Self Localization and Mapping using the Kinect

Processamento de imagem e visão 2014/2015

Professores:
Jorge Marques
João Paulo Costeira

Ricardo Ferreira 69407
Afonso Clara 70118
Nuno Mendes 73716
Lisboa, 30 de Dezembro de 2014

Introdução

O objetivo global do projeto trata-se de conseguir obter um modelo 3D de um determinado local, para que um sistema autónomo móvel se consiga localizar nele, com a ajuda dos seus sensores.

Para resolver este problema é utilizado um dispositivo capaz de obter uma point cloud de um local estático, um Kinect. Logo, este dispositivo é incapaz de reconstruir toda a cena, sendo que apenas informação local está disponível. É então necessária a recolha e fusão de informação sobre múltiplos locais (*point cloud* 3D local) de forma a construir o modelo 3D da cena. Este problema pode ser resolvido, sabendo a posição e orientação do Kinect em cada observação, em todos os pontos da sua trajetória, de modo a fundir as várias *point clouds* locais.

Este problema é abordado de duas formas diferentes, a primeira com uma câmara auxiliar estática para determinar a posição e orientação do Kinect (parte 1) e a segunda realizada através do algoritmo SIFT (parte 2), também com o mesmo objetivo.

De notar que o Kinect não devolve diretamente os pontos (x,y,z) no referencial da câmara de profundidade, é necessário passar os pontos projetivos da câmara IR para o referencial 3D da mesma, sendo que para isso são utilizados os parâmetros intrínsecos da câmara IR.

Para determinar as cores correspondentes dos pontos (x,y,z) determinados, é necessário executar uma transformação do referencial da câmara de profundidade para o referencial da câmara RGB e depois projetá-los nesta última câmara. Assim, finalmente, é possível encontrar as cores respetivas (vermelho, verde e azul) dos pontos iniciais.

Parte 1

De forma a resolver o problema da posição e orientação do Kinect, foi utilizada uma biblioteca existente, *ArUco*, que consegue determinar as posições relativas de códigos "QR", numa caixa com 6 faces, sendo que nas faces laterais se encontram estes códigos, cada uma com 3 deles, e uma câmara estática que consideramos o referencial global da cena.

O problema global trata então de encontrar as translações e rotações do referencial da câmara de profundidade para o referencial global da cena, para cada observação, sendo que para isso é necessário calcular as rotações e translações da câmara estática para o Aruco. Tendo este passo sido completo é então necessário ter em conta a rotação e translação do Aruco para a câmara

de profundidade do Kinect. Finalmente, com este passo concluído, são então calculadas a rotação e translação do referencial da câmara de profundidade do Kinect para o referencial global.

Vão existir também dois casos importantes, quando só aparecem na câmara estática (referencial global) códigos "QR" pertencentes a apenas uma face (homografia) e quando aparecem na câmara estática mais do que uma face na fotografia. Estes dois casos serão tratados de duas formas diferentes no decorrer desta parte do projecto.

Para construir a point cloud da cena é então necessário calcular a transformação dos pontos obtidos (x,y,z), com a rotação e translação calculados.

Parte 2

Uma abordagem alternativa utilizada nesta segunda fase do projeto foi a determinação da posição e orientação do Kinect, mas agora sem a câmara auxiliar estática. É para isso utilizado o algoritmo SIFT de forma a realizar o *matching* entre as várias observações do Kinect, permitindo detetar *features* em duas imagens consecutivas com características semelhantes e que de alguma forma estejam relacionadas.

De seguida, é determinado a rotação e translação entre os pontos das duas imagens, conseguindo fazer a fusão entre as *point clouds* das duas imagens.

Finalmente, é necessário definir um referencial fixo para o qual se vai transformar as *point clouds* de cada observação, tendo em conta todas as rotações e translações calculadas.

Abordagem

Composição Aruco

Trata-se de uma caixa de cartão com 12 códigos QR e um Kinect integrado no topo da caixa. É a partir deste objeto que é possível determinar as rotações e translações da câmara estática para a câmara de profundidade do Kinect. As posições relativas dos códigos QR são determinadas através da biblioteca ArUco, tendo em conta o tamanho especificado dos códigos QR.

