Universidade de Aveiro
2021

Gonçalo Marques Gomes

Braço robótico como jogador de jogos de tabuleiro

Robot arm as a player of board games

Universidade de Aveiro
2021

Gonçalo Marques Gomes

Braço robótico como jogador de jogos de tabuleiro

Robot arm as a player of board games

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Nuno Lau, Professor associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Artur Pereira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Professor Doutor António José Ribeiro Neves
Professor Auxiliar, Universidade de Aveiro

vogais / examiners committee

Professor Doutor António Paulo Gomes Mendes Moreira
Professor Associado Com Agregação, Universidade do Porto - Faculdade de Engenharia

Professor Doutor José Nuno Panelas Nunes Lau
Professor Associado, Universidade de Aveiro

agradecimentos / acknowledgements

Quero agradecer aos meus pais, Jorge e Alzira pela ajuda, apoio e confiança sempre transmitidos, às minhas irmãs, Andreia e Catarina, pela ajuda e apoio incondicional que sempre mostraram.
Aos meus orientadores, professor Nuno Lau e professor Artur Pereira, pela disponibilidade e conhecimento comigo partilhado no esclarecimento de dúvidas e orientação.

Palavras Chave

braço robótico, jogos de tabuleiro, jogo das damas, jogo dos peões, tabuleiro de xadrez, visão por computador.

Resumo

Com o aumento da presença de robôs no dia a dia das pessoas, seja no trabalho, onde estes executam atualmente tarefas repetitivas ou perigosas, ou em casa, através de robôs domésticos, aumenta também o nível de interação entre robôs e humanos. Esta interação é, para já, ainda bastante reduzida, uma vez que na grande maioria dos casos, um robô é utilizado para fazer algo sem intervenção humana ou sem necessitar da ajuda deste. No entanto, é expectável que num futuro próximo os humanos venham a lidar com robôs num patamar próximo ao de outros humanos, por exemplo na indústria dos jogos de azar, onde robôs humanoides poderão substituir os *dealers* dos casinos ou até os próprios jogadores. Para que isto aconteça é necessária uma destreza enorme por parte do robô para que consiga manipular cartas, mas também que o humano se sinta confortável e em segurança a jogar com um robô. Esta tese procura, assim, apresentar um sistema capaz de interagir com humanos através de jogos de tabuleiro, onde o braço robótico e o humano estejam ao mesmo nível como sendo dois jogadores iguais. Sendo que objetivo principal é permitir que o braço robótico jogue mais do que um jogo de tabuleiro com a mesma arquitetura. Fazem parte desta arquitetura a deteção do tabuleiro, feita por meio de visão por computador auxiliada na biblioteca OpenCV, a segmentação das peças, a partir da filtragem da *point cloud* obtida pela câmara, a identificação e localização das peças no tabuleiro de jogo, a descoberta das jogadas feitas pelo humano, através da monitorização do estado do tabuleiro, ou seja, da posição das peças neste e, finalmente, a integração dos motores de jogo como criadores da jogada do robô e a manipulação das peças por parte deste. Os resultados mostram a exequibilidade de uma arquitetura deste género através de testes, bem-sucedidos, em ambiente real da maior parte dos componentes que a compõem.

Keywords robotic arm, board games, checkers, pawn game, chess board, computer vision.

Abstract

With the increasing presence of robots in people's daily lives, whether at work, where they currently perform repetitive or dangerous tasks, or at home, through domestic robots, the level of interaction between robots and humans also increases. This interaction is, for now, still quite reduced, since in the vast majority of cases, a robot is used to do something without human intervention or without the need for human assistance. However, it is expected that in the near future humans will deal with robots at a level close to other humans, for example in the gambling industry, where humanoid robots can replace casino dealers or even the players themselves. For this to happen it is necessary a huge dexterity on the part of the robot to be able to manipulate cards, but also that the human feels comfortable and safe playing with a robot. Thus, this thesis seeks to present a system capable of interacting with humans through board games, where the robotic arm and the human are at the same level as two equal players. The main objective being to allow the robotic arm to play more than one board game with the same architecture. This architecture is composed of the detection of the board, made by means of computer vision aided in the OpenCV library, the segmentation of the pieces, from the filtering of the point cloud obtained by the camera, the identification and location of the pieces on the game board, the discovery of the moves made by the human, through monitoring of the state of the board, that is, the position of the pieces on it and, finally, the integration of the game engines as creators of the robot's move and the manipulation of the pieces by the latter. The results show the feasibility of such an architecture through successful tests, in a real environment, of most of its components.

Conteúdo

Conteúdo i

Lista de Figuras iii

Lista de Tabelas vii

Glossário ix

1 Introdução 1

1.1 Motivação 1
1.2 Estrutura da tese 2

2 Estado da arte 3

2.1 Introdução 3
2.2 Perceção 3
2.3 Deteção de jogadas 6
2.4 Decisão 8
2.5 Manipulação 8

3 Arquitetura da solução 11

3.1 Introdução 11
3.2 Restrições 12
3.3 Similaridades entre diferentes jogos de tabuleiro 13
3.4 Estrutura 13
3.4.1 Deteção do tabuleiro 15
3.4.2 Deteção das peças 15
3.4.3 Estado do tabuleiro 15
3.4.4 Deteção da jogada 15
3.4.5 Decisor 16
3.4.6 Motor de jogo 16
3.4.7 Manipulação 16

4 Perceção **19**

4.1 Introdução 19

4.2 Filtragem da *point cloud* 20

4.2.1 Deteção do plano da mesa 21

4.2.2 Transformação entre o *frame* da câmara e a mesa 23

4.2.3 Remoção dos pontos restantes não pertencentes às peças 23

4.3 Deteção do tabuleiro de jogo 24

4.3.1 Transformação entre a *frame* do tabuleiro e a mesa 25

4.3.2 Construção dos limites dos quadrados do tabuleiro 37

4.4 Estado do tabuleiro 38

5 Decisão **47**

5.1 Introdução 47

5.2 Regras dos jogos 47

5.2.1 Damas Inglesas 47

5.2.2 Jogo dos peões 48

5.3 Motor do jogo das damas 49

5.4 Integração do motor de jogo das damas neste sistema 50

5.5 Motor do jogo dos peões 57

5.5.1 O algoritmo *minimax* 57

5.5.2 Implementação do motor de jogo dos peões 60

6 Resultados **65**

6.1 Introdução 65

6.2 Experiências e Resultados 65

6.2.1 Deteção do tabuleiro 65

6.2.2 Segmentação das peças e estado do tabuleiro 69

6.2.3 Deteção das jogadas 71

6.2.4 Situação de jogo 74

6.2.5 Execução de jogadas complexas 76

7 Conclusão **79**

Referências **81**

Lista de Figuras

2.1 Hierarquia dos classificadores para a deteção das peças no tabuleiro 5
2.2 Deteção do tabuleiro através da deteção das arestas 6
2.3 Deteção da jogada a partir do estado do jogo antes e depois 7
2.4 Subtração entre a imagem anterior ao movimento do humano e após 8

3.1 Peças criadas para uso nos vários jogos 13
3.2 Arquitetura geral do sistema 14
3.3 Posições fora do tabuleiro 18

4.1 Estrutura para determinação do estado do tabuleiro. 20
4.2 Captações do tipo *point cloud* 21
4.3 Distância máxima entre a câmara e o tabuleiro 22
4.4 *Point clouds* vistas de topo antes e após filtragem 24
4.5 Notação algébrica de xadrez 25
4.6 Imagem inicial recebida da câmara 25
4.7 Retorno da função *goodFeaturesToTrack()* sobreposto na imagem inicial 27
4.8 As linhas vermelhas representam a distância de um determinado canto ao canto médio. . 27
4.9 Resultado após remoção dos cantos fora do tabuleiro, só ficaram os pontos a vermelho. . 28
4.10 Resultado de ligar todos os pontos por uma linha entre si. 29
4.11 Linha azul corresponde ao contorno retornado pela função *findContours*. 29
4.12 Resultado da função *convexHull()* quando aplicada ao contorno da figura 4.11. 30
4.13 Resultado da função *skimage.measure.ransac* quando aplicada aos pontos que originam o contorno a branco da figura 4.12. 31
4.14 Remoção das linhas retornadas pela função RANSAC 31
4.15 Distorção causada pelo ângulo da câmara com a mesa 32
4.16 Ângulos azuis estão mais perto da câmara pelo que parecem ser inferiores a 90º e a vermelho mais distantes pelo que parecem ser superiores a 90º. 32
4.17 Exemplo de uma transformação de perspetiva 33
4.18 Má remoção das linhas retornadas pela função RANSAC 33
4.19 Resultado da procura do centroide e do ângulo do tabuleiro 34

4.20 Determinação do ângulo do tabuleiro 35
4.21 Definição do lado do tabuleiro que é do humano com base na posição do robô 36
4.22 Limites do tabuleiro 37
4.23 Resultado da identificação das peças com base na sua localização no tabuleiro em simulação. Os eixos do tabuleiro também estão presentes. 40
4.24 Resultado da identificação das peças com base na sua localização no tabuleiro em real. Os eixos do tabuleiro também estão presentes. 40
4.25 Comparação entre uma má deteção e uma correta deteção dos topos das peças 41
4.26 Resultado da projeção dos pontos da *point cloud* pertencentes ao topo de uma peça numa imagem 2D 42
4.27 Resultado da aplicação da tranformação morfológica dilatação às imagens da figura 4.26 . 43
4.28 A vermelho o resultado da função *findContours()* quando aplicada à figura 4.27 e a preto o resultado da função *minAreaRect()* quando aplicada aos contornos a vermelho 44

5.1 Posição inicial do jogo dos peões. 48
5.2 Um estado de jogo qualquer S_i pode originar n estados S_{i+1}. O estado S_{i+1} será apenas um dos estados S_{j_n}. 51
5.3 Representação de uma jogada composta por dois saltos e consequentemente duas capturas. 52
5.4 Grafo representativo dos quadrados onde as peças podem existir sobreposto no tabuleiro 53
5.5 Caminhos retornados pela função *graph-tool.all_paths()* 55
5.6 Nesta jogada a peça vermelha é uma dama pelo que pode capturar em qualquer direção, assim, após o primeiro salto pode capturar a segunda peça ao fazer um salto para trás. . 56
5.7 Diferença entre caminhos que cumprem os requisitos mas apenas um é legal 56
5.8 Árvore representativa do método de pesquisa *minimax* 59
5.9 Matriz do estado de jogo do motor de jogo dos peões. 60
5.10 Representação da jogada *en passant*, os valores das células vazias foram removidos para melhorar a visualização. 62
5.11 Interfaces gráficas para interação direta com o motor de jogo dos peões. 63

6.1 Estas figuras apresentam a configuração utilizada para a realização das experiências . . . 66
6.2 Tabuleiros usados para teste da deteção do tabuleiro 67
6.3 Estas figuras apresentam a configuração utilizada na deteção dos tabuleiros em ambiente real. 68
6.4 Resultado da deteção do tabuleiro de madeira 69
6.5 Resultado da deteção do tabuleiro impresso 69
6.6 Resultado da deteção do tampo da mesa e segmentação das peças 70
6.7 Visualização do estado do tabuleiro. 71
6.8 Figuras da *point cloud* recolhida pela câmara nos vários momentos de uma jogada, com as etiquetas identificativas do estado do tabuleiro sobrepostas nas peças 72

6.9 Human Robot Interface (HRI) antes da jogada 73
6.10 HRI depois da jogada 74
6.11 Posicionamento dos vários componentes usados para simular um jogo completo. 75
6.12 Vista de cima do tabuleiro onde se pode comprovar a disposição das peças com vários ângulos. 75
6.13 No lado direito do tabuleiro pode-se ver o bloco cinzento usado para simular a presença de um braço humano na altura de se fazer a jogada deste. 76
6.14 Imagem adaptada do vídeo do jogo dos peões. 76

Lista de Tabelas

2.1 Valores usados para avaliação do tabuleiro após uma jogada 8

4.1 Intervalos das componentes HSV na biblioteca OpenCV para a identificação da cor azul e da cor vermelha. 39

5.1 Valores usados para determinar a avaliação de um estado do jogo. 62

6.1 Posição do tabuleiro, em relação à base do braço robótico, em todas as experiências. . . . 66

6.2 Posição do tabuleiro, em relação à base do braço robótico, em todas as experiências. . . . 66

Glossário

ROS	Robot Operating System
PCL	Point Cloud Library
RANSAC	Random Sample Consensus
OpenCV	Open Source Computer Vision Library
ROI	Region of Interest
HSV	Hue Saturation Value
RGB	Red Green Blue
HOG	Histogram of Oriented Gradients
SVM	Suport Vector Machine
GUI	Graphical User Interface
HRI	Human Robot Interface

CAPÍTULO 1

Introdução

Existem, atualmente, várias implementações com a intenção de dotar braços robóticos com a capacidade para interagir com humanos em tarefas comuns. Uma destas tarefas que mostra o nível de interação robô-humano é jogar jogos de tabuleiro. No entanto, são raras as que existem com a intenção de adaptar a arquitetura criada para jogar um jogo de tabuleiro numa que consiga jogar vários.

Esta tese tem assim como objetivo construir uma arquitetura que permita que um braço consiga substituir um jogador de jogos de tabuleiro na integra e que este se adapte facilmente a diferentes jogos de tabuleiro.

As principais componentes que constroem esta arquitetura são: a perceção, a manipulação e a decisão. No capítulo 2 são apresentados alguns dos avanços já realizados neste âmbito. No capítulo 3 é descrita a estrutura geral da arquitetura aqui implementada. O capítulo 4 descreve todo o desenvolvimento na área de deteção do tabuleiro das peças e criação do estado do tabuleiro, o capítulo 5 mostra o modo como os motores de jogo foram implementados na arquitetura geral, assim como é que foi feito um motor de jogo de raiz baseado no algoritmo de decisão *minimax*.

A programação foi feita usando o *Robot Operating System (ROS)*[1] e segue uma estrutura de vários nós que trabalham sobre tarefas muito específicas que por sua vez comunicam os resultados destas tarefas a outros nós, que os necessitem, através de mensagens.

1.1 Motivação

Ultimamente, tem-se notado uma maior aproximação entre o ser humano comum e o mundo da robótica. Esta aproximação é evidente na quantidade de eletrodomésticos que partilham a palavra robô no nome, por exemplo robô de cozinha ou robô de limpeza, mas principalmente o convívio cada vez maior entre pessoas e braços robóticos no local de trabalho, por exemplo em fábricas automóveis, onde tarefas repetitivas e perigosas são já feitas por braços robóticos há algum tempo, na indústria espacial, entre outros.

[1] `http://wiki.ros.org/`

Esta tese surge assim como mais um passo nesta aproximação, para informar as pessoas das capacidades de um braço robótico, sendo que nesta tese é explorado o grau de interação possível de se obter entre robô e humano a partir de jogos de tabuleiro, especialmente a flexibilidade da plataforma robótica de conseguir jogar mais do que um tipo de jogo. Estes avanços tecnológicos poderão ser depois demonstrados em feiras, universidades ou exposições ao cidadão comum para que aumente o seu interesse neste tipo de tecnologias.

1.2 Estrutura da tese

Esta tese constrói-se em 7 capítulos sendo o atual a introdução onde é apresentado o âmbito e objetivo da mesma.

No capítulo 2 são estudadas, resumidamente, algumas abordagens e desenvolvimentos existentes na criação de sistemas robóticos para jogar jogos de tabuleiro.

O capítulo 3 define, a partir de uma visão de alto nível, a arquitetura geral do sistema implementado.

O capítulo 4 descreve as várias soluções criadas com base na percepção, ou seja, usando a câmara para detetar os vários componentes do ambiente de um jogo normal, o tabuleiro, a mesa, as peças, entre outros.

O capítulo 5 foca-se na parte da decisão descrevendo como é que foi dada autonomia ao robô para que possa escolher as suas jogadas em resposta a jogadas do humano.

Os resultados obtidos e as experiências que mostraram as funcionalidades do sistema encontram-se no capítulo 6.

Finalmente, no capítulo 7 discute-se a conclusão deste estudo e apresentam-se possíveis alterações ou acréscimos futuros a este sistema.

CAPÍTULO 2

Estado da arte

2.1 Introdução

Uma das razões principais que mais contribui para o constante desenvolvimento de braços robóticos ou robôs no geral é a ideia de atingir paridade plena com a capacidade humana de realizar diversas tarefas. Portanto, não é surpresa nenhuma que o problema de programar braços robóticos para que joguem jogos de tabuleiro contra humanos ou contra outros robôs não seja novo.

Isto porque é uma tarefa que mostra não só a destreza que um robô já possui, mas também a inteligência que este pode ter na sua capacidade de se adaptar a novos movimentos ou a fazer novas tarefas. Esta é uma razão importante pois um ser humano pouco familiarizado com o mundo dos robôs projeta nos jogos de tabuleiro, nomeadamente o xadrez, uma certa capacidade intelectual que não é inerente ao ser humano. Estas emoções podem facilitar a captação da sua atenção ou o seu interesse futuro em ajudar no desenvolvimento desta área.

Com esta ideia em mente será de seguida apresentado, de forma resumida, algum do desenvolvimento que já foi feito nesta área no âmbito da manipulação de peças, deteção do tabuleiro e das peças, na decisão das jogadas, etc...

2.2 Perceção

A perceção é um dos componentes mais importantes em sistemas cujo objetivo é criar algum tipo de interação entre um braço robótico e objetos reais. No caso concreto desta tese é importante para detetar o tabuleiro, identificar e detetar as peças, monitorizar a existência de movimento sobre o tabuleiro, entre outros.

