University (2001). 1–13
18. Rusu, R.B., Cousins, S.: 3d is here: Point cloud library (pcl). In: International Conference on Robotics and Automation. Shanghai, China (2011)
19. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edn. Addison-Wesley Professional (2010)
20. *Eigen*: (2012). URL http://eigen.tuxfamily.org/. Acessado em 12 de agosto de 2012.
21. Tomasi, C., Manduchi, R.: Bilateral filtering for gray and color images. In: Proceedings of the Sixth International Conference on Computer Vision, ICCV '98, pp. 839–. IEEE Computer Society, Washington, DC, USA (1998)
22. Tsuchiyama, R., Nakamura, T., Iizuka, T., Asahara, A., Miki, S.: The OpenCL Programming Book. Fixstars Corporation (2010)
23. Turk, G., Levoy, M.: Zippered polygon meshes from range images. In: Proceedings of the 21st annual conference on Computer graphics and interactive techniques, SIGGRAPH '94, pp. 311–318. ACM, New York, NY, USA (1994)
24. Vasudevan, R., Zhou, Z., Kurillo, G., Lobaton, E., Bajcsy, R., Nahrstedt, K.: Real-time stereo-vision system for 3d teleimmersive collaboration. In: IEEE International Conference on Multimedia and Expo, pp. 1208–1213 (2010)
25. Vieira, T., Velho, L., Lewiner, T., Peixoto, A.: Registro automatico de superficies usando spin-images. In: Proceedings of SIBGRAPI. VI Workshop de Teses de Dissertações (2007)
26. Weise, T.: Real-time 3d scanning. Ph.D. thesis, ETH Zurich (2010)
27. Weise, T., Bouaziz, S., Li, H., Pauly, M.: Realtime performance-based facial animation. ACM Trans. Graph. **30**(4), 77:1–77:10 (2011)
28. Weise, T., Wismer, T., Leibe, B., Gool, L.V.: Online loop closure for real-time interactive 3d scanning. Comput. Vis. Image Underst. **115**(5), 635–648 (2011)
29. Witkin, A.P.: Scale-space filtering. In: Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 2, IJCAI'83, pp. 1019–1022. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1983)
30. Woo, M., Neider, J., Davis, T., Shreiner, D.: OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
31. Young, T.: Handbook of Pattern Recognition and Image Processing: Computer Vision. Academic Press (1994)