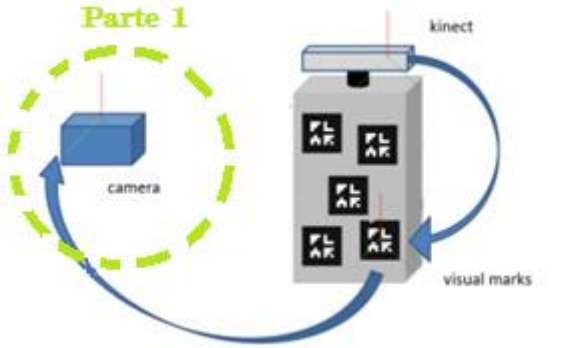

Figura 1 - Caixa Aruco com esquema operacional da parte 1 do projeto.
Fonte: http://users.isr.ist.utl.pt/~jsm/teaching/piv/Project_1415.pdf

Composição do Kinect

O Kinect é um aparelho de entretenimento que engloba um emissor IR 2D com uma resolução 640x480 em conjunto com um sensor de profundidade IR com resolução de 1600x1200 e uma câmara RGB normal com resolução 640x480.

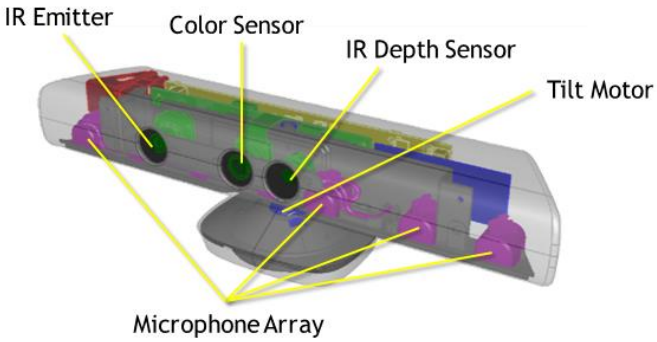

Figura 2 - Esquema estrutural do dispositivo Kinect. Fonte: http://msdn.microsoft.com/en-us/library/jj131033.aspx

Obtenção de dados do Kinect

Os dados obtidos do Kinect em cada instante são dois tipos de ficheiros:

- Uma fotografia em formato *.png*, respetivo à câmara RGB. Esta imagem é lida no Matlab, compondo uma matriz 480x640x3, em que as duas primeiras dimensões correspondem à resolução da fotografia e a terceira corresponde ao conjunto R,G e B das cores dos pixéis, numerados de 0 até 255 bit.
- Uma matriz no formato do Matlab com dimensões 480x640. Cada entrada desta matriz corresponde à profundidade detetada no referencial do sensor IR do Kinect.

Tratamento de dados do Kinect

Para utilizar a point cloud local registada pelo Kinect, é necessário obter as coordenadas (x,y,z) no referencial da câmara IR (denominado a partir de agora de referencial 1, por simplicidade). Para isso, são utilizados os parâmetros intrínsecos da câmara IR para passar os pontos (u,v) para coordenadas 3D, obtendo-se desta forma as coordenadas (x,y,z) no referencial 1. De notar que neste passo foi efetuada uma filtragem do fotograma recebido pela câmara IR, visto que existiam pixéis com profundidade 0, o que não faria sentido. Foram assim eliminados os pontos que apresentavam este defeito.

Para obter as cores respetivas de cada ponto obtido no referencial 1 foi necessário passar estes pontos para o referencial da câmara RGB (denominado a partir de agora como referencial 2), com recurso às matrizes de translação e rotação desta transformação (referencial 1 para o referencial 2). Depois de concluído este passo é necessário calcular a projeção destes pontos obtidos na câmara RGB de forma a encontrar a correspondência efetiva.

Parte 1

Para resolver esta primeira parte do problema, foram calculadas iterativamente as rotações e translações entre o referencial da câmara de profundidade e o referencial global estático da câmara auxiliar, para cada observação, tendo para isso sido subdividido o problema nas seguintes partes:

- Deteção do número de faces visíveis pela câmara auxiliar estática. Este passo é muito importante, visto o algoritmo se comportar de formas diferentes caso o número de faces visíveis seja igual a 1 ou 2 ou mesmo até nenhuma.
    - Caso o número de faces visíveis pela câmara auxiliar seja igual a duas: é aplicado o método dos mínimos quadrados, com a utilização da decomposição em

valores singulares e finalmente a determinação da matriz R e vetor t. Esta matriz é decomposta em duas submatrizes "Q" e "q". De seguida é calculado alfa de acordo com a seguinte fórmula:

$$\alpha = \frac{\sqrt{3}}{\|Q\|_f}$$

Tendo em conta que é utilizada a norma de frobinius de Q. Pode já ser calculada a translação, $t = alfa * q$, aproximadamente. Para determinar a matriz de rotação tem que ser calculada utilizando o algoritmo de Procrustes, ficando no final $R = U\Sigma V^T$.