Paweł Kołosowski *et al.*[1] desenvolveram um sistema que permite que dois braços robóticos joguem o jogo de xadrez entre si, um braço jogue contra um ser humano ou o ser humano insira jogadas num terminal que por sua vez são feitas fisicamente por um dos robôs associados a si.

A deteção do tabuleiro foi feita usando o detetor de cantos *Harris* para encontrar os cantos interiores do tabuleiro, ou seja, uma matriz de 7x7 cantos, esta deteção só é considerada

válida se todos os 49 cantos forem detetados. Após cada jogada é feita uma nova deteção do tabuleiro. A posição de cada quadrado é então inferida a partir da posição destes cantos.

As peças são verdes e vermelhas para facilmente serem separadas do tabuleiro. A identificação das peças é feita a partir da sua posição inicial no tabuleiro pelo que não é possível começar um jogo numa posição aleatória.

Após passagem da imagem captada pela câmara para o espaço de cor HSV e com os limites adequados para as cores vermelho e verde é possível separar os pixeis das peças dos pixeis do tabuleiro, assim, uma peça é dada como estando num determinado quadrado quando o número de pixeis da cor dessa peça nesse quadrado ultrapassa um certo limite.

Com esta informação é criada uma matriz onde cada entrada corresponde a um quadrado, cada uma destas entradas da matriz pode ser de uma de três classes: verde, vermelho ou vazio. Desta forma obtêm a localização de todas as peças no tabuleiro.

Já Cynthia Matuszek *et al.*[2] apresentam um sistema composto por um braço robótico, o *Gambit robot arm*, de custo moderado e precisão relativamente elevada, características que consideram importantes para que manipulação robótica tenha um impacto mais abrangente. Este braço apresenta 6 graus de liberdade e uma garra paralela.

A capacidade sensorial é fornecida por uma câmara PrimeSense que para além de fornecer três canais de cor RGB fornece também um valor de distância por cada pixel no intervalo $[0.5, 5]m$.

A localização do tabuleiro de jogo é feita atráves da deteção dos cantos do tabuleiro na imagem 2D (RGB), a informação de profundidade é depois incorporada nestes pontos. De seguida e com o auxílio de um algoritmo Random Sample Consensus (RANSAC) é procurado o plano que melhor encaixa nos vários pontos que correspondem aos cantos do tabuleiro. Para obter a localização exata do tabuleiro estes vários pontos são comparados com uma grelha *template* de 8x8 células para que a melhor correspondência retorne a localização do tabuleiro nos vários pontos previamente filtrados pelo método RANSAC.

Com os pontos da *point cloud* imediatamente acima do plano do tabuleiro é determinada a ocupação das peças neste, para isso estes pontos são projetados no plano do tabuleiro por forma a criar uma imagem 2D com a projeção de todas as peças. Os vários pontos projetados são agrupados em *clusters* correspondendo cada um a uma peça, estes *clusters* são de seguida atribuídos a uma célula na grelha 8x8, encontrada anteriormente, que corresponde ao quadrado que a peça ocupa.

Sempre que existir movimento por cima do tabuleiro a deteção da ocupação das peças é pausada. Para que a cor da peça seja atribuída é calculada a cor média das *point clouds* correspondentes às peças, as cores identificadas são preto ou branco.

Este estudo não possui deteção das peças no tabuleiro num instante qualquer do jogo, pelo que é necessário as peças estarem no estado inicial conhecido à partida para que a sua identificação seja possível, ainda assim implementaram um sistema de deteção de peças com o objetivo de no futuro vir a integrar um detetor de peças num estado qualquer. Este detetor é composto por quatro classificadores hierárquicos: um detetor de quadrados, um detetor binário de fundo/peça, um classificador de cor (branco/preto) e dois classificadores do tipo de

peça cada um com seis classes {B,N,K,P,Q,R}, cada uma destas classes corresponde a um tipo de peça do xadrez.

Estes classificadores seguem a hierarquia reprensentada na figura 2.1 para que a deteção final seja conseguida.

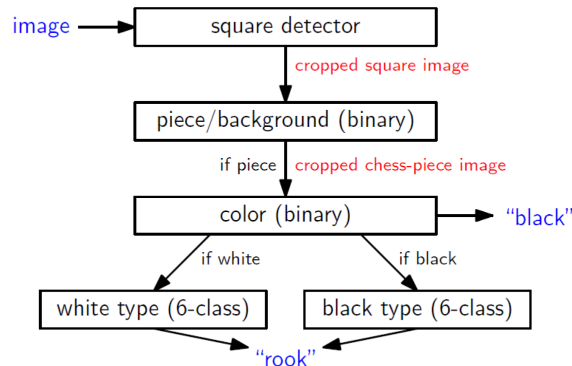


Figura 2.1: Hierarquia dos classificadores para a deteção das peças no tabuleiro, imagem retirada de [2].

Aos classificadores são fornecidas imagens de 640x480 pixeis. O detetor de quadrados procura quadrados independentemente da rotação destes na imagem. Por forma a aumentar a robustez do detetor à rotação do quadrado foram criadas 20 imagens 220x220 de um quadrado com várias rotações e de seguida por cada imagem foi extraído o *Histogram of Oriented Gradients (HOG)*, estes histogramas permitem detetar mais facilmente quadrados numa imagem e formam o *training set*. Na fase de teste para procurar os quadrados na imagem foram criadas várias imagens mais pequenas através do método *sliding window*, cada uma destas imagens foi avaliada conforme uma função de custo, aquelas cujo resultado fosse superior a um *threshold* eram incorporadas na imagem final, imagem esta composta pelo contorno do quadrado.

Para compensar o tamanho do tabuleiro real e assim também o tamanho dos quadrados, o detetor de quadrados é aplicado 20 vezes com várias variações de escala nas imagens recolhidas em direto para que a escala destas fosse o mais próxima das imagens de treino.

O detetor que separa a peça do plano de fundo, ou seja, do tabuleiro, foi treinado com as mesmas imagens que o detetor de quadrados e com peças no seu centro para os casos positivos, para os casos negativos foram usadas imagens dos quadrados do tabuleiro com o mesmo tamanho mas sem peça.

O classificador treinado foi um Suport Vector Machine (SVM) binário que define o pixel da imagem como sendo peça ou não peça.

Para diferenciar a cor das peças foi criada uma Region of Interest (ROI) no centro de uma imagem com peça e com a forma de um quadrado. Os pixeis pertencentes a esse quadrado são os dados fornecidos ao classificador para treino, de forma a que a cor da peça seja descoberta, as possibilidades são branco ou preto.

Com base na deteção da cor o classificador seguinte é escolhido, assim, um classificador diferencia os vários tipos de peças de cor branca enquanto o outro diferencia os tipos de peças

de cor preta. Desta forma o treino de cada um destes classificadores tem por base apenas peças da mesma cor.

No sistema robótico, para jogar xadrez com base em visão por computador, implementado por Hafiz Luqman e Mubeen Zaffar [3] a câmara é posicionada sobre o tabuleiro, de modo a ter visão sobre todo este e a imagem é restringida à área do tabuleiro para poupar tempo de processamento. O processamento de imagem é feito em imagens de tons de cinzento, pelo que a captação da câmara teve de ser convertida. Foi usado um filtro de gradiente para redução de sombras aumentando assim a resiliência dos algoritmos seguintes a mudanças de iluminação no tabuleiro e nas peças.

Para a deteção do tabuleiro começam por aplicar um detetor de arestas à imagem em tons de cinzento. O detetor usado foi o *canny edge detection algorithm.* As cores dos quadrados foram escolhidas para que existisse o maior contraste possível entre arestas quando em tons de cinzento, neste caso foi usado o laranja para os quadrados escuros e branco para os claros com a borda do tabuleiro a preto. Após a segmentação das arestas estas são dilatadas através do operador morfológico para que fiquem mais evidentes. O resultado deste processo pode ser visto na figura 2.2.

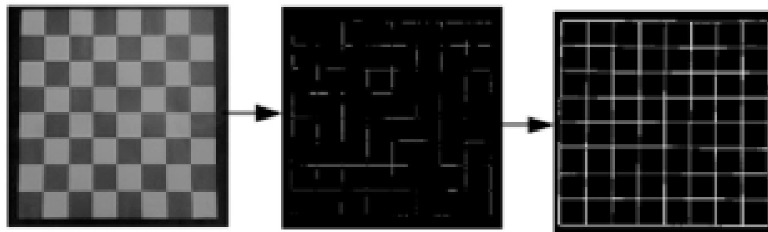


Figura 2.2: A primeira imagem mostra a captação do tabuleiro, a segunda corresponde ao resultado do detetor de arestas e na última pode-se ver a aplicação de dilatação à imagem anterior, imagem adaptada de [3].

2.3 Deteção de jogadas

A deteção de jogadas consiste em interpretar a informação recolhida pelos sistemas sensoriais, na sua maior parte câmaras, de modo a que seja descoberta a jogada feita pelo humano.

Paweł Kołosowski *et al.* [1] detetaram as jogadas através da comparação entre uma matriz antes de uma jogada ser efetuada e de uma matriz após a jogada ter sido efetuada, ver figura 2.3, esta matriz representa o estado do jogo.

A deteção de uma jogada tem por base a monitorização da presença de movimento do humano no tabuleiro, assim que a mão do humano sai para fora do campo de visão da câmara é lançado um evento para determinar a posição das peças no tabuleiro e criar a nova matriz e assim descobrir a jogada feita.

A abordagem tomada por Cynthia Matuszek *et al.*[2] foi a de criar um tuplo por cada célula do tabuleiro: $(s_i, c_i, p_i)_t$ onde s é uma variável binária e representa a ocupação ou não dessa célula por uma peça, c corresponde à cor e p define o tipo de peça de entre as várias

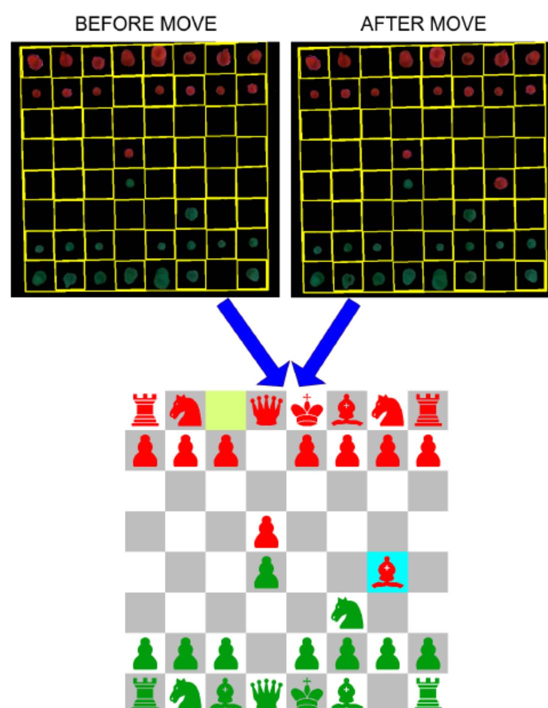


Figura 2.3: Exemplo da deteção da jogada a partir do estado do jogo antes e depois, imagem retirada de [1].

possibilidades do jogo de xadrez, finalmente t corresponde à iteração temporal. Para que jogadas possam ser detetadas são comparados dois estados do tabuleiro consecutivos, se a mudança detetada for inválida é pedido ao humano que corrija a jogada.

No início de um jogo é criada uma matriz de ocupação com a informação das peças que ocupam certos quadrados, assim como a informação das alturas de cada peça, pelo que subsequentemente é apenas necessário extrair a informação de cor e profundidade para os vários estados seguintes do jogo, de forma a que as jogadas sejam detetadas a partir das variações entre estados.

Na situação criada por Hafiz Luqman e Mubeen Zaffar[3] o vídeo é constantemente monitorizado para procurar movimento da mão. Assim que o movimento termina é tirada uma imagem que é por sua vez subtraída à imagem anterior à movimentação da mão para que seja descoberta a jogada do humano. O resultado obtido nesta subtração são os quadrados de partida e chegada da peça movida como pode ser visto na figura 2.4.

As coordenadas dos quadrados de partida e chegada são inferidos a partir da imagem 2.4 e traduzidos para a notação algébrica de xadrez. A jogada do humano é posteriormente enviada para um motor do jogo de xadrez para que este possa determinar a jogada seguinte, esta é depois enviada para o controlador do braço robótico.

Outra abordagem tiveram Ekaterina E. Kopets *et al.*[4] ao usarem uma forma de deteção das peças no tabuleiro através de ímans em cada quadrado do tabuleiro, isto foi possível pois, ao contrário da generalidade, o jogo implementado foram as damas russas, jogo que não necessita de uma identificação individual das peças, apenas que se separem as peças de

jogadores opostos.

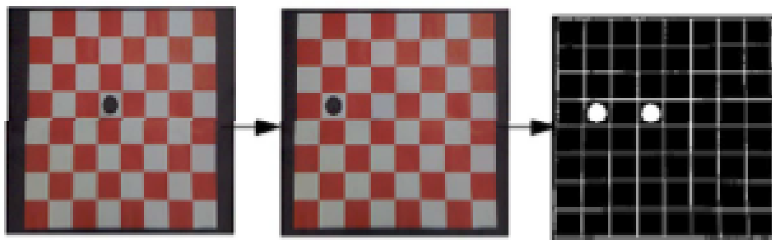


Figura 2.4: Resultado da subtração entre a imagem anterior ao movimento do humano e após, imagem adaptada de [3].

2.4 Decisão

A decisão consiste em dotar um sistema robótico da capacidade de criar uma resposta a um situação gerada pelo humano, neste caso, de fazer um jogada no jogo que está a decorrer com o humano e que esta faça sentido no contexto da jogada feita pelo humano.

A implementação de Paweł Kołosowski *et al.*[1] para a decisão da jogada a ser feita pelo robô consiste no uso de um algoritmo de *minimax* que vai avaliando o estado do jogo após várias jogadas até uma certa profundidade, esta avaliação é feita com base na tabela 2.1, onde cada peça tem uma certa pontuação com base na sua importância para o jogo.

Piece	*Pawn*	*Knight*	*Bishop*	*Rook*	*Queen*	*King*
Value	100	300	350	500	1000	∞

Tabela 2.1: Valores usados para avaliação do tabuleiro após uma jogada, tabela retirada de [1].

Implementaram este método com *alpha-beta pruning* para aumentar a eficiência na procura da melhor jogada para o robô. O resultado obtido é um sistema que mostra funcionar bem em ambiente real.

2.5 Manipulação

A manipulação das peças no sistema de Cynthia Matuszek *et al.*[2] é feita da seguinte forma: sempre que há necessidade de mover uma peça o *end effector* é posicionado por cima da peça no centro do quadrado, de seguida é baixado até à altura da peça e a garra é fechada. Caso a peça não esteja no centro do quadrado é determinado o ajuste a ser feito através de *visual servoing* que tem por base a deteção da peça na imagem para determinar a sua posição e ângulo.

O braço robótico de Hafiz Luqman e Mubeen Zaffar[3] assenta sobre uma base que se move sobre dois eixos paralelos com o tabuleiro de jogo, desta forma quando for necessário agarrar numa peça numa determinada coluna é preciso deslizar o braço até à coluna dessa

peça, de seguida através do controlo de três articulações é que é possível atingir a linha onde essa peça reside.

Isto deve-se ao facto da base do robô não rodar sobre si própria para que os movimentos executados sejam menos complexos. Apenas os movimentos de mudar uma peça de sítio e retirar uma peça do tabuleiro foram implementadas.

Assim movimentos específicos do xadrez como são o *en-passant* ou o roque o braço não tem capacidade de os executar.

CAPÍTULO 3

Arquitetura da solução

3.1 Introdução

Este capítulo aborda a arquitetura implementada de um ponto de vista de alto nível, para que seja claro o papel desempenhado pelos vários módulos, como é que eles interagem entre si e qual a sua importância no sistema final.

O problema que é proposto ser resolvido é o de rapidamente adaptar um braço robótico a jogar um jogo de tabuleiro qualquer, sem que para isso seja necessário programar um sistema de raiz para cada jogo e desta forma evitar criar uma arquitetura nova para interagir com o braço ou para processar a informação recolhida por parte da câmara, cada vez que um jogo é adicionado.

O ideal é criar um sistema onde a maior parte dos seus componentes é partilhado entre os vários jogos, sendo apenas necessário acoplar um motor de jogo por cada jogo adicionado. No limite pode ser necessário acoplar mais um componente de identificação individual das peças, caso o jogo o necessite, como acontece por exemplo no xadrez onde as peças não são todas iguais.

O desenvolvimento de um sistema robótico que jogue jogos de tabuleiro contra humanos é um problema que já conta com um progresso substancial, como são exemplo os estudos expostos no capítulo 2, no entanto é fácil verificar a primazia dada ao xadrez em detrimento de outros jogos de tabuleiro. Isto poderá ser fruto da fama do xadrez que facilita o reconhecimento por parte do público em geral mas também porque sendo um jogo com um grau de dificuldade elevado as suas regras são no essencial bastante simples.

Não deixa de ser interessante abordar jogos diferentes, que não tendo a complexidade em termos de peças que existe no xadrez têm a vantagem de apresentar novas mecânicas de movimentação de peças. Até porque, como já foi possível verificar, muitas vezes o problema da identificação das peças é contornado ao se considerar que um jogo tem necessariamente de começar na posição inicial, onde a localização das peças é conhecida à partida, eliminando assim a questão de distinguir uma peça de outra visualmente.

No caso específico desta tese os jogos implementados foram: o jogo das damas(versão Inglesa), que tem a particularidade da peça dama só se mexer uma casa de cada vez mas em qualquer direção, e o jogo dos peões.

Ambos os jogos foram escolhidos pois permitem a prova de conceito mais rapidamente, ao eliminar a necessidade de distinguir peças complexas e estas poderem ter formas simples e fáceis de agarrar. A escolha da versão Inglesa do jogo das damas baseou-se no facto de existir um motor de jogo disponível para essa versão que verificava as necessidades deste sistema.

3.2 Restrições

Atualmente, neste sistema só foi implementado o tabuleiro de xadrez, que é composto por uma matriz de 8x8 quadrados pretos e brancos. A escolha deste tabuleiro deve-se ao facto de ser a plataforma base para vários jogos, o que permitiria rapidamente a implementação de novos jogos facilitando assim a prova de conceito aqui proposto. A integração de diferentes tabuleiros necessitaria de um gasto adicional de tempo que conflituaria com o desenvolvimento do restante sistema.

A forma das peças convém ser prismática ou cilíndrica, com pelos menos 2 cm de altura para facilitar o agarrar por parte da garra do braço. Para já apenas foram implementados dois tipos de peças diferentes para serem usadas por jogo, que são distinguidas pela sua altura, o topo da peça tem de ser totalmente pintado com a cor azul para um jogador e vermelho para o outro.

Na figura 3.1 podem ser vistas 4 peças criadas segundo estas regras para serem usadas neste sistema, as mais pequenas têm 2 cm de altura enquanto que as mais altas têm 4 cm. Estas peças forçam a que o robô tenha apenas dois modos de pegar nas mesmas, podendo agarrá-las por duas faces opostas.

O tampo da mesa sobre qual o tabuleiro está poisado deve ser o mais plano possível para permitir a sua remoção por filtragem da *point cloud*, que será mais à frente explicado.

Embora o posicionamento da câmara possa ser feito de um qualquer lado do tabuleiro, esta tem obrigatoriamente de estar numa posição superior ao tabuleiro mas a menos de 1,2 m do tampo da mesa e deve abranger a totalidade da zona de jogo que corresponde ao tabuleiro e à área envolvente, onde serão posicionadas as peças fora do tabuleiro. O ângulo entre a câmara e a normal do plano da mesa deve ser o menor possível para permitir a deteção correta do tabuleiro, no entanto alguma liberdade é permitida neste ângulo.

Na zona delimitada por um quadrado com 60 cm de lado e centrado no centro do tabuleiro não devem existir objetos com altura superior a 10 cm visto ser esta a área monitorizada pelo sistema para a deteção de movimento por parte do humano ou do robô.

No caso de novas peças serem necessárias para diferentes jogos estas terão de ser implementadas para que sejam detetadas segundo as suas características de altura, forma ou cor.

O tabuleiro deve-se manter na mesma posição enquanto um jogo decorre, caso contrário o jogador humano pode forçar uma nova deteção do tabuleiro.

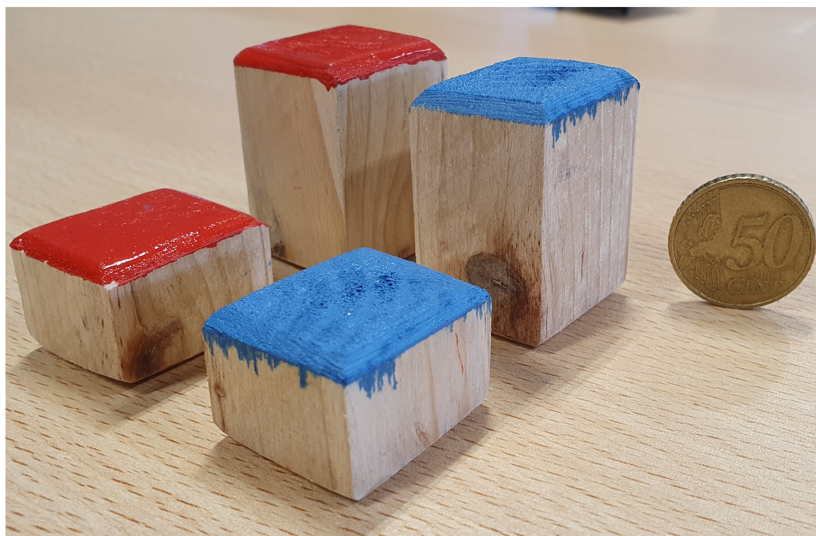


Figura 3.1: Peças criadas para uso nos vários jogos, as peças do canto superior direito têm 4 cm de altura enquanto que as peças do canto inferior esquerdo têm 2 cm, vistas de topo a forma é retangular.

3.3 Similaridades entre diferentes jogos de tabuleiro

Com a intenção de criar um sistema facilmente adaptável a vários jogos com o mínimo de esforço é importante identificar que partes esses jogos partilham, especialmente aqueles implementados nesta tese, de forma a criar uma arquitetura flexível mas o mais abrangente possível.

Assim, nos dois jogos implementados, existe um tabuleiro de jogo com um padrão xadrez de 8x8 quadrados. Este tabuleiro está sobre uma mesa que auxilia o jogo no posicionamento das peças que são capturadas e têm de ser removidas do tabuleiro ou peças que não estão ainda em jogo.

As peças nestes dois jogos têm a particularidade de não serem diferentes entre si, com a exceção da dama que num jogo normal costumam ser duas peças empilhadas mas por uma questão de facilitar o agarrar por parte do braço robótico foi criada uma peça única com o dobro da altura de uma normal. Com estas peças não há necessidade de fazer uma identificação complexa como seria o caso do xadrez, chega conseguir separar uma peça normal de uma dama pela sua altura.

Estas peças foram criadas para facilitar o agarrar do robô pelo que são prismas quadrangulares e são distinguidas entre os dois jogadores pela cor do topo. As cores escolhidas foram o vermelho e o azul para permitir uma fácil identificação por computador e para ter algum contraste com o preto e branco do tabuleiro.

3.4 Estrutura

A arquitetura do sistema está presente na figura 3.2 e nesta, é possível notar com algum detalhe o fluxo de informação na implementação desta arquitetura. Esta informação é, inicialmente, recolhida por uma câmara, sendo esta a única forma de perceção do mundo

real por parte do sistema. De seguida, é necessário processar esta informação para que sejam recolhidos dados importantes, como são a posição do tabuleiro no mundo e a posição da peças, com estas duas partes é construído o estado do tabuleiro. Estados consecutivos são então comparados para que seja detetada a jogada feita.

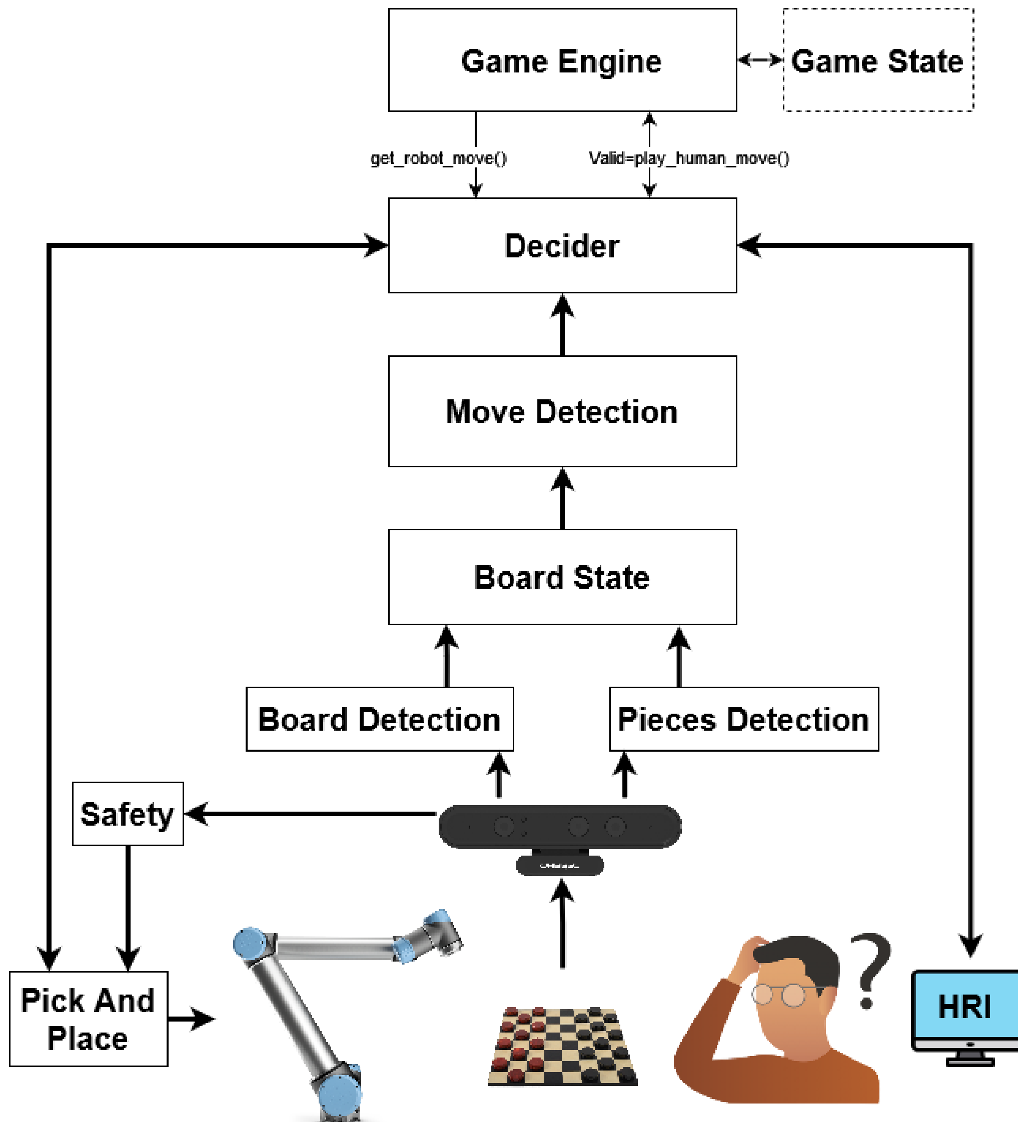


Figura 3.2: Arquitetura geral do sistema para jogar jogos de tabuleiro com um braço robótico.

Ao decisor corresponde a capacidade de sincronizar as várias partes do sistema, seja para informar o humano quando fez um jogada inválida, para enviar a jogada do humano para o motor de jogo, iniciar um novo jogo por ordem do humano, pedir ao motor de jogo a jogada do braço robótico, enviar a jogada recebida do motor de jogo para o braço executar, etc.

O motor de jogo engloba toda a parte de decisão para criar uma jogada para o braço robótico mas também para validar a jogada do humano e manter o estado do jogo atualizado com base na informação que recebe e envia.

Finalmente, a manipulação combina a informação da jogada a fazer com a informação do

posicionamento real das peças para que o braço as possa movimentar. Esta movimentação deve ocorrer sempre que o humano se encontrar numa posição segura e fora do alcance do movimento do robô, este módulo não foi, no entanto, implementado na íntegra, não estando apto para detetar a diferença entre movimentos do humano ou do braço robótico.

O humano pode efetuar várias ações através da HRI, como por exemplo iniciar um novo jogo, decidir as cores com que cada jogador joga, quem começa a jogar, entre outras.

De seguida são apresentados os módulos um pouco mais detalhadamente, sendo que alguns, pelo facto de terem capítulos dedicados, serão abordados com o mesmo detalhe apenas mais à frente.

3.4.1 Deteção do tabuleiro

Esta arquitetura começa pela deteção do tabuleiro, que é feita antes do início de um jogo e apenas uma vez, embora sejam possiveis posteriores deteções a mando do jogador humano no caso do tabuleiro sair da sua posição original, com o tabuleiro detetado a localização deste e de todos os quadrados compostos por ele são transmitidos ao sistema. A arquitetura inclui um módulo que ativaria esta deteção do tabuleiro sempre que este se tivesse movido para fora da sua posição, no entanto, acabou por não ser implementado por ser um objetivo de menor importância para o sistema final.

3.4.2 Deteção das peças

Também com base na informação recolhida pela câmara, a deteção das peças ocorre através da segmentação destas na *point cloud* original, para isso o tampo da mesa, o tabuleiro e o chão devem ser removidos da *point cloud*. Com os pontos das peças segmentados da *point cloud* têm de ser criados *clusters* por cada peça para que a posição destas possa ser determinada.

3.4.3 Estado do tabuleiro

Com as posições das várias peças e em conjunto com a informação do posicionamento do tabuleiro as peças podem ser identificadas unicamente entre si com base no quadrado que ocupam no tabuleiro. Esta informação permitirá detetar as jogadas do humano por comparação entre dois estados de jogo consecutivos mas também encontrar um peça com base no nome do quadrado que ocupa para que o robô a consiga mexer.

Estes três módulos serão explicados mais detalhadamente no capítulo 4.

3.4.4 Deteção da jogada

A deteção da jogada significa descobrir que peça foi movida no tabuleiro, ou seja, descobrir o quadrado de onde partiu e o quadrado para onde a peça foi posta de novo, no entanto, esta informação pode não ser suficiente para que um motor de jogo consiga perceber a jogada que foi feita, por exemplo nas damas é possível uma peça capturar em cadeia, quando isto acontece o quadrado detetado como partida e o quadrado detetado como chegada não correspondem a uma jogada válida no motor de jogo, este necessita dos vários saltos individuais feitos um de cada vez para poder ir mantendo o estado do jogo atualizado e validar as várias jogadas. No capítulo 5 esta característica será de novo abordada mais profundamente.

3.4.5 Decisor

O decisor é um módulo de sincronização que serve principalmente para pedir ao motor de jogo a jogada de resposta que deve enviar ao braço rótico para que este a execute, mas este pedido tem apenas de acontecer após a jogada do humano ter sido feita e validada, da mesma forma só deve enviar a jogada recebida a partir do módulo deteção de jogada quando for a vez do humano jogar caso contrário a jogada do robô seria também ela detetada e enviada de novo para o motor de jogo. É este módulo que deve receber os vários *inputs* do utilizador para definir a construção da arquitetura de maneira a que esta funcione com base na informação atual e real, por exemplo, caso se mude de jogo o utilizador deve usar este canal para atualizar a arquitetura.

3.4.6 Motor de jogo

O motor de jogo recebe as jogadas feitas pelo humano e determina a jogada que o braço robótico deve fazer, este motor deve respeitar as regras do jogo na íntegra, serve também para validar as jogadas feitas pelo humano, avisando-o quando cometer um erro, no capítulo 5 este tema será mais amplamente abordado.

3.4.7 Manipulação

Finalmente a jogada decidida pelo motor é por sua vez enviada para o robô para que seja feita na realidade. Para isso foi criado um serviço com a estrutura da listagem 3.1, este serviço recebe no argumento *src_dst_labels* o nome do quadrado onde está a peça que deve ser movida pelo braço, seguido dos nomes dos quadrados para onde o braço deve ir com a peça, sendo o último quadrado aquele onde este a deve largar.

No caso das damas pode ser feita mais do que uma captura seguida e para representar a captura de várias peças o braço deve na mesma de se mover até aos quadrados intermédios para mostrar o salto que captura uma determinada peça.

Por exemplo, se for enviado o *array* [**c3**,**g5**,**e7**] isto significa que a peça a mover está em **c3**, o braço deve movê-la até **g5**, não a pode largar neste quadrado pois não é o último e de seguida deve mover esta peça até ao quadrado **e7**, aqui sim sendo o último quadrado do *array* já pode largar a peça.

Listagem 3.1: Protótipo da mensagem *PlayEngineMove*.

```
string[] src_dst_labels
string[] eaten_labels
bool promote_dama
bool eat_first
---
bool success
```

O *array eaten_labels* corresponde aos quadrados onde se localizam as peças adversárias capturadas pela jogada do motor de jogo e assim do braço robótico. Estas devem ser removidas do tabuleiro para a zona fora deste.

O argumento *promote_dama* indica se esta jogada inclui uma promoção da peça a dama, neste caso a peça quando chegar ao último quadrado de *src_dst_labels* deve ser removida do tabuleiro e o braço deve ir buscar uma das peças que representam a dama e que estão localizadas fora do tabuleiro no início do jogo.

O argumento *eat_first* serve para informar o braço da ordem de operações que deve tomar, no caso das damas a remoção das peças deve ocorrer depois de se mover a peça do jogador que captura, isto porque estas peças têm de ir obrigatoriamente para quadrados livres de peças. No entanto, existem outros jogos, como são o caso do jogo dos peões ou do xadrez, onde a captura acontece quando a peça que captura passa a ocupar o quadrado da peça capturada, devendo neste caso o robô remover a peça capturada primeiro para que seja de seguida movida a sua peça para esse quadrado.

Este serviço retorna o sucesso ou não da execução da jogada do braço robótico.

Como forma de apoio à execução das jogadas do braço foram criados outros serviços para obter a posição no mundo da peça a mover ou das peças a capturar, para pesquisar a posição no mundo de um quadrado vazio para onde uma peça deve ser jogada, para procurar a localização das damas fora do tabuleiro e para obter uma posição fora do tabuleiro onde as peças capturadas devem ser postas.