- Se todos os códigos "QR" visíveis forem coplanares, temos o caso de uma homografia e por isso tem que se proceder de outra forma, visto que existe uma coluna da matriz P que é multiplicada por zeros, não sendo relevante.
  As coordenadas dos pontos dos códigos "QR" estão no referencial do marcador número 2, sendo que aqui é necessário passar para o referencial do primeiro código "QR" da face visível, de forma a poder solucionar o problema. É calculada a matriz M, com estes pontos transformados.
  Esta matriz M é decomposta em valores singulares de forma a obter a matriz H aproximada.
  De seguida, é multiplicada a inversa da matriz dos parâmetros intrínsecos da câmara auxiliar pela matriz H aproximada, obtendo-se a partir desta a matriz Q.
  É possível agora calcular alfa:

$$\alpha = \frac{\sqrt{2}}{\|Q\|_f}$$

  São calculados agora os vetores r1, r3 e r2 a partir do produto externo dos primeiros. Com estes vetores podemos construir uma nova matriz, a matriz Qh, que possui estes vetores ordenados em coluna.
  Aplicando agora a decomposição em valores singulares à matriz Qh, é possível calcular a matriz rotação para o referencial provisório, sendo necessário passar esta rotação para o código "QR" número 2.
  Finalmente é possível calcular a translação neste referencial com a matriz H aproximada e alfa, sendo também necessário passar esta translação para o referencial do código "QR" número 2.

- De seguida é adquirida a *point cloud* local de uma observação, com as suas cores correspondentes no referencial da câmara IR.
- Posteriormente, é calculada a rotação e translação do referencial da câmara IR para o referencial global, utilizando para isso a rotação e translação calculadas anteriormente e as transformações de calibração fornecidas pelo utilizador (da câmara IR para a caixa do Aruco).
- Finalmente, a *point cloud* observada é transformada para o referencial global com a transformação calculada no passo anterior.

É realizado este ciclo para todas as observações do Kinect. No final, são filtrados os pontos redundantes e é feita uma voxelização da *point cloud* da cena.

Parte 2

Esta segunda parte do problema, já não possui a câmara auxiliar estática, tendo sido utilizado pois, o algoritmo SIFT, como já foi referido anteriormente.

Foram executados os seguintes passos, de uma forma iterativa para cada imagem:

- É adquirida a *point cloud* local de uma observação, com as suas cores correspondentes no referencial da câmara IR.
- São obtidos os *features* e os *descriptors* de cada imagem utilizando o algoritmo SIFT

Tendo agora sido obtidos os *features* e os *descriptors* de cada observação, são executados os seguintes passos, iterativamente:

- É agora feito o *matching* entre duas imagens consecutivas de forma a achar os features que se encontram presentes nestas duas imagens. É obtida uma lista de pontos (u,v), já que correspondem a projeções na câmara RGB.
- É necessário agora passar estes pontos para coordenadas 3D no referencial da câmara de profundidade do Kinect, tendo também um passo de filtragem de features que tenham coordenada nula do eixo cartesiano z.
- Finalmente, é calculada a rotação e translação entre as duas imagens com o algoritmo de Procrustes, sendo estas guardadas num *array* para utilização futura.

Por último, são então calculadas as translações e rotações de cada *point cloud* local para o referencial definido da primeira observação (fixo) e é efetuada a transformação de cada *point cloud* para o referencial fixo. São também filtrados os pontos redundantes e é feita uma voxelização da *point cloud* da cena.

Experimentação

Antes de se apresentarem alguns resultados, é importante referir alguns aspetos relacionados com as funções *pivwitharuco* e *pivimagematching*.