Assim, o serviço *get_outside_position* retorna uma posição no mundo onde o braço pode poisar as peças capturadas. Estas posições são determinadas com base no tamanho do tabuleiro e sempre do lado do tabuleiro mais próximo do braço robótico. Foram criadas, então, 12 posições com a disposição no mundo representada na figura 3.3, estas estão sempre no alinhamento das arestas dos quadrados do tabuleiro sendo que horizontalmente estão de dois em dois quadrados e verticalmente a distância entre estas posições é de apenas um quadrado.

Esta disposição foi considerada pois é uma forma simples de garantir que o robô posiciona as peças no tampo da mesa e não numa zona fora da mesa, para isso apenas tem de se garantir que do lado do robô existe espaço equivalente a metade da largura do tabuleiro usado.

Sempre que necessário o serviço retorna uma destas posições memorizando aquelas que já estão ocupadas para que o robô não posicione duas peças no mesmo sítio.

O serviço *get_obj_position* procura no estado do tabuleiro, que será determinado no capítulo 4, o quadrado da peça a mover ou das peças a capturar apenas e só com base na *label* ou nome deste quadrado.

Para obter a posição da dama o serviço anterior não serve, isto porque o serviço anterior espera como argumento o nome de um quadrado do tabuleiro, enquanto que neste caso a peça se encontra fora dele. As peças fora do tabuleiro possuem uma *label* especial para indicar que a mesma não se encontra no tabuleiro, para além desta *label* o estado do tabuleiro possui, para todas as peças, a sua localização no mundo, o tipo de peça e a sua cor. Ora mais uma vez se verifica que pedir ao serviço anterior a peça com a *label* que a identifica como estando fora do tabuleiro não é suficiente pois podem estar vários tipos de peças fora do tabuleiro e de várias cores.

Com isto em mente foi criado o serviço *get_dama_position*, este tem apenas um argumento, a cor da peça, esta simplificação pode ser feita porque já se sabe à partida que a peça está

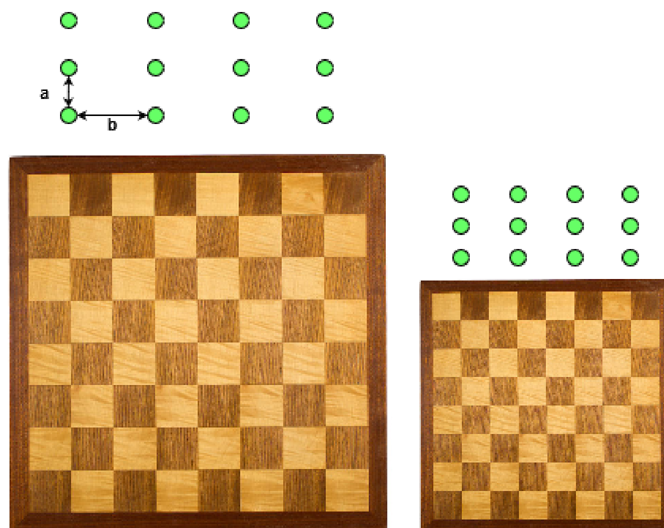


Figura 3.3: Os 12 círculos correspondem às posições pré-determinadas para poisar as peças capturadas fora do tabuleiro, a distância **a** entre eles corresponde à largura de um dos quadrados do tabuleiro e a distância **b** à largura de dois quadrados, desta forma garante-se que as posições são adaptáveis ao tamanho do tabuleiro.

fora do tabuleiro e é uma dama pelo que não há necessidade de enviar esta informação, sendo suficiente pedir a cor da dama pretendida.

Finalmente, o serviço *get_label_position* retorna a posição no mundo de um quadrado do tabuleiro com base na *label* que o identifica.

A forma de identificação dos quadrados do tabuleiro será alvo de análise no capítulo 4, assim como o estado do tabuleiro usado por estes serviços para obter a informação das peças com base no quadrado que ocupam, na sua cor ou no tipo de peça.

Capítulo 4

Perceção

4.1 Introdução

Este capítulo descreve o processamento feito à captação de vídeo obtido a partir de uma câmara para obtenção dos vários elementos que compõem um jogo normal, como são: o tabuleiro, as peças, a deteção de movimento por parte do braço robótico e do jogador humano, mas acima de tudo o estado do tabuleiro para monitorização do mesmo.

Para que se atinja o objetivo de obter o estado do tabuleiro duas importantes deteções têm de ser feitas, a deteção do tabuleiro de jogo, feita antes do começo do jogo e apenas uma vez e a deteção das peças, feita sempre que não há movimento sobre o tabuleiro que obstrua a visão da câmara. O diagrama da figura 4.1 apresenta a estrutura criada para que o estado do tabuleiro seja determinado. Do lado esquerdo, a deteção do tabuleiro, que usa para isso apenas a captação de vídeo seguida de uma deteção dos cantos presentes na imagem e a filtragem dos cantos fora do tabuleiro. Com estes cantos é possível inferir a posição do tabuleiro e construir assim os limites dos quadrados que o compõem. Do lado direito, na captação da *point cloud* é constantemente procurado o plano da mesa, este é então removido e de seguida filtra-se a *point cloud* para que restem apenas os pontos pertencentes às peças. Finalmente no estado do tabuleiro os pontos das peças são agrupados e a sua posição comparada com as posições dos quadrados do tabuleiro para que as peças sejam identificadas e assim determinado o estado do tabuleiro.

A câmara usada foi a Orbbec Astra que disponibiliza duas captações diferentes, a primeira corresponde a uma captação de vídeo no espaço de cor RGB, a segunda a uma captação de vídeo no formato *point cloud*, isto significa que por cada pixel é possível obter a informação XYZ no referencial da câmara. É ainda possível obter uma captação mista das duas onde cada ponto, para além da informação XYZ, tem a cor RGB.

A câmara é a única forma de o robô ter perceção sobre o estado do tabuleiro, o que o rodeia ou a localização das peças a mover, a perceção é assim uma parte importantíssima do sistema geral.

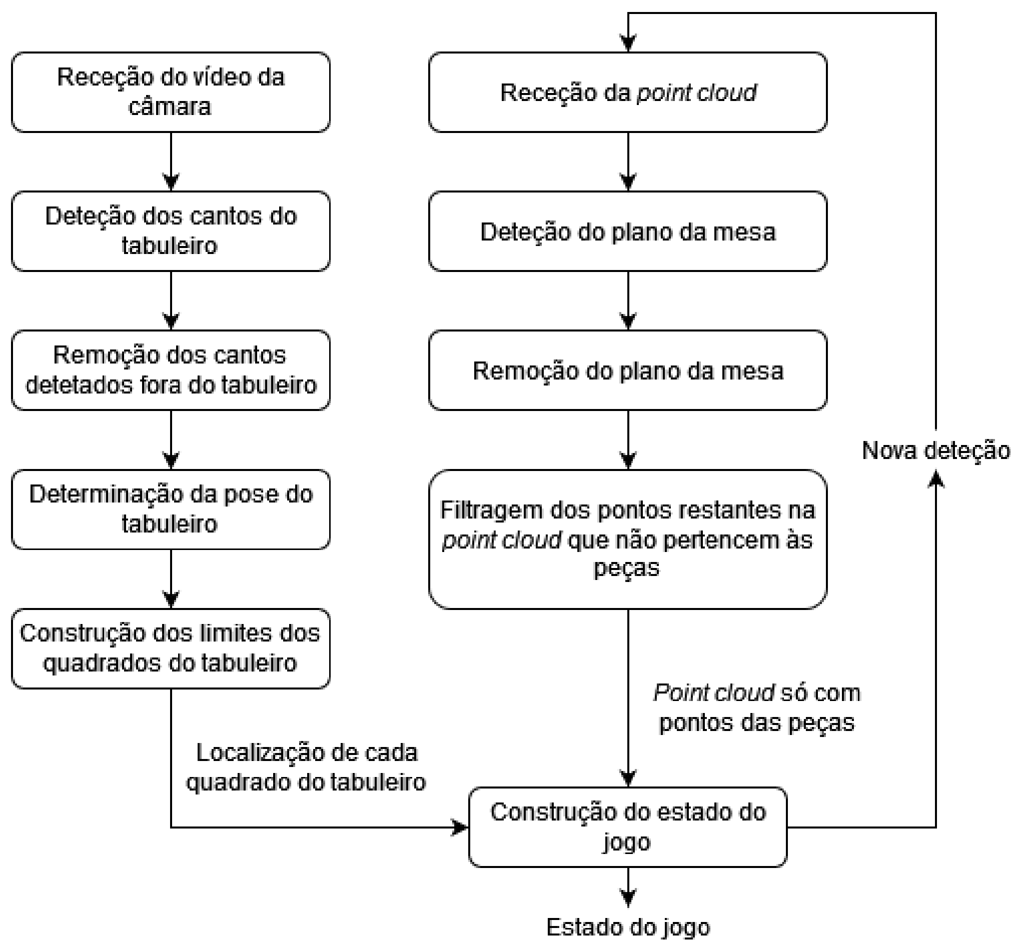


Figura 4.1: Estrutura para determinação do estado do tabuleiro.

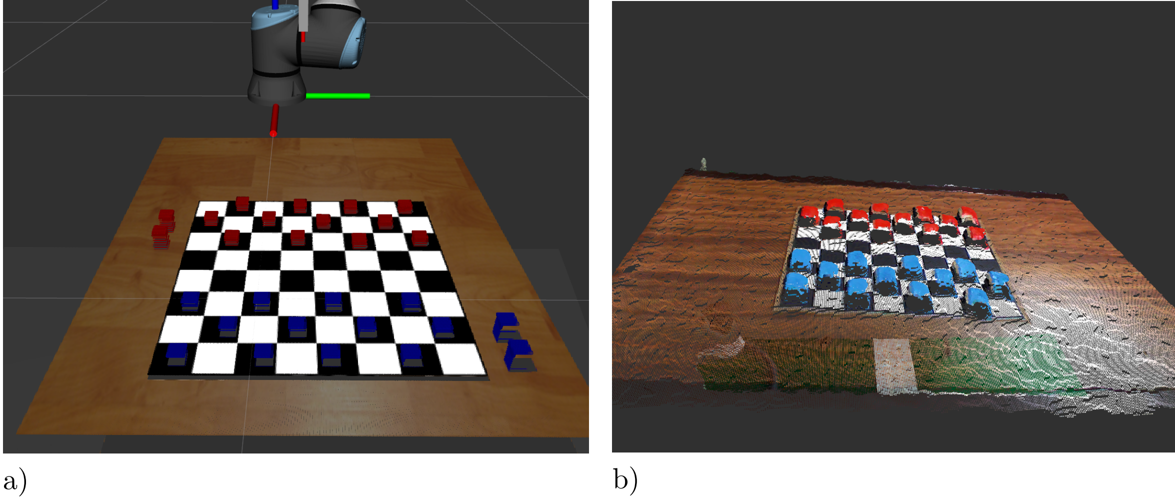
No entanto ter acesso ao vídeo que é recolhido pela câmara ou à *point cloud* é pouco útil se mais nenhum processamento for feito que permita identificar os elementos importantes em ambos, isto porque muitos dos pontos contidos na *point cloud* não são necessários, por exemplo os pontos do chão ou da mesa, mas também porque os pontos que pertencem a uma peça são dificilmente discerníveis dos pontos da mesa, tornando a aplicação de algoritmos de *clustering*, para a segmentação das várias peças, impraticável. Assim, é necessário fazer uma filtragem para remover estes pontos indesejáveis.

4.2 Filtragem da *point cloud*

Antes de se aplicar uma filtragem é preciso identificar aquilo que se quer ou não filtrar. A figura 4.2 mostra dois exemplos de captações de *point clouds*, uma em ambiente de simulação e outra em ambiente real, onde será aplicada a filtragem para obtenção das peças.

É assim o propósito desta filtragem remover todos os pontos que não pertencem às peças, ou seja, remover todos os pontos que pertencem à mesa, chão, objectos fora da mesa, etc.

Se imaginarmos a origem de um referencial cartesiano com 3 dimensões cujo plano XY corresponde ao plano da mesa e se todos os pontos estiverem neste referencial, é fácil concluir



a) b)

Figura 4.2: a) *point cloud* captada em ambiente de simulação, repare-se na perfeição do tampo da mesa que mostra a ausência de ruído. b) *point cloud* captada em ambiente real, note-se agora a presença de ruído, por exemplo, nas irregularidades do tampo da mesa.

que a filtragem se resume a limitar os valores da componente z. É necessário, então, fazer a transformação dos pontos de modo a passá-los do referencial da câmara para o referencial da mesa.

4.2.1 Deteção do plano da mesa

Para que se crie um referencial com origem no plano da mesa é preciso identificar um ponto que pertença a esse plano e que funcione como origem, para isso é preciso saber quais os coeficientes que definem esse plano. Com a ajuda da *Point Cloud Library (PCL)* é possível aplicar o método de *RANSAC* com um modelo de planos para que retorne o maior plano encontrado na *point cloud* e assim também os seus coeficientes.

Este método extrai aleatoriamente 3 pontos da nuvem com os quais cria um plano, de seguida procura nos restantes pontos aqueles que também pertencem a este plano, ou seja, pontos cuja distância ao plano seja inferior a um parâmetro fornecido pelo utilizador. O método corre várias vezes de modo a determinar primeiro o maior plano presente na nuvem de pontos, aquele que mais pontos contém, e retorna os coeficientes desse plano.

O nó responsável por determinar os coeficientes do plano da mesa é o */coef_finder*. Este nó subscreve o tópico */camera/depth/points* que é publicado pela câmara e contém a mensagem com a *point cloud*.

A esta *point cloud* é aplicado o método RANSAC para planos com um máximo de 1000 iterações e uma tolerância de $1cm$. Após a primeira passagem pela *point cloud* os quatro coeficientes do plano são usados para determinar a interseção do plano com o eixo z da câmara através da equação geral do plano 4.1. É usado o eixo z porque que neste caso corresponde à direção apontada pela lente da câmara, eixo azul da figura 4.3.

$$Ax + By + Cz + D = 0 \tag{4.1}$$

Para obter o ponto de interseção basta substituir $x = 0$, $y = 0$ e resolver em ordem a z. Com o ponto de interseção é possível agora verificar se o maior plano detetado corresponde à mesa ou se terá sido por exemplo o chão. Se o valor z do ponto for superior a $1.2m$ significa que a distância da câmara a esse plano é superior a $1.2m$ no eixo z pelo que se considera que o plano detetado não foi o da mesa. Os pontos que pertencem a este plano são removidos da *point cloud* e o método é aplicado de novo.

Se a distância for agora inferior a $1.2m$ é porque o plano detetado é o da mesa e assim o objetivo de encontrar os coeficientes da mesa está cumprido, os pontos deste plano sendo os da mesa são também eles removidos da *point cloud*. Os coeficientes são, de seguida, publicados no tópico */coefs*. Isto implica assim que a câmara esteja a uma distância inferior a $1.2m$ em relação à mesa e que o chão esteja a mais de $1.2m$ da câmara, como se pode ver na figura 4.3.

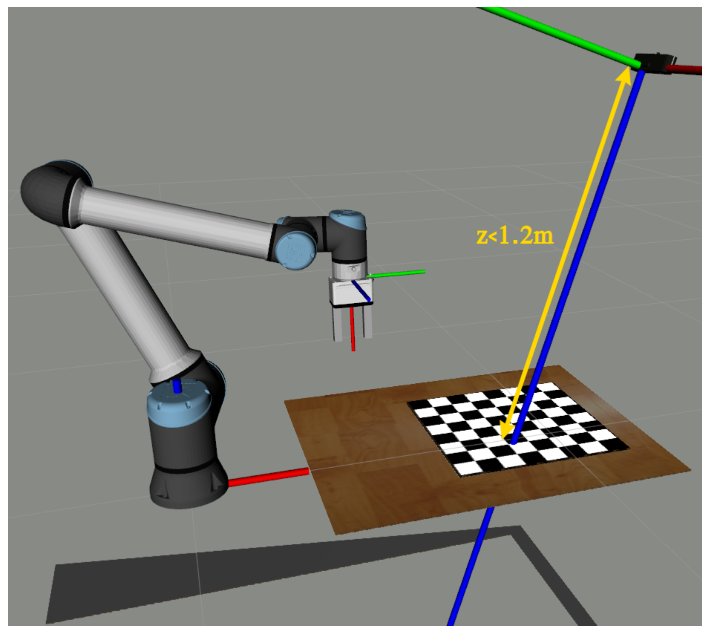


Figura 4.3: Mediação da distância entre a câmara e o plano detetado na *point cloud*. A seta amarela representa a distância máxima que câmara pode estar da mesa para que o plano desta seja detetado.

No fim do processamento deste nó podem existir dois tipos de *point clouds* distintas: a primeira sem os pontos do chão e sem os pontos da mesa, quer isto dizer que o plano do chão coincidiu ser maior que o da mesa e foi removido primeiro, no entanto, é possível obter uma segunda *point cloud* onde apenas os pontos da mesa foram removidos, significa isto que o plano da mesa foi o maior encontrado, neste caso é necessário fazer-se ainda uma outra filtragem para garantir que restam apenas os pontos das peças.

Aplicar o método RANSAC após a remoção do plano da mesa para se remover os pontos do chão não é viável, pois não é garantido que o chão seja sempre perfeitamente plano, podem existir objetos ou pessoas que interfiram e tornem os pontos fora da mesa um pouco mais espalhados pelo espaço. Este problema será corrigido mais à frente.

Esta *point cloud* é publicada no tópico */cloud_without_table*.

4.2.2 Transformação entre o *frame* da câmara e a mesa

O nó que trata de encontrar e publicar a transformação entre o *frame* da câmara e o da mesa é o */plane_tf_broadcaster* que subscreve o tópico */coefs*.