Como *inputs* opcionais, a função *pivwitharuco* tem as opções: *epnp* que utiliza o algoritmo descrito em [1] para estimar a pose nas fotos do Aruco, *backproj*, em que faz o *plot* das retroprojeções e estima o erro das mesmas (de notar que ao output pode ser adicionado um 3º argumento, para retornar um vetor com os erros das retroprojeções), *voxel_alt,* que permite criar a voxelização da *point cloud* com um valor de voxel atribuído, *pcd e cc*, para criar ficheiros que podem ser usados para visualizar a *point cloud* no PCL viewer e no Cloud Compare, respectivamente, *ccimage*, em que ocorre a criação de um ficheiro (Cloud Compare) para cada imagem, e por ultimo a opção *cloud*, em que o *plot* da *point cloud* é realizado no Matlab.

Além dos *inputs* obrigatórios, a função *pivimagematching* tem ainda como *inputs* opcionais os seguintes: *threshold*, que permite ao utilizador definir o limiar usado pela biblioteca *vl*_feat para considerar a existência de uma correspondência entre dois pontos de diferentes imagens (caso não seja indicado nenhum valor, a função assume um limiar igual a 4); voxel_*alt*, que permite escolher o tamanho de voxel usado na voxelização; *pcd*, que permite a criação de um ficheiro compatível com *PCL viewer*; *cc* que permite a criação de um ficheiro compatível com *Cloud Compare*; *cloud* que permite obter o gráfico em *Matlab* da nuvem de pontos.

Parte 1

Relativamente à primeira parte do projeto foram detetados 3 problemas que podem levar a um mau registo de diferentes imagens, e a erros acentuados na formação da point cloud:

O primeiro é a incapacidade do sensor Kinect de detetar a distância de superfícies de cor negra, uma vez que os raios infravermelhos utilizados para o processo de determinação da profundidade são absorvidos por estas superfícies.

Uma forma de contornar este problema será fotografar a mesma imagem de diferentes orientações e com uma boa iluminação, de modo a se poder realizar uma sobreposição das *clouds* resultantes do registo, com uma posterior análise dos pontos redundantes. Esta pode ser realizada com recurso ao processo de voxelização, em que a partir dos pontos obtidos se realiza uma seleção dos que se encontram dentro do mesmo volume unitário, isto é, caso seja definido um *voxel* de dimensão 0.1, todos os pontos da *point cloud* final que se encontrem dentro de um volume de 0.1x0.1x0.1 serão convertido num único, sendo a cor atribuída a mediana das componentes RGB.

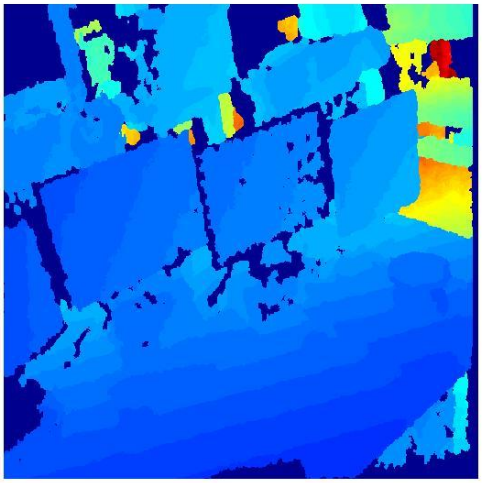

Figura 3 - Gráfico 3D com representação cromática das profundidades detetadas pela câmara IR do Kinect. De notar que as cores mais escuras são os pontos onde não são detetados os valores de profundidade, pois os ecrãs são negros.

O segundo problema é a instabilidade do Kinect aquando a medição de profundidades elevadas, visto que o fabricante garante uma medição correta apenas quando a distância é inferior a cerca de 4 metros (http://msdn.microsoft.com/en-us/library/hh973078.aspx#Depth_Ranges).

O terceiro problema é a estimação da pose do Aruco, uma vez que usando métodos lineares básicos, como a nossa implementação do registo do Aruco, contém erros associados, que quando analisados com alguns *datasets* (por exemplo os do *testing*) pode chegar mesmo a ter cerca de 130 pixéis de erro (nos casos não-planares) quando é feita a medição do erro com recurso à retroprojeção.