Com os coeficientes do plano da mesa é preciso escolher um ponto desse plano para servir de origem do novo referencial. Esse ponto tem $x = -0.1$, $y = -0.1$, a razão pela qual estes valores foram escolhidos é para que numa visualização gráfica a origem estivesse no tampo da mesa da simulação usada, visto que esta não se alterava, mas os valores podiam ser outros, apenas a componente z deste ponto não pode ser aleatória e foi determinada da mesma forma que na secção 4.2.1, através da equação geral do plano.

Com o ponto determinado já é possível calcular a translação entre referenciais, no entanto não permite compensar em termos de rotação, pelo que são precisos 3 vetores normais entre si, para determinar a diferença de rotação num espaço de 3 dimensões, entre a câmara e a mesa.

Define-se o vetor normal ao plano como um destes vetores e que corresponderá ao eixo z do referencial da mesa. O vetor normal ao plano é dado por $\vec{n} = (A, B, C)$ sendo A, B e C retirados dos coeficientes do plano.

Para se obter um segundo vetor perpendicular à normal do plano basta escolher um vetor pertencente ao plano, assim foi criado um novo vetor $\vec{u}$ entre o ponto de origem e um outro ponto qualquer, neste caso $p1 = (2, 2, 0)$. Este vetor é dado por $\vec{u} = p1 - origem$ e é seguidamente normalizado. Finalmente o terceiro vetor é o resultado do produto externo entre os outros dois vetores, $\vec{v} = \vec{n} \times \vec{u}$.

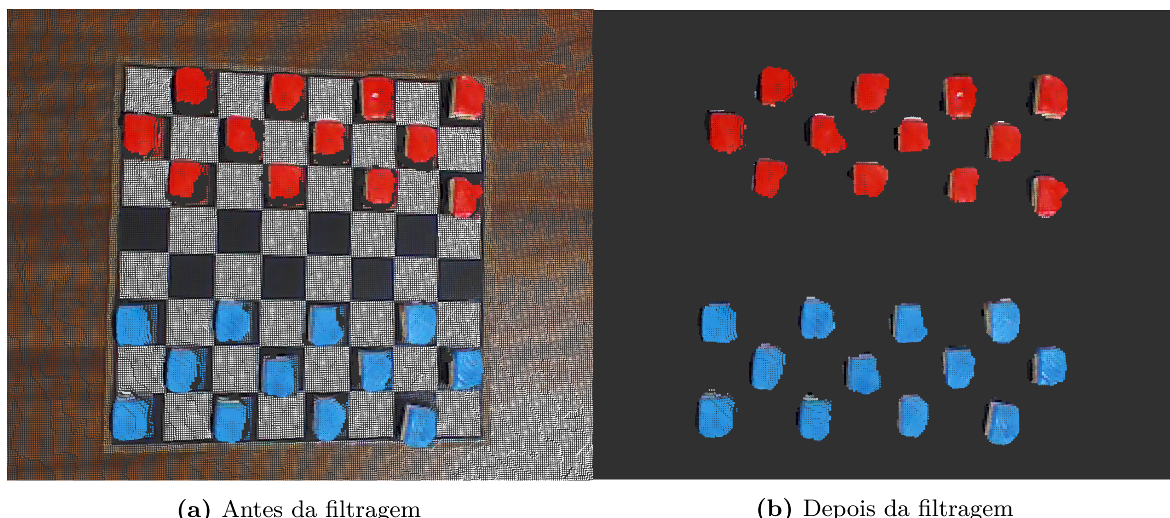
Com os três vetores cria-se um objeto tf2::Matrix3x3 [5] para que através do método getRotation() se obtenha um quaternião com a rotação entre as *frames* e assim, com a translação e rotação o nó publica uma *static transfom* entre o *frame* da câmara: *camera_rgb_optical_frame*, e o *frame* da mesa: *table_frame*.

Por fim, o nó */plane_tf_listener* subscreve o tópico */cloud_without_table*, aplica a transformação anterior e publica a *point cloud* resultante no tópico */cloud_transformed*, que está agora na *frame /table_plane*.

4.2.3 Remoção dos pontos restantes não pertencentes às peças

Com a *point cloud* no referencial da mesa é possível aplicar o último passo desta filtragem, que corrigirá a *point cloud* resultante da secção 4.2.1. Esta filtragem consiste em remover todos os pontos que estejam abaixo do tampo da mesa e acima de $25cm$ da mesa. Assim, todos os pontos com $z \in [0, 0.25]m$ não são removidos, uma vez que o referencial está no tampo da mesa e o eixo z é perpendicular ao tampo. A biblioteca PCL disponibiliza uma classe, pcl::PassThrough[6], que permite aplicar este intervalo, segundo um eixo, a toda a *point cloud*.

A filtragem resultante possui apenas pontos das peças e é publicada no tópico */objects_table* e pode ser vista na imagem 4.4b.



(a) Antes da filtragem

(b) Depois da filtragem

Figura 4.4: *Point clouds* vistas de topo antes e após filtragem ambas correspondem a captações em ambiente real. Após filtragem já só existem pontos das peças.

4.3 Deteção do tabuleiro de jogo

O tabuleiro é uma parte crucial de um jogo, é a partir dele que são definidas as posições iniciais das peças, as regras dos movimentos, situações de vitória ou derrota assim como promoção de peças quando uma certa parte do tabuleiro é alcançada, como acontece, por exemplo no jogo das damas.

O tabuleiro ganha uma importância ainda maior na perspetiva do braço robótico.

Enquanto que para um jogador humano chega conseguir distinguir as peças dele das do adversário pela cor, para um computador é necessário que cada peça tenha uma identidade única, que permita que essa peça seja distinguida de todas as outras. Isto porque quando um humano decide jogar um peça, ele sabe que peça mover, não precisa de identificar por exemplo o quadrado onde ela está, no entanto quando o braço robótico decide fazer uma jogada ele precisa de saber exatamente que peça mover e para onde.

Mas mais do que saber identificar a peça o braço precisa de saber a localização no mundo para a conseguir agarrar.

Se as peças forem identificadas pelo quadrado do tabuleiro onde estão e se o nome desse quadrado seguir a notação algébrica do jogo de xadrez, como na figura 4.5 isto resolve a primeira parte.

No entanto para resolver a segunda é necessário saber onde no tabuleiro é que o centroide de uma peça se encontra para que esta seja localizada, e para isso é preciso saber a localização no mundo real dos limites de cada um dos quadrados do tabuleiro, se o centroide estiver dentro desses limites a peça fica com o nome desse quadrado.

Daqui se retira a necessidade de se ter de localizar o tabuleiro no mundo real e a importância de o fazer com grande precisão.

[1] https://pt.wikipedia.org/wiki/Nota%C3%A7%C3%A3o_alg%C3%A9brica_de_xadrez

	a	b	c	d	e	f	g	h	
8	a8	b8	c8	d8	e8	f8	g8	h8	8
7	a7	b7	c7	d7	e7	f7	g7	h7	7
6	a6	b6	c6	d6	e6	f6	g6	h6	6
5	a5	b5	c5	d5	e5	f5	g5	h5	5
4	a4	b4	c4	d4	e4	f4	g4	h4	4
3	a3	b3	c3	d3	e3	f3	g3	h3	3
2	a2	b2	c2	d2	e2	f2	g2	h2	2
1	a1	b1	c1	d1	e1	f1	g1	h1	1
	a	b	c	d	e	f	g	h	

Figura 4.5: Notação algébrica de xadrez permite identificar cada quadrado do tabuleiro unicamente. Imagem retirada da Wikipédia, A enciclopédia livre[1].

4.3.1 Transformação entre a *frame* do tabuleiro e a mesa

O nó */board_info_rgb* trata da deteção do tabuleiro, subscreve os tópicos: */coefs*, */camera/rgb/image_raw* e */camera/rgb/camera_info*. O primeiro são os coeficientes do plano da mesa, o segundo corresponde à captação de vídeo da câmara e finalmente o último tópico contém os parâmetros intrínsecos da câmara.

O nó começa por receber uma imagem parecida com a figura 4.6 onde se pode ver o tabuleiro com algumas peças, o tampo da mesa e o chão.

Figura 4.6: Imagem inicial recebida da câmara, pode-se ver o tabuleiro, parte do tampo da mesa e o chão, assim como as peças que estão por cima do tabuleiro.

Todo o processamento de imagem será auxiliado na biblioteca de visão por computador

OpenCV[7].

Deteção dos cantos presentes no tabuleiro

A primeira informação a ser procurada é a posição do centro do tabuleiro na imagem e posteriormente convertida para coordenadas do mundo real. Para tal usa-se o método *goodFeaturesToTrack()*[2], com o protótipo descrito na listagem 4.1, para se obterem as coordenadas dos cantos entre os quadrados do tabuleiro.

Listagem 4.1: Protótipo da função *cv2.goodFeaturesToTrack()*

```
corners = cv2.goodFeaturesToTrack(image, maxCorners,
qualityLevel, minDistance, blockSize, useHarris, k)
```

A esta função foram fornecidos os seguintes parâmetros por ordem de aparecimento no protótipo 4.1: a imagem a ser processada, 150 como limite máximo de cantos que a função pode retornar, 0.05 para nível de qualidade, 10 para a distância mínima euclidiana entre cantos, 2 para o tamanho do bloco, *True* para ativar o uso do algoritmo *Harris detector* e finalmente o parâmetro livre *k* do *Harris detector* com valor de 0.04.

Um tabuleiro de xadrez tem 79 cantos(2 cantos não são possíveis de serem detetados se o tabuleiro tiver uma borda preta), idealmente com um limite de cantos igual a 79 só seriam retornados esses, no entanto, pode acontecer existirem cantos fora do tabuleiro com melhor qualidade que cantos dentro do tabuleiro e estes substituiriam os cantos pertencentes ao tabuleiro, não sendo no fim retornados pela função. Assim, define-se o limite como bastante superior para garantir que o maior número de cantos do tabuleiro são retornados e no fim faz-se uma filtragem para remover os pontos que não estão no tabuleiro.

Visto que a redução do limite não garantiria o não retorno de cantos fora do tabuleiro a filtragem teria de ser feita sempre, é preferível assim trabalhar sobre um maior número de cantos do que menor.

Na figura 4.7 pode-se ver que mesmo com um limite máximo de 150, apenas 7 cantos foram retornados fora do tabuleiro e ainda assim não foi retornado um canto dentro do tabuleiro, isto deve-se ao parâmetro nível de qualidade que foi escolhido como sendo 0.05.

Dentro da função cada canto possui um valor de qualidade, o nível de qualidade garante que todos os cantos que tenham esse valor inferior a 5% do canto com melhor qualidade não sejam retornados, assim o número de cantos fora do tabuleiro é reduzido porque os cantos do tabuleiro terão, na sua generalidade, sempre melhor qualidade que os cantos fora visto possuírem um maior contraste, no entanto isto não acontece para todos.

A distância mínima euclidiana foi definida como sendo inferior à largura de um quadrado de um tabuleiro com tamanho relativamente reduzido para que cantos adjacentes pudessem ser retornados mesmo que se varie o tamanho do tabuleiro.

Tanto o tamanho do bloco como a parâmetro do algoritmo *Harris detector* foram deixados com os seus valores padrão e por experimentação foram obtidos resultados ótimos.

[2]Documentação da função *goodFeaturesToTrack()*: `https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga1d6bb77486c8f92d79c8793ad995d541`

Figura 4.7: Cada círculo verde corresponde a um cantos retornado pela função *goodFeaturesToTrack()* sobreposto na imagem inicial.

Filtragem dos cantos fora do tabuleiro

Para se removerem os cantos fora do tabuleiro começa-se por determinar o canto médio resultante dos vários cantos presentes na figura 4.7, desta forma os cantos do tabuleiro devido à sua maior densidade têm uma maior contribuição para este canto médio que os cantos fora.

De seguida, são calculadas as distâncias entre todos os cantos e o canto médio, na figura 4.8 podem-se ver as distâncias representadas por linhas vermelhas e como os cantos fora do tabuleiro têm distâncias bastante maiores do que os cantos do tabuleiro estes podem ser removidos através de uma análise estatística, para tal obtêm-se a média e o desvio padrão das distâncias.

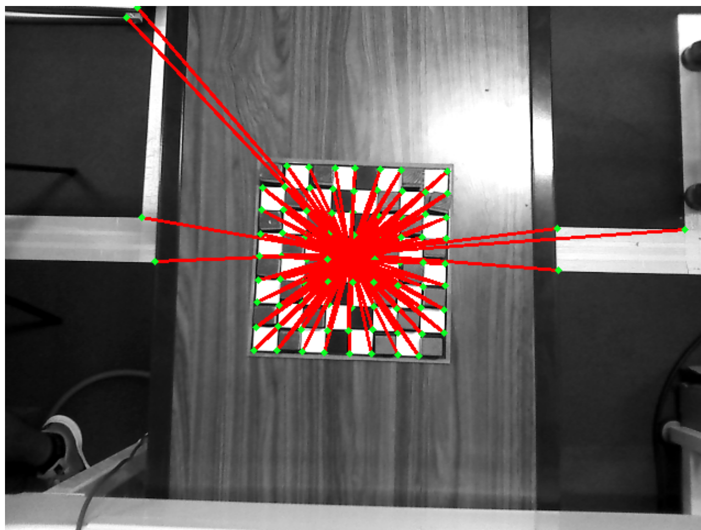


Figura 4.8: As linhas vermelhas representam a distância de um determinado canto ao canto médio.

A partir da média(μ) e desvio padrão(σ) calcula-se o coeficiente de variação(c_v) que é

dado pela equação 4.2.

$$c_v = \frac{\sigma}{\mu} \tag{4.2}$$

É usado o coeficiente de variação por ser uma medida adimensional o que permite que qualquer que seja o tamanho do tabuleiro os valores do c_v mantêm-se. Isto porque, um qualquer conjunto de cantos que gere um c_v superior a 0.4 significa que possui cantos que não pertencem ao tabuleiro, tendo sido este valor limite de 0.4 calculado para uma deteção ideal, ou seja, uma deteção com todos os cantos do tabuleiro e nenhum fora.

Este cálculo foi feito para tabuleiros de vários tamanhos, em ambiente de simulação, e rondava o valor 0.36, o limite foi definido ligeiramente acima para que houvesse alguma histerese e não fossem filtrados pontos imediatamente após uma variação pequena no c_v.

Assim, para c_v superiores a 0.4 filtra-se com limites mais apertados e para c_v inferiores os limites são alargados de modo a acomodarem um maior número de distâncias.

Estes limites são definidos da seguinte forma: para $c_v < 0.4$ só ficam os cantos cuja distância ao canto médio esteja dentro do intervalo $[\mu - 1.5\sigma, \mu + 1.5\sigma]$ e para $c_v >= 0.4$ o intervalo é $[\mu - 2.5\sigma, \mu + 2.5\sigma]$. Esta filtragem é aplicada um máximo de 5 vezes ou até o c_v ser inferior a 0.4, o resultado obtido corresponde à figura 4.9.

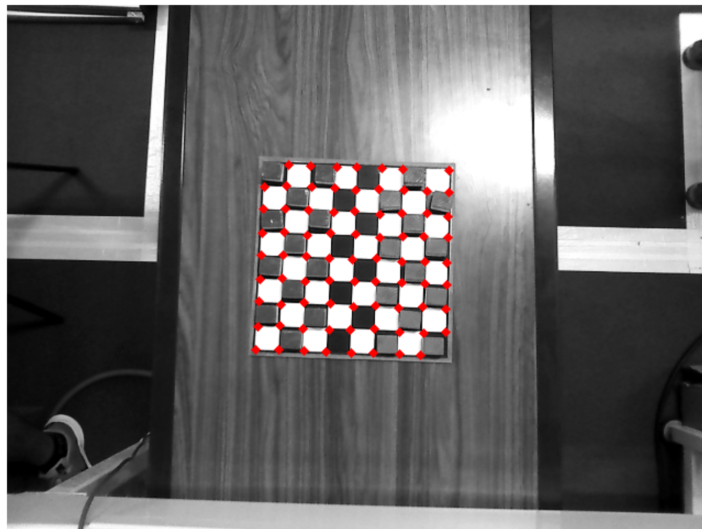


Figura 4.9: Resultado após remoção dos cantos fora do tabuleiro, só ficaram os pontos a vermelho.

Cálculo do centroide do tabuleiro

Os pontos da figura 4.9 são conectados todos entre si por uma linha resultando na figura 4.10.

A esta figura é aplicada a função *findContours()*[3] da biblioteca OpenCV com o modo *cv.RETR_EXTERNAL* para retornar apenas o contorno externo e com o algoritmo de

[3]Documentação da função *findContours()*: `https://docs.opencv.org/4.5.2/d3/dc0/group__imgproc__shape.html#gadf1ad6a0b82947fa1fe3c3d497f260e0`

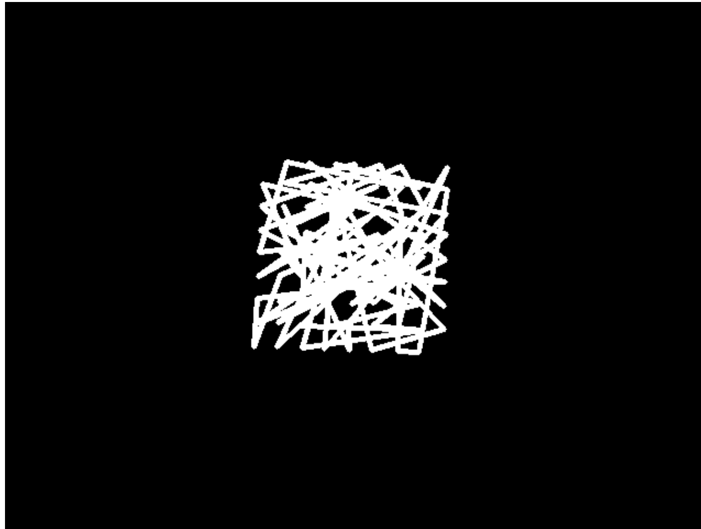


Figura 4.10: Resultado de ligar todos os pontos por uma linha entre si.

aproximação *cv.CHAIN_APPROX_NONE* para que não seja feita nenhuma aproximação aos pontos do contorno e este seja devolvido na integra.