A solução encontrada passa por usar métodos lineares mais sofisticados que são robustos e realizam a computação de um registo em tempo *O(n),* como o EPnP (*Efficient Perspective-n-Point Camera Pose Estimation*) [1] ou o RPnP (*Robust O(n) Solution to the Perspective-n-Point Problem*) [2].

Na seguinte tabela é apresentado o erro da retroprojeção dos pontos usados para o registo das 35 imagens no *dataset* de teste.

Número da imagem de teste	Método linear normal	EPnP [1]		Número da imagem de teste	Método linear normal	EPnP [1]
1	16,65952	4,850337		18	8,554224	1,502817
2	17,42005	4,686038		19	7,705269	1,265589
3	18,305	4,428392		20	8,39328	1,597585
4	18,86195	4,1506		21	8,75105	1,712459
5	18,82678	3,871464		22	9,303893	1,884818
6	20,09885	3,578772		23	135,6078	2,856992
7	21,04087	3,365237		24	43,9512	3,873341
8	21,61471	3,234896		25	38,87281	3,805146
9	21,95183	3,071999		26	2,921512	1,361516
10	22,82412	2,967794		27	1,798424	1,236526
11	36,85226	1,668377		28	1,59681	1,335947
12	56,18773	1,943783		29	2,000393	1,348847
13	7,871895	0,981334		30	2,75762	1,316502
14	7,191174	0,745449		31	4,5529	2,885997
15	7,306473	0,868044		32	8,093846	2,968215
16	7,768577	1,03232		33	7,911039	2,415767
17	7,601195	1,014466		34	8,007123	2,526788
18	8,554224	1,502817		35	7,891235	2,639977
Média	18,2015	2,428404				

Tabela 1 – Análise dos erros da retroprojeção com o dataset de 35 imagens utilizando o método linear normal e o método EPnP

Neste caso foi apenas implementado o método descrito em [1] com uma otimização de *Gauss-Newton*, o que melhora substancialmente o tempo de processamento, e minimiza também os erros na estimação da pose. São então mostradas de seguida duas retroprojeções ilustrando a melhoria na estimação da pose do Aruco.

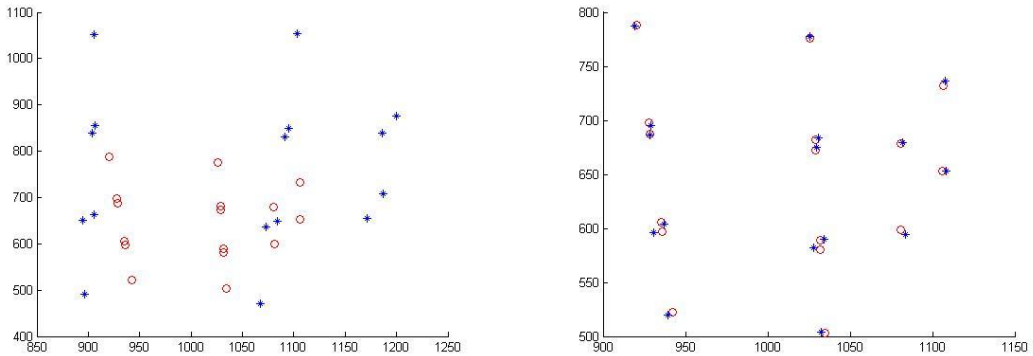

Figura 4 – Representação gráfica da estimação da pose do Aruco utilizando na esquerda o método linear normal e na direita o método EPnP

Estas melhorias são então propagadas às point clouds, onde há um exemplo flagrante de que a estimação da posição de todos os pontos no referencial global é substancialmente melhorado.

Note-se que o banco à direita na imagem sofre menos *overlaping* de sucessivas imagens estimadas quando se usa o EPnP [1] (figura 5, em cima).

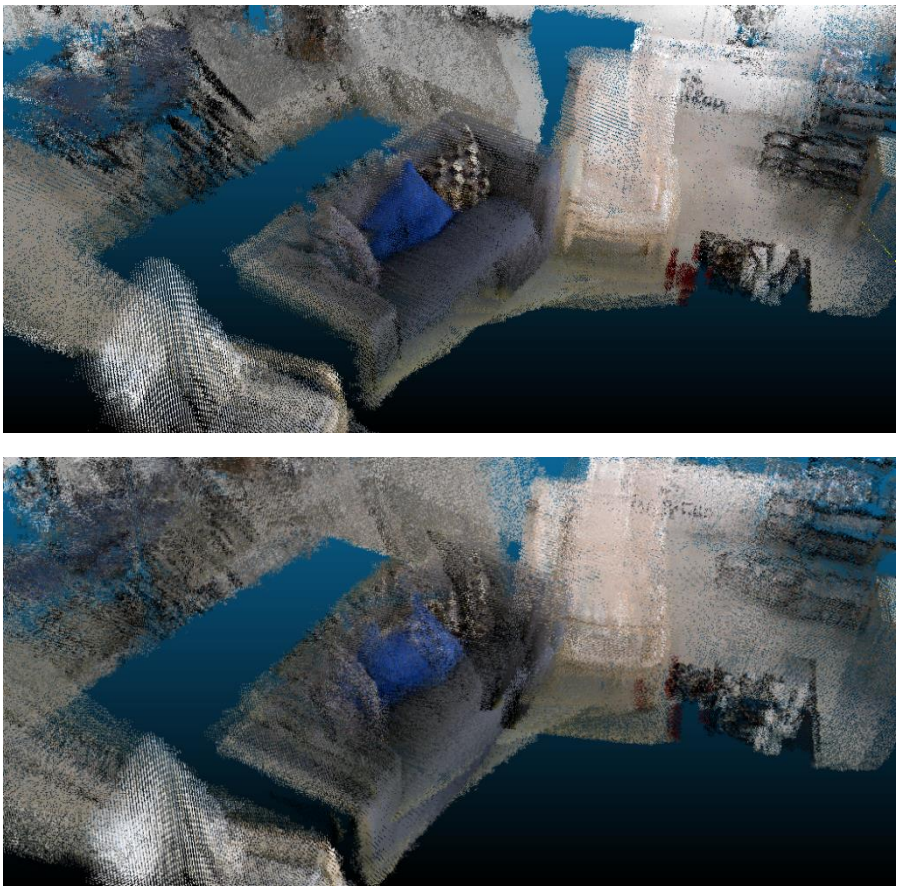

Figura 5 - Representação gráfica das point clouds obtidas com o método EPnP (em cima) e com o método linear normal (em baixo)

Por último é de salientar que para um *dataset* menor (*newpiv1*) todo este processo funciona com muito menos erros, sendo os resultados bastante positivos, pois tem erros de retroprojeção na ordem de 1 pixel, conseguidos com o EPnP [1]. Também o tempo de processamento é bastante rápido, sendo apenas a criação de ficheiros ou a voxelização os processos mais demorados. Já a obtenção de uma *point cloud raw* demora aproximadamente 0.2 segundos a concretizar para cada imagem.

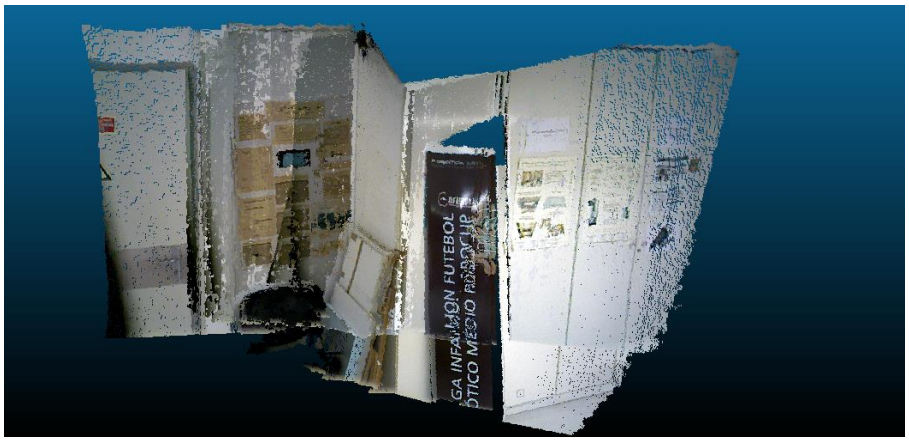

Figura 6 – Representação gráfica da point cloud obtida com o método EPnP para o dataset 'newpiv1'

Parte 2

Os problemas detetados na parte 1 estendem-se para esta parte também, nomeadamente as limitações do sensor *Kinect*.