O contorno devolvido é a linha a azul na figura 4.11

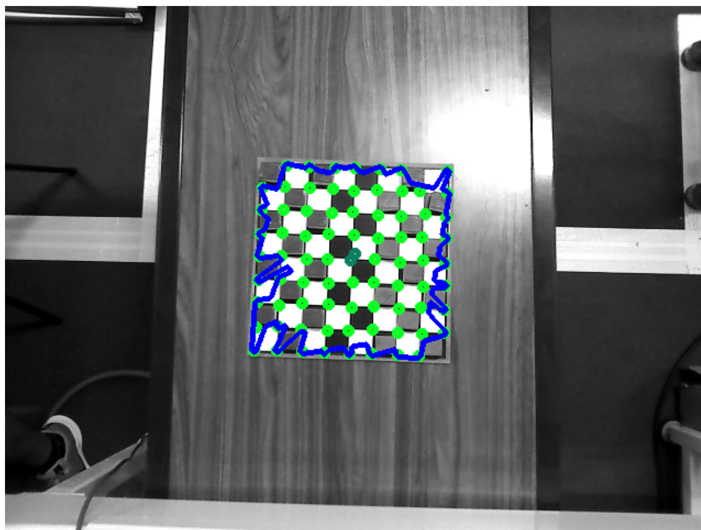


Figura 4.11: Linha azul corresponde ao contorno retornado pela função *findContours*.

Este contorno é fornecido à função *convexHull()*[4] para que sejam removidas as concavidades e suavizado assim o contorno, o resultado pode ser observado na figura 4.12, as linhas desta figura são o mais finas possível para que se tenha uma maior precisão na deteção dos cantos do tabuleiro uma vez que os pixeis dos cantos serão convertidos para coordenadas reais.

É importante agora extrair as 4 linhas que compõem as arestas do tabuleiro para que se encontrem as interseções entre elas, ou seja, os vértices do tabuleiro e com estes seja inferido o centro do tabuleiro.

[4]Documentação da função *convexHull()*: `https://docs.opencv.org/4.5.2/d3/dc0/group__imgproc__shape.html#ga014b28e56cb8854c0de4a211cb2be656`

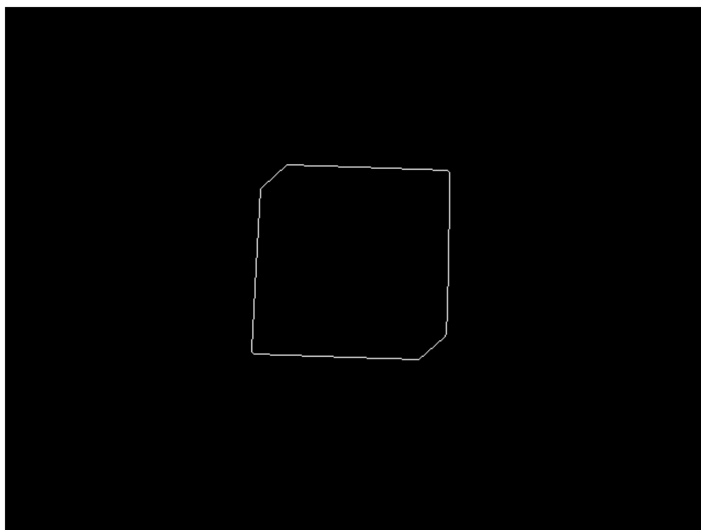


Figura 4.12: Resultado da função *convexHull()* quando aplicada ao contorno da figura 4.11.

Os pontos que fazem parte do contorno são assim fornecidos ao método da listagem 4.2 através parâmetro *data*. Este método implementa um algoritmo RANSAC da biblioteca **Scikit-image**[8] que procura pontos que encaixem no modelo fornecido. Neste caso o objetivo é procurar linhas pelo que *model_class = skimage.measure.LineModelND*.

Listagem 4.2: Protótipo da função *skimage.measure.ransac()*

```
model, inliers = skimage.measure.ransac(data, model_
_class, min_samples, residual_threshold, max_trials)
```

O número mínimo de amostras(*min_samples*) que o método deve usar para criar uma linha são 3. Uma vez que uma linha pode ser feita a partir de 2 pontos apenas, isto resultaria em linhas com um ponto pertencente a uma aresta e outro ponto pertencente a outra aresta que no fim não iriam resultar em modelos com muitos pontos agregados a essa linha, os chamados *inliers*.

Desta forma não se usa o número mínimo de amostras com o valor 2 para eliminar logo à partida a criação destas linhas cujos modelos não agrupam mais pontos das arestas uma vez que apenas as intersetariam em dois pontos.

Um número muito alto para este valor pode resultar no método não retornar linhas nenhumas visto que se torna mais difícil encaixar uma linha que passe por esses pontos todos.

O limite residual(*residual_threshold*) foi definido com o valor 2 e corresponde à menor distância que um ponto pode ter para ser considerado *inlier*. O número máximo de iterações(*max_trials*) foi limitado a 1000, todos os parâmetros restantes são opcionais e não foram usados pelo que não aparecem no protótipo da função acima mencionada.

Finalmente, só são aceites linhas com um número de *inliers* superior a 50, desta forma são excluídas imediatamente as linhas que seriam formadas no canto superior esquerdo da figura 4.12 e no canto inferior direito, as linhas resultantes dão origem à figura 4.13.

Pode, no entanto, acontecer que estas linhas indesejáveis tenham mais do que 50 *inliers* e sejam assim incluídas no resultado final, a figura 4.14 é exemplo disso, na imagem a) pode-se

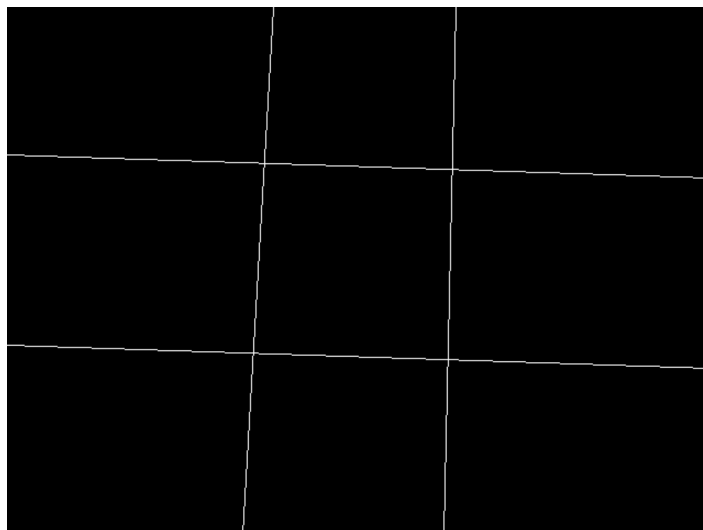
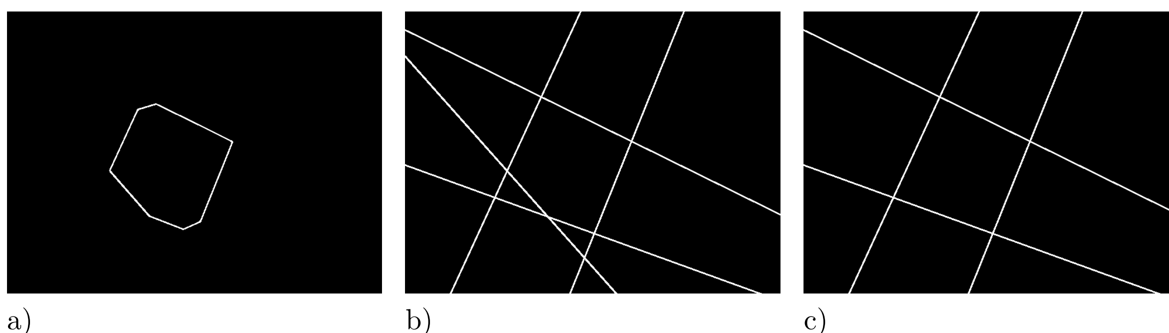


Figura 4.13: Resultado da função *skimage.measure.ransac* quando aplicada aos pontos que originam o contorno a branco da figura 4.12.

ver o resultado da função *convexHull* que originou um canto comprido o suficiente para que a linha contivesse mais do 50 pontos, como se pode ver na imagem b) isto resultou em 5 linhas.



a) b) c)

Figura 4.14: a) resultado da função *convexHull* para um número mais reduzido de pontos. b) resultado da função RANSAC aplicada à figura anterior. c) resultado após remoção das linhas que não formam um ângulo de, aproximadamente, 90º.

Para resolver este problema removem-se todas as linhas que não formam pelo menos um ângulo de, aproximadamente, 90º com outra linha.

No entanto, uma vez que a câmara pode não estar exatamente perpendicular ao tabuleiro de jogo, a imagem tem alguma distorção causada pelo ângulo da câmara com a mesa, quanto maior este ângulo, θ na figura 4.15, maior a distorção.

Desta forma as arestas do tabuleiro que estão a 90º, aparecem com ângulos inferiores a 90º quando próximos da câmara(ângulos azuis da figura 4.16) e ângulos superiores a 90º quando distantes da câmara(ângulos vermelhos da figura 4.16).

Uma possível forma de corrigir a perspetiva da imagem seria através do uso da função *getPerspectiveTransform()*[5] para obter a matriz de transformação de perspetiva e da função

[5] Documentação da função *getPerspectiveTransform*: `https://docs.opencv.org/4.5.2/da/d54/group__imgproc__transform.html#ga20f62aa3235d869c9956436c870893ae`

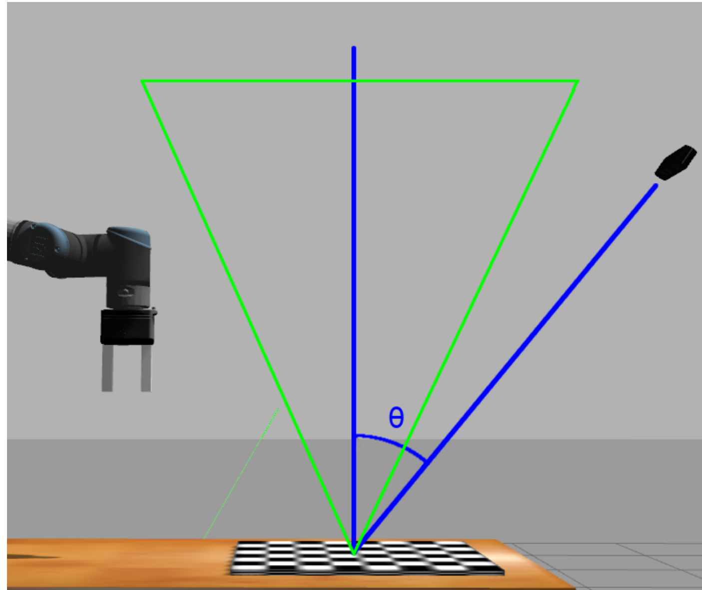


Figura 4.15: Quanto maior o ângulo θ maior a distorção causada pela perspetiva. A zona limitada pelo triângulo a verde representa a zona correta para o posicionamento da câmara.

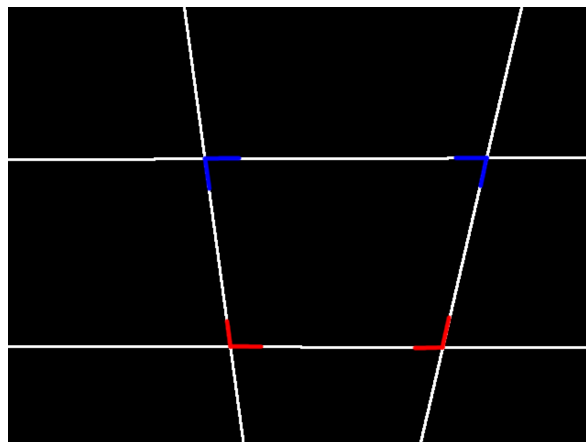


Figura 4.16: Ângulos azuis estão mais perto da câmara pelo que parecem ser inferiores a 90º e a vermelho mais distantes pelo que parecem ser superiores a 90º.

warpPerspective()[6] para aplicar essa matriz à imagem, ambas da biblioteca OpenCV. Estas funções transformam a imagem de tal modo que as linhas ficam perpendiculares entre si, ver figura 4.17.

No entanto para se obter a matriz de transformação são precisos os 4 pontos dos vértices do tabuleiro, informação que ainda não é possível obter uma vez que existem mais do que 4 linhas e a interseção entre elas não garante que sejam os vértices e sendo o objetivo filtrar as linhas que não pertencem às arestas do tabuleiro não existe ainda forma de obter estes vértices.

Para além disso fazer uma transformação deste género à imagem para encontrar o centroide implicaria fazer a transformação inversa de modo a que esse centroide ficasse posicionado na

[6] Documentação da função *warpPerspective*: `https://docs.opencv.org/4.5.2/da/d54/group__imgproc__transform.html#gaf73673a7e8e18ec6963e3774e6a94b87`

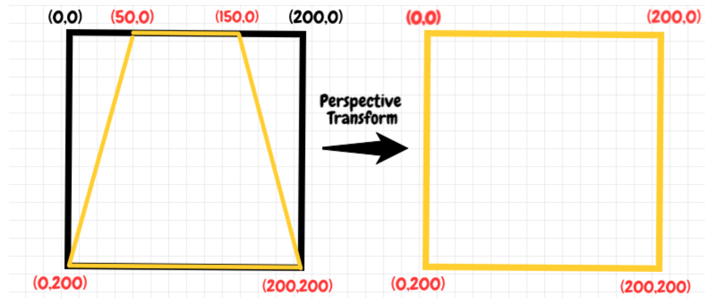


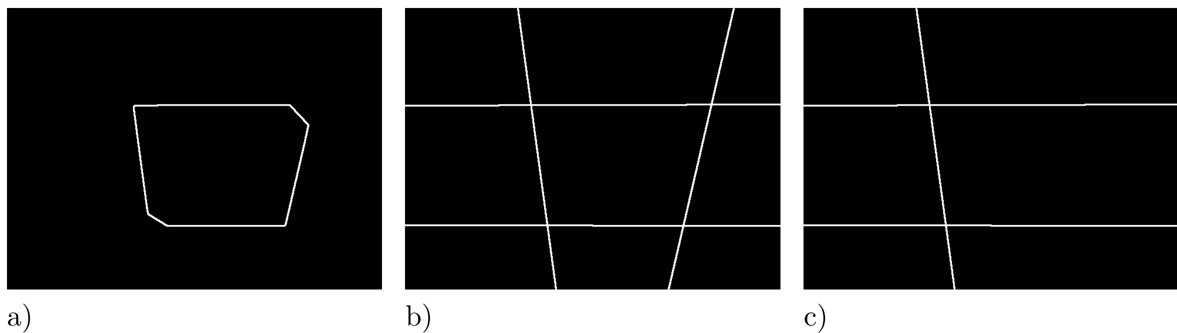
Figura 4.17: Exemplo de uma transformação de perspetiva, imagem de Shaikh[9].

imagem original para que com o uso da classe *image_geometry::Pinhole-CameraModel*[10] fosse possível mapear o pixel desse centroide em coordenadas do *frame* da câmara e assim obter a posição do tabuleiro, ora estas transformações podem gerar erros que afetam a determinação da posição do tabuleiro.

A solução implementada consiste em permitir uma maior liberdade para o ângulo entre as linhas, neste caso o intervalo [1.4,1.74] radianos ou [80.21,99.69] graus foi usado. Esta implementação tem a desvantagem de limitar o posicionamento da câmara ao triângulo verde da figura 4.15 cuja melhor posição é diretamente acima do tabuleiro, mesmo assim existe liberdade suficiente para que o posicionamento não seja rígido ou demasiado limitativo.

No caso da imagem c) da figura 4.14, onde a câmara foi montada com uma inclinação dentro do triângulo verde, pode-se ver que a filtragem ocorreu sem problemas e a linha correta foi eliminada.

Já no caso da imagem c) da figura 4.18 uma linha que não devia ser removida foi eliminada, isto deveu-se ao facto da câmara estar fora do triângulo verde da figura 4.15 gerando assim ângulos fora do intervalo definido anteriormente.



a) b) c)

Figura 4.18: a) resultado da função *convexHull* para um ângulo θ maior. b) resultado da função RANSAC aplicada à figura anterior. c) resultado após remoção das linhas por filtragem de ângulo que fazem entre si, uma linha foi erradamente removida.

Após se obter uma imagem igual à figura 4.13, onde se encontram apenas as 4 linhas, aplica-se de novo a função da listagem 4.1 agora com um número máximo de 4 cantos e um nível de qualidade de 0.01, mais restritivo que o anterior uma vez que a qualidade destes cantos será sempre superior por terem sido criados a partir da interseção de linhas desenhadas por software e que têm uma intensidade máxima em contraste com um fundo o mais escuro

possível.