Os primeiros testes foram realizados com *datasets* contendo poucas imagens. O principal objetivo foi averiguar a qualidade do algoritmo e ajustar, caso fosse necessário, alguns parâmetros, nomeadamente aqueles relacionados com a biblioteca que implementava a *SIFT*.

A figura 7 apresenta os resultados obtidos para um dos *datasets*. Após alguns testes com vários conjuntos de imagens, concluiu-se que a melhor abordagem seria a função ter como *input* opcional o limiar usado pela função da biblioteca *vlfeat* que obtém as correspondências entre *keypoints* de duas imagens.

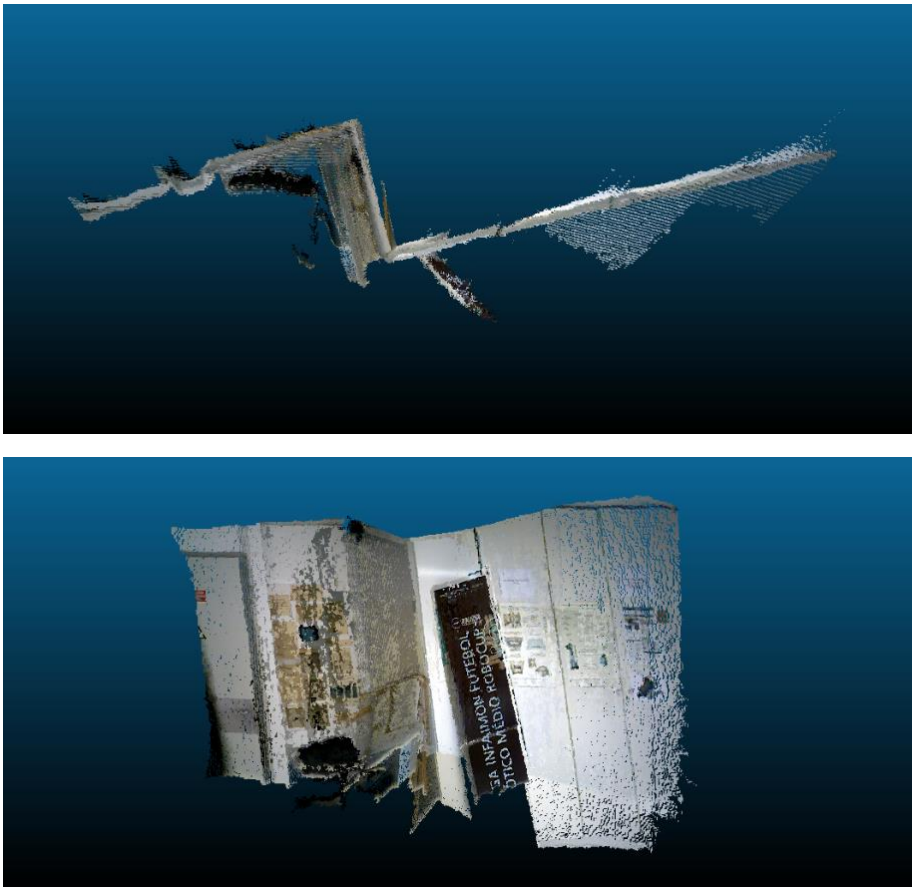

Figura 7 - Resultados obtidos para o conjunto de fotografias *newpiv1*. O limiar usado foi 2.5. Não foi utilizada voxelização. Estão representados 1.5 milhões de pontos, aproximadamente.

A figura 8 apresenta os resultados obtidos para o conjunto de imagens *testing*. Este conjunto apresenta um tamanho considerável e portanto foi possível observar a eficácia da função ao nível do tempo de computação. Neste campo, salientar que de facto a função é razoavelmente rápida. A maioria do tempo de computação é gasto a calcular a *SIFT* de cada uma das imagens. Além disso, se o utilizador optar por realizar uma voxelização seguida de gravação da nuvem de pontos o tempo de computação é também afetado na proporção inversa do tamanho do voxel escolhido.

Por outro lado, este conjunto de imagens revelou ainda uma fragilidade da função. A função não é capaz de lidar com "saltos" bruscos entre imagens, ou seja, apenas é capaz de construir uma

nuvem de pontos através de um conjunto de imagens que tenham sido obtidas de forma contínua. Uma das possíveis soluções seria usar geometria epipolar usando a nuvem de pontos atualizada e a nova imagem que se pretende inserir.

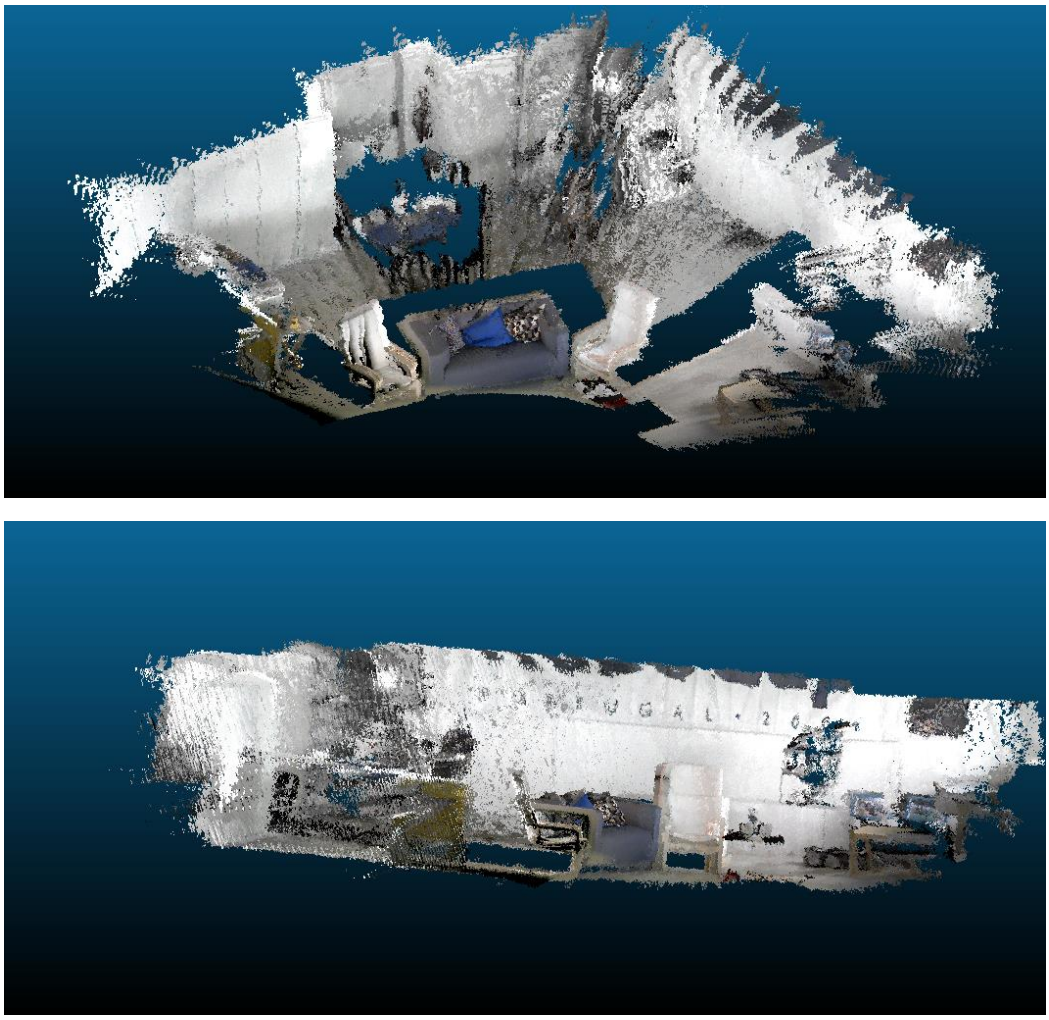

Figura 8 - Resultados obtidos para o conjunto de imagens *testing.* O limiar usado foi 4. O tamanho de voxel usado foi 0.01, obtendo-se aproximadamente 3 milhões de pontos.

Referências

[1] Lepetit, V., Moreno-Noguer, F., & Fua, P. (2009). Epnp: An accurate o (n) solution to the pnp problem. International journal of computer vision, 81(2), 155-166.

[2] Li, S., Xu, C., & Xie, M. (2012). A robust O (n) solution to the perspective-n-point problem. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 34(7), 1444-1450.