O resultado são os 4 pontos verdes da figura 4.19.

Para cada um dos 4 pontos é determinado um raio com origem na câmara e que passa por cada um desses pontos através do método *projectPixelTo3dRay()* da classe *image_geometry::Pinhole-CameraModel*, é depois determinada a interseção deste raio com o plano da mesa, cujos coeficientes são já conhecidos, estes pontos da interseção estão ainda em coordenadas da *frame* da câmara pelo que são posteriormente transformados para a *frame* da mesa.

Com os 4 pontos na *frame* da mesa as distâncias entre eles são calculadas e são escolhidos dois pontos cuja distância entre eles seja a maior possível, garantido-se assim ser uma diagonal do tabuleiro, com estes dois pontos obtém-se o ponto médio entre eles que corresponde ao centroide do tabuleiro e ao círculo vermelho da figura 4.19.

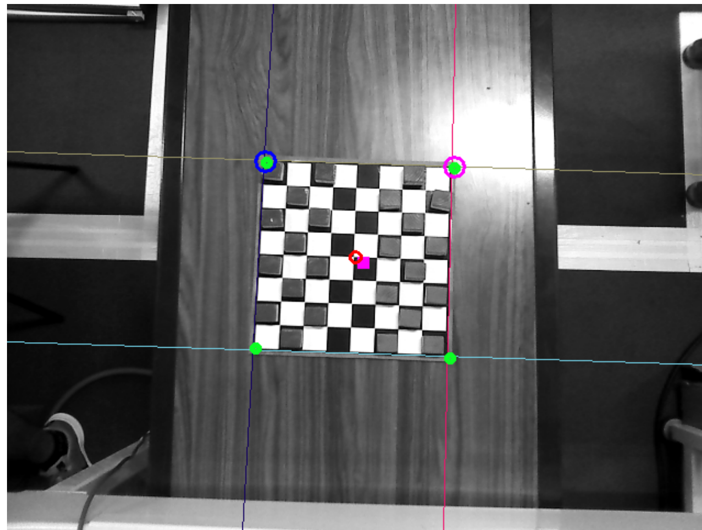


Figura 4.19: Resultado da procura do centroide e do ângulo do tabuleiro. Os pontos a verde são os vértices do tabuleiro, o círculo vermelho no centro do tabuleiro é o centroide determinado, o quadrado rosa é a ROI para definir o sentido positivo dos eixos e os círculos azul e roxo foram usados para determinar o ângulo com o eixo z da *frame* da mesa.

Com o centro do tabuleiro determinado falta agora descobrir o ângulo do tabuleiro em relação à *frame* da mesa para compensar a rotação que este pode ter na mesa. Este ângulo é então medido entre uma aresta do tabuleiro e o eixo y do referencial da mesa, ver figura 4.20. Para tal é necessário procurar primeiro o ponto mais próximo deste eixo, que corresponde a ter o valor da componente x menor em módulo. De seguida, são formados vetores entre este ponto e os restantes, e calculados os ângulos destes com o eixo y, os pontos que formem o menor ângulo são escolhidos, assim este ângulo está sempre no intervalo [0,90[graus.

Os pontos usados para este cálculo são representados por um círculo azul e um círculo roxo, na figura 4.19, sendo o primeiro o ponto mais próximo do eixo e o segundo o ponto usado para criar o vetor que originou o menor ângulo com o eixo y.

Com o centro e o ângulo já é possível criar um novo referencial no centro do tabuleiro cujos eixos x e y são paralelos às arestas deste. No entanto, uma vez que há necessidade de

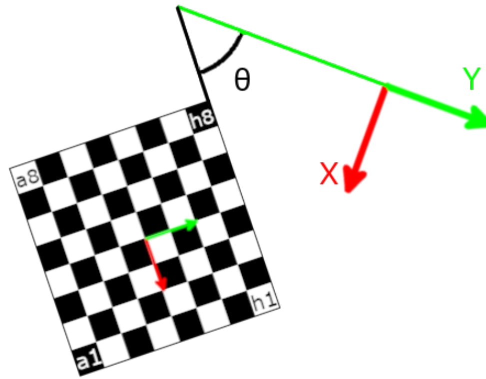


Figura 4.20: Os eixos x e y fora do tabuleiro correspondem ao referencial da mesa visto de cima(sem componente z). Partindo do vértice mais próximo do eixo y o vértice adjacente a este é escolhido de modo a que origine um vetor com o menor ângulo possível com o eixo y, este é o ângulo θ. Os eixos dentro do tabuleiro correspondem ao referencial do tabuleiro, o eixo x é o vermelho e o eixo y é o verde.

identificar cada quadrado do tabuleiro individualmente segundo a notação algébrica de xadrez e esta tem certas condições para o fazer, por exemplo o quadrado **a1** ser preto no canto inferior esquerdo do lado das peças brancas, é preciso garantir assim, uma certa ordem no sentido dos eixos deste referencial, por forma a facilitar a posterior identificação dos quadrados.

Visto que o objetivo é criar uma forma de identificar os quadrados independentemente do jogo a ser jogado, não é possível basear esta identificação na cor das peças, pelo que se define o quadrado **a1** ser o quadrado preto mais à esquerda do tabuleiro no lado a partir do qual o jogador humano joga. Deste modo o tabuleiro internamente ao programa tem sempre um estado fixo e apenas é necessário fazer uma tradução para cada motor de jogo, seja damas, peões, xadrez, etc...

Define-se ainda que o eixo x deste referencial tem sentido positivo no sentido decrescente dos números e que o eixo y corresponde às letras e tem sentido positivo correspondente à ordem crescente alfabética das mesmas, ver figura 4.20.

Para verificar que os eixos estão de acordo com esta definição é criada uma máscara com forma quadrangular e menor que o tamanho de um quadrado do tabuleiro (ver quadrado roxo da figura 4.19) esta máscara é aplicada numa imagem a preto e branco, ao quadrado imediatamente abaixo e à direita do centro do tabuleiro, na figura 4.20 corresponde ao quadrado branco entre os eixos x e y dentro do tabuleiro, é feita a média do valor dos pixeis dentro da máscara e se este valor for superior a 128 significa que o quadrado em questão é branco e os eixos estão corretos, esta é a situação da figura 4.20, caso seja inferior a 128 o quadrado é preto e é necessário rodar os mesmos 90º para que entre eles fique sempre um quadrado branco.

No entanto, no caso da câmara estar invertida ou se a câmara não estando invertida estiver

posicionada do lado do robô, o que acontece é que a face que contêm o quadrado **a1** fica no lado do tabuleiro mais próximo do robô e assim a definição de que o humano fica sempre do lado do quadrado **a1** torna-se inconsistente com a realidade. Para restabelecer o sentido dos eixos é necessário, nesta situação, rodá-los 180º por forma a que o eixo x aponte para o lado do jogador humano.

É criado então um vetor, roxo na figura 4.21, entre a base do robô e o centro do tabuleiro e se este fizer um ângulo com o eixo x do tabuleiro superior a 90º significa que os eixos do tabuleiro necessitam de correção, como acontece na situação do lado direito da figura 4.21, no caso do ângulo ser inferior a 90º como na situação do lado esquerdo da figura 4.21 não há necessidade de corrigir, os eixos já se encontram corretos. De referir ainda que se ângulo for exatamente 90º significa que o tabuleiro não se encontra com as faces, onde as peças são posicionadas, viradas para cada um dos jogadores pelo que qualquer uma delas pode ser a do robô ou do humano. Este não é um problema pois nunca se joga jogos de tabuleiro a partir de outra face que não aquela mais próxima do jogador.

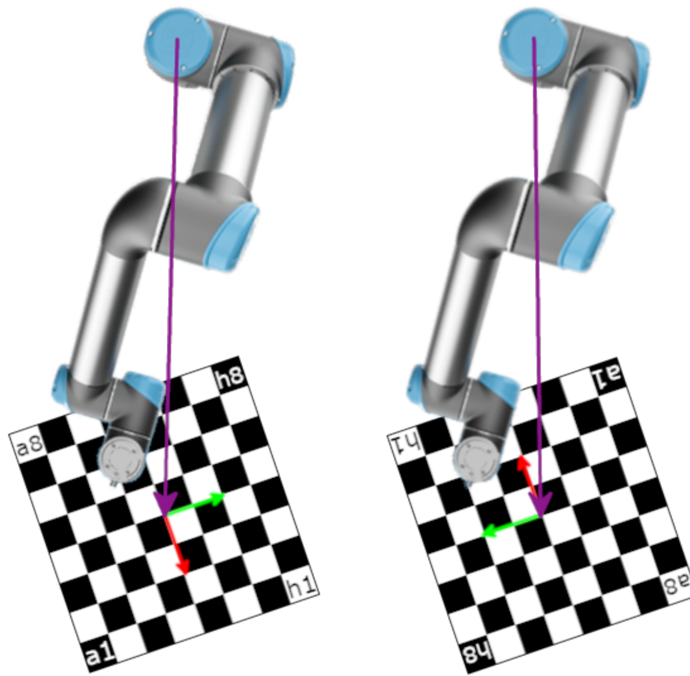


Figura 4.21: Na situação do lado esquerdo o vetor roxo faz com o vetor vermelho(eixo x do tabuleiro) um ângulo inferior a 90º pelo que não é preciso corrigir a orientação dos eixos, já no caso do lado direito este ângulo é superior a 90º e há necessidade de rodar os eixos 180º para que o eixo x aponte no sentido do jogador humano.

Finalmente, é criada uma transformação estática entre a *frame* da mesa e a *frame* do tabuleiro e publicada no tópico */board_transform*.

A distância entre dois vértices adjacentes é calculada e corresponde ao tamanho do

tabuleiro, é publicada no tópico */board_size*.

4.3.2 Construção dos limites dos quadrados do tabuleiro

O nó que trata de criar os limites dos vários quadrados é o */board_coords* e subscreve o tópico */board_size*.

Com um referencial no centro tabuleiro e com a largura deste é possível criar limites para cada quadrado do tabuleiro, de modo a que numa fase posterior as peças possam ser identificadas com o nome do quadrado com base na posição das mesmas no tabuleiro.

É criada então uma grelha de pontos com uma distância entre eles igual à largura de um quadrado, estes pontos correspondem aos cantos dos vários quadrados. Para um determinado quadrado é suficiente conhecer a informação do canto superior direito e do canto inferior esquerdo. Também seria suficiente saber as coordenadas de um ponto e com a largura do quadrado inferir os limites deste. Na figura 4.22 o quadrado **b3** é limitado pelo canto verde e pelo canto azul.

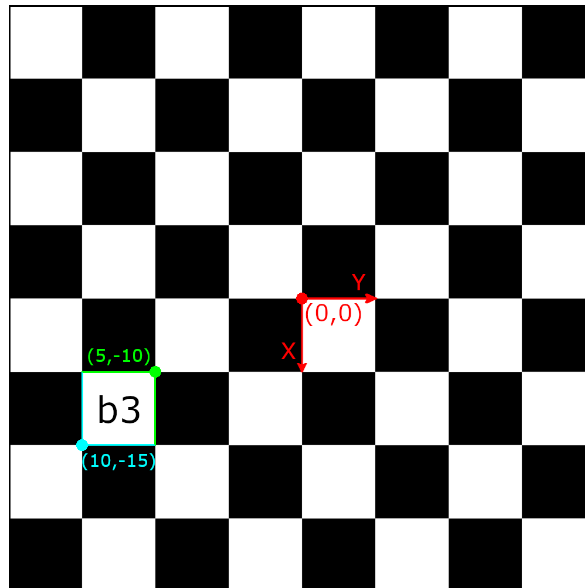


Figura 4.22: A origem do referencial do tabuleiro está a vermelho, considerando que cada quadrado tem uma largura de 5cm as coordenadas do ponto verde e do ponto azul do quadrado **b3** são (5,-10) e (10,-15), respetivamente.

Esta grelha tem por base tabuleiros de 8x8 quadrados e começa a identificação dos quadrados pelo quadrado **a1** que corresponde ao ponto com maior valor de x e o menor valor de y. De seguida percorrendo os pontos restantes no sentido do eixo y altera-se a letra do quadrado mantendo o número e percorrendo no sentido contrário ao eixo x altera-se o número do quadrado mantendo a letra. Ficam assim todos os quadrados identificados e limitadas as áreas que cada um deles ocupa no mundo real.

Falta apenas publicar esta informação para que outros nós a possam usar. Com este objetivo foi criada uma mensagem com a estrutura seguinte:

Listagem 4.3: Protótipo da mensagem *BoardSquare*.

```
string label
geometry_msgs/PointStamped pointTR
geometry_msgs/PointStamped pointBL
```

A *label* contém o nome do quadrado segundo a notação algébrica de xadrez, a variável *pointTR* contém as coordenadas do pontos do canto superior direito e a *pointBL* do canto inferior esquerdo. A seguinte mensagem serve para aglomerar a informação de todos os quadrados de um tabuleiro numa mensagem apenas:

Listagem 4.4: Protótipo da mensagem *BoardInfo*.

```
std_msgs/Header header
float32 board_width
BoardSquare[] squares
```

Esta última contém assim um *array* do tipo *BoardSquare* com a informação de todos os quadrados, para além da largura do tabuleiro e um *header* para sincronização no ROS.

Este nó publica a mensagem do tipo *BoardInfo* no tópico */board_coords.*

4.4 Estado do tabuleiro

O estado do tabuleiro corresponde a saber num determinado momento que peças estão no tabuleiro e onde.

Com a posição do tabuleiro encontrada, todos os seus quadrados identificados e a *point cloud* com apenas pontos pertencentes às peças, é possível agora identificar individualmente estas com base no quadrado do tabuleiro onde estão. Para isso é necessário fazer *clustering* dos vários aglomerados de pontos na *point cloud* para que cada aglomerado corresponda apenas a uma peça.

A partir destes *clusters* é possível inferir a posição no mundo real determinando o seu centroide e ângulo de manipulação para que o braço robótico possa agarrar a peça. No final é preciso organizar estes dados para que sejam utilizados por outros nós no sistema.

O nó que trata do estado do tabuleiro é o */board_state* e subscreve os tópicos */objects_table* que é a *point cloud* filtrada, ou seja, só com pontos das peças e o */board_coords* que é o tópico com a informação sobre cada quadrado do tabuleiro.

Para segmentar a *point cloud* da imagem b) da figura 4.4, nas várias peças é usada a classe *pcl::EuclideanClusterExtraction*[11] que com uma tolerância de $4mm$, um número mínimo de amostras de 100 e um número máximo de 25000 retorna um *std::vector<pcl::PointIndices>*.

Cada *pcl::PointIndices* contém um std::vector<int> com os índices dos pontos pertencentes a um determinado *cluster*.

Por cada *cluster* é criada uma nova *point cloud*, os pontos desta são adicionados a um objecto da classe *pcl::CentroidPoint* que com o método *get()* retorna o centroide dessa *point cloud* e que corresponde ao centroide da peça, este é considerado como sendo a localização da peça no mundo.

É agora possível identificar a peça com o nome do quadrado onde esta se encontra.

Suponhamos que temos uma peça num quadrado qualquer no tabuleiro, começa-se então por transformar o ponto do centro dessa peça da *frame /table_plane* para a *frame /board* e que após a transformação este ponto é dado por $centro = (c_x, c_y)$, com o ponto no referencial do tabuleiro itera-se sobre os vários quadrados deste, por cada um existe a informação de dois vértices não adjacentes, sejam estes os pontos: $verde = (v_x, v_y)$ e $azul = (a_x, a_y)$ a peça é identificada com o nome desse quadrado se as condições $v_x <= c_x <= a_x$ e $v_y >= c_y >= a_y$ se verificarem.

No caso da figura 4.22 se o ponto da peça tivesse os valores $(7.5, -12.5)$, após a transformação, esta era identificada como sendo a peça **b3**. Numa situação de jogo com várias peças o resultado corresponde à figura 4.23 quando em simulação e à figura 4.24 no mundo real.

Se uma peça não verificar as duas condições para nenhum dos quadrados é identificada como estando fora do tabuleiro.

Com a cor média dos vários pontos pertencentes a um determinado *cluster* e através do espaço de cor HSV, pode-se identificar a cor de uma determinada peça, se esta cor tiver as suas componentes dentro dos intervalos definidos na tabela 4.1 a peça é identificada como azul ou vermelha.

Azul	*mínimo*	*máximo*
hue	90	140
saturation	150	255
value	0	255

Vermelho	*mínimo*	*máximo*
hue	0	10
hue	170	180
saturation	70	255
value	50	255

Tabela 4.1: Intervalos das componentes HSV na biblioteca OpenCV para a identificação da cor azul e da cor vermelha.

As peças cuja componente z, após transformação para a *frame /board*, for superior a $3cm$ são consideradas damas, pelo que todas as peças normais têm de ter altura inferior a $3cm$ e as damas têm de ter altura superior.

Finalmente, para que o robô pegue na peça de forma correta, ou seja, pegar na peça a partir de duas faces opostas, evitando ao máximo que o robô pegue na peça a partir dos cantos que podem não oferecer consistência suficiente para uma pega bem-sucedida, é necessário encontrar o ângulo que esta faz no referencial em que se insere. O objetivo é assim maximizar a área de contacto entre a peça e a garra, algo que pode também ser alcançado alterando o tipo de garras para que sejam, por exemplo, mais moldáveis ao objeto.

Uma vez que as peças usadas são prismas quadrangulares, é suficiente descobrir a rotação da face do topo da peça para que a garra possa compensar a rotação da peça toda. Esta é uma conclusão importante pois permite reduzir o problema a duas dimensões tornando-o possível de ser resolvido com ferramentas de tratamento de imagem. Desta forma, é preciso primeiro transformar o topo de cada uma das peças numa imagem.

Uma vez que o topo das peças é plano e já estão disponíveis *clusters* que separam cada peça numa *point cloud*, se a cada *cluster* for aplicado o algoritmo de RANSAC consegue-se

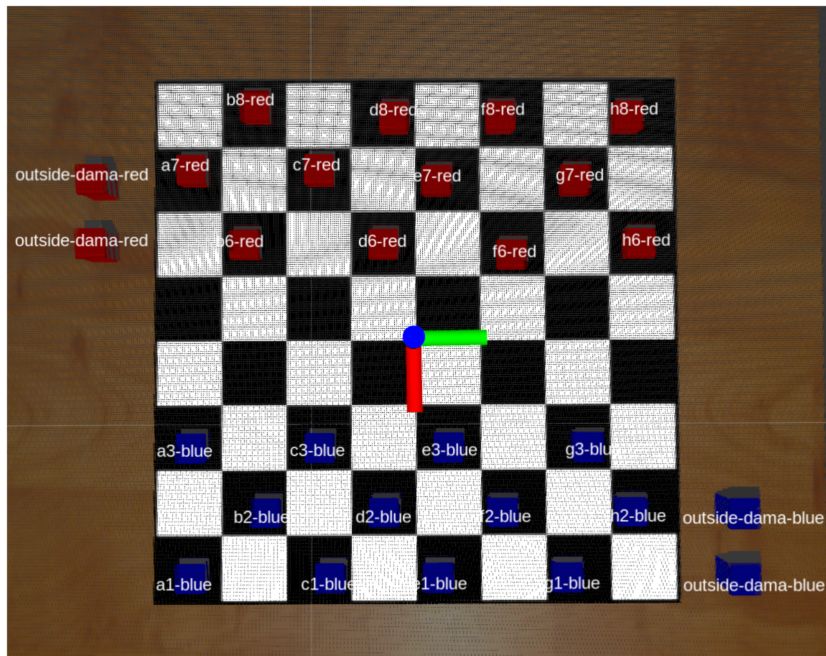


Figura 4.23: Resultado da identificação das peças com base na sua localização no tabuleiro em simulação. Os eixos do tabuleiro também estão presentes.

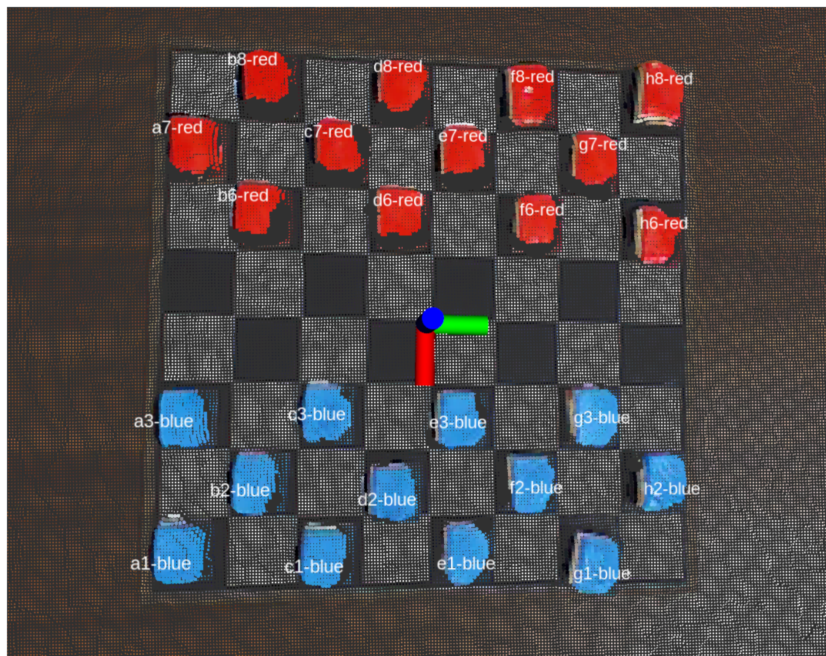
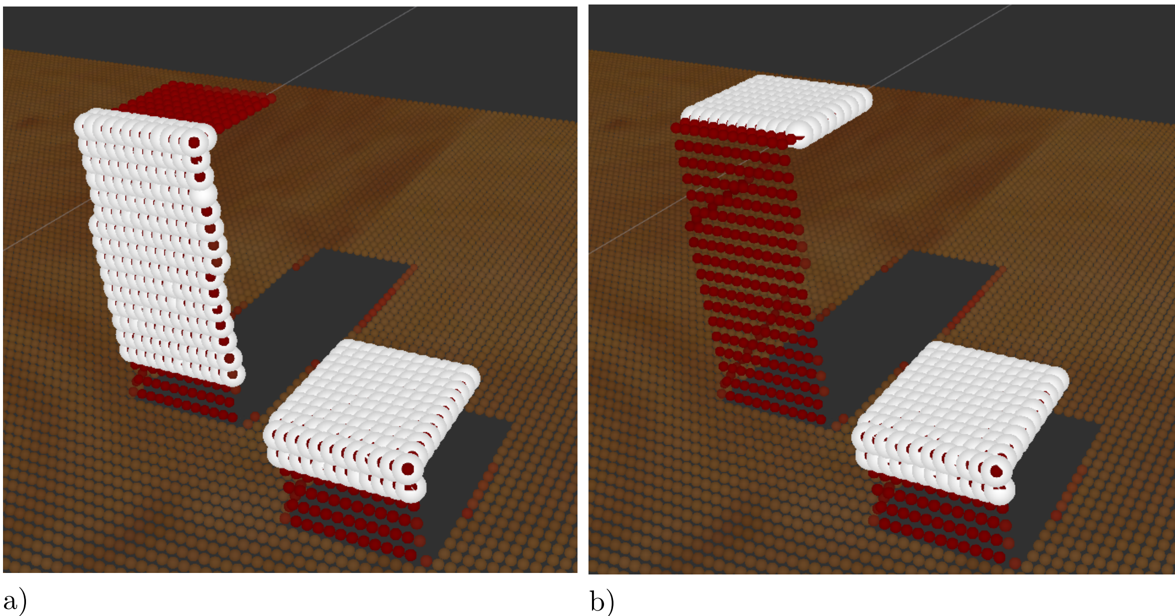


Figura 4.24: Resultado da identificação das peças com base na sua localização no tabuleiro em real. Os eixos do tabuleiro também estão presentes.

facilmente obter os pontos que pertencem ao plano do topo da peça, isto se este plano for o maior nesse *cluster*, no caso do maior plano detetado ser uma lateral, que pode acontecer, por exemplo, na dama, cuja altura faz com que a lateral contenha mais pontos do que o topo.

É assim importante garantir que o plano retornado corresponde ao topo e não a uma lateral. Como dito anteriormente, o algoritmo RANSAC retorna, para além dos pontos pertencentes ao plano encontrado, os coeficientes dele e com estes pode-se obter o vetor normal ao plano. Uma vez que este é, aproximadamente, paralelo ao tampo da mesa, os vetores normais ao plano da mesa e ao topo da peça são também eles paralelos, daqui resulta que o ângulo entre estes dois vetores ou é, aproximadamente, 0º se apontarem no mesmo sentido ou, aproximadamente, 180º se apontarem em sentidos opostos.

Visto que as laterais estão praticamente perpendiculares ao plano da mesa os intervalos para considerar que um plano corresponde a uma lateral ou topo podem ser bastante largos sem perda de funcionalidade. Desta forma se o ângulo estiver no intervalo $[0.5, \pi - 0.5]$ radianos o plano é considerado uma lateral como na imagem a) da figura 4.25, se estiver fora desse intervalo é considerado um topo como na imagem b) da mesma figura.



a) b)

Figura 4.25: Comparação entre uma má deteção e uma correta deteção dos topos das peças, os pontos a branco correspondem aos pontos detetados pelo método RANSAC. Na figura a) a peça dama por ser mais alta tem um plano lateral com mais pontos do que o plano do topo pelo que esse é o primeiro encontrado. Na figura b), com a aplicação da filtragem por ângulo entre vetores normais, o plano lateral da dama é descartado e o plano do topo é corretamente encontrado.

Com os pontos do topo da peça é agora possível construir uma imagem onde cada ponto da *point cloud* é mapeado num pixel preto, com esta imagem é possível descobrir o ângulo que a peça faz para que o robô a consiga agarrar.

O mapeamento dos pontos em pixeis foi feito usando as equações 4.3:

$$\begin{aligned} pixel_x &= cloud_point_x/scale + offset \\ pixel_y &= cloud_point_y/scale + offset \end{aligned} \tag{4.3}$$

A variável $cloud_point_x$ contém a informação em metros da componente x de um certo ponto, este é dividido pelo valor *scale*, que inicialmente foi determinado como sendo a diferença entre as componentes x de dois pontos consecutivos, isto garantiria que dois pontos consecutivos seriam representados por dois pixeis distintos e adjacentes.

No entanto, este valor podia calhar ser demasiado pequeno fazendo com que o topo da peça aparecesse muito pequeno na imagem, para colmatar este problema a escala foi fixada num valor determinado empiricamente de modo a aumentar um pouco o tamanho e salientar assim as arestas do quadrado para facilitar o cálculo do ângulo.

Aumentar o topo tem a desvantagem de espalhar os pixeis, ver figura 4.26, fazendo com que a função *findContours()* da biblioteca OpenCV tenha dificuldade em retornar o contorno correto, visto que esta retorna os pontos contínuos da borda da forma.

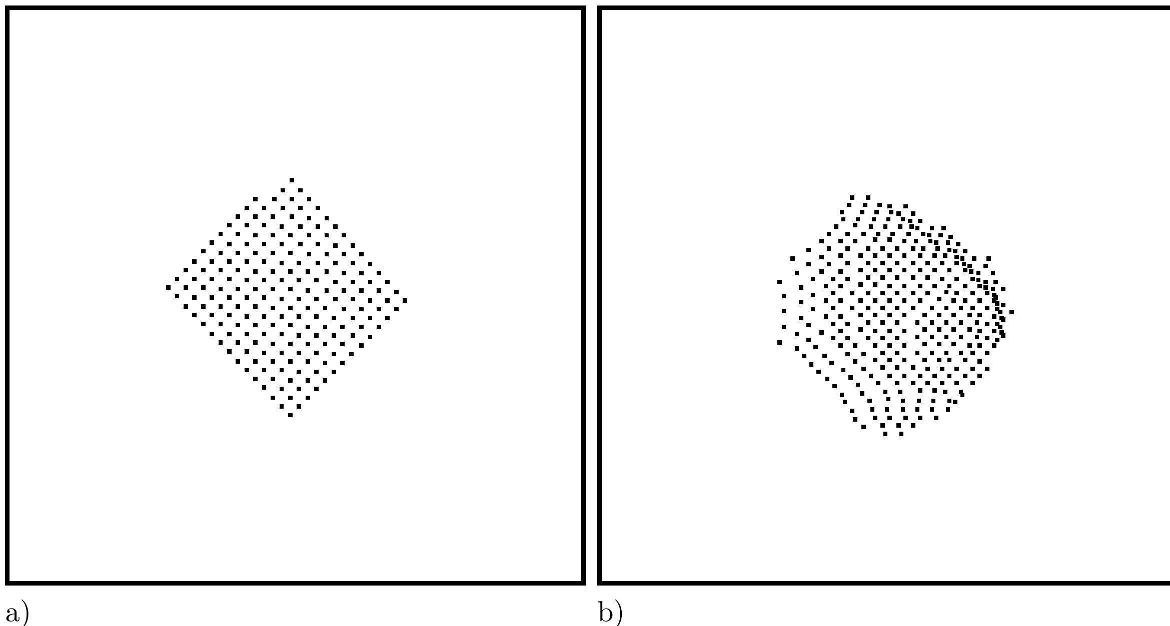


Figura 4.26: Resultado da projeção dos pontos da *point cloud* pertencentes ao topo de uma peça numa imagem 2D. a) projeção dos pontos obtidos em simulação. b) projeção dos pontos obtidos em ambiente real. As imagens foram alteradas para melhor visualização.

Para que seja possível obter um contorno correto a partir das imagens da figura 4.26 é preciso criar uma mancha mais uniforme, para isso aplica-se uma transformação morfológica de dilatação, função *dilate()*[7] da biblioteca OpenCV, com um elemento quadrangular de 3 por 3 pixeis, desta forma são acrescentados pixeis em todas as direções a partir da origem do elemento, resultado presente na figura 4.27.

[7]Documentação da função *dilate()*: `https://docs.opencv.org/3.4/d4/d86/group__imgproc__filter.html#ga4ff0f3318642c4f469d0e11f242f3b6c`

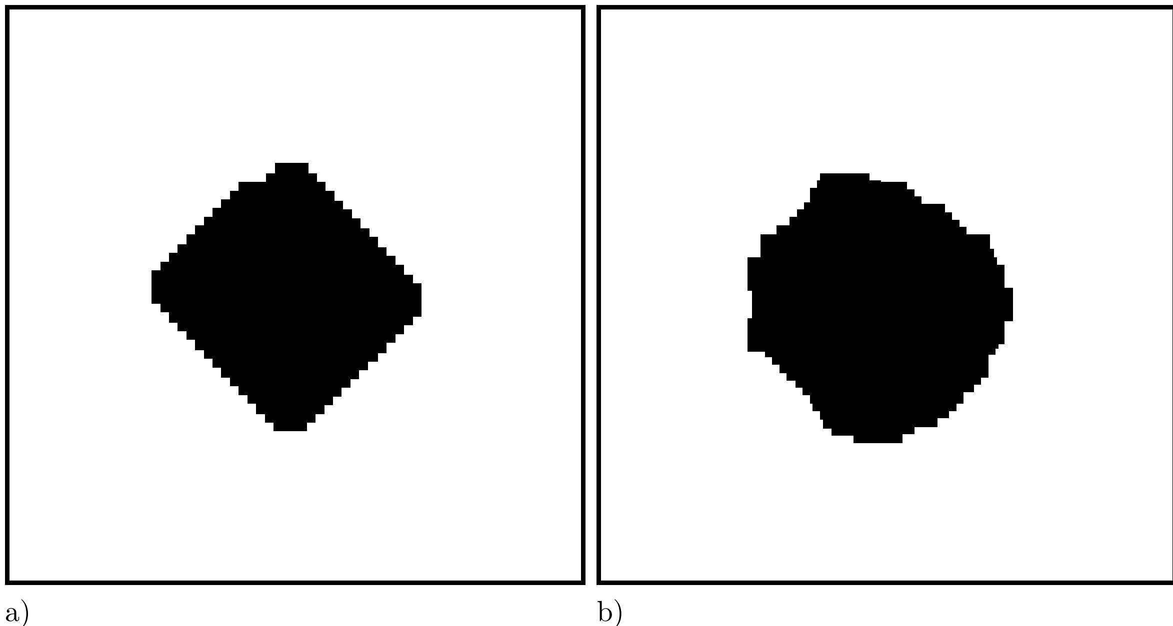


Figura 4.27: Resultado da aplicação da tranformação morfológica dilatação às imagens da figura 4.26. a) dilatação da projeção dos pontos obtidos em simulação. b) dilatação da projeção dos pontos obtidos em ambiente real. As imagens foram alteradas para melhor visualização.

Agora já se pode procurar com segurança o contorno do topo da peça com as imagens da figura 4.27, este contorno é fornecido ao método *minAreaRect()*[8] para que seja encontrado o retângulo com a menor área que envolva o contorno. A componente x da *point cloud* foi mapeada no eixo horizontal da imagem e a componente y no eixo vertical, o ângulo retornado pela função *minAreaRect()* é dado em relação ao eixo horizontal e no sentido dos ponteiros do relógio pelo que está no intervalo $[-90, 0[$ graus e corresponde a uma rotação no eixo z (yaw).

Com o valor de rotação de *yaw* rapidamente se obtém o quaternião da pose da peça que agregado ao centroide da mesma é informação suficiente para ser possível transformar a pose da peça para qualquer *frame* publicada. Esta propriedade é importante pois a peça na *frame* do tabuleiro permite determinar em que quadrado se localiza para lhe dar um nome, no entanto para que o robô consiga pegar é preciso saber a localização na *frame* deste.

Nas imagens da figura 4.28 podem-se ver a preto os retângulos retornados pela função *minAreaRect()* e a vermelho os contornos retornados pela função *findContours()*.

Na imagem a) uma vez que foi obtida em simulação o ruído presente na *point cloud* é praticamente nulo, o que permitiu obter a forma do topo da peça quase perfeita. Neste caso como o topo era quadrado o contorno é também um quadrado, consequentemente a função *minAreaRect()* consegue encontrar um retângulo que o envolva quase perfeitamente.

Já na imagem b) o contorno vermelho está mais desfigurado uma vez que o ruído é bastante perceptível fazendo com que uma peça com topo quadrado perca um pouco a consistência da sua forma e esta não seja obtida perfeitamente na imagem. A maior degradação está presente nos vértices do topo, no entanto, se as arestas preservarem a maior parte da sua forma, como

[8]Documentação da função *minAreaRect()*: `https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga3d476a3417130ae5154aea421ca7ead9`

é o caso desta imagem, o retângulo retorna na mesma o ângulo, aproximadamente, correto, para que as arestas preservem bem a sua forma ajuda ter peças não muito pequenas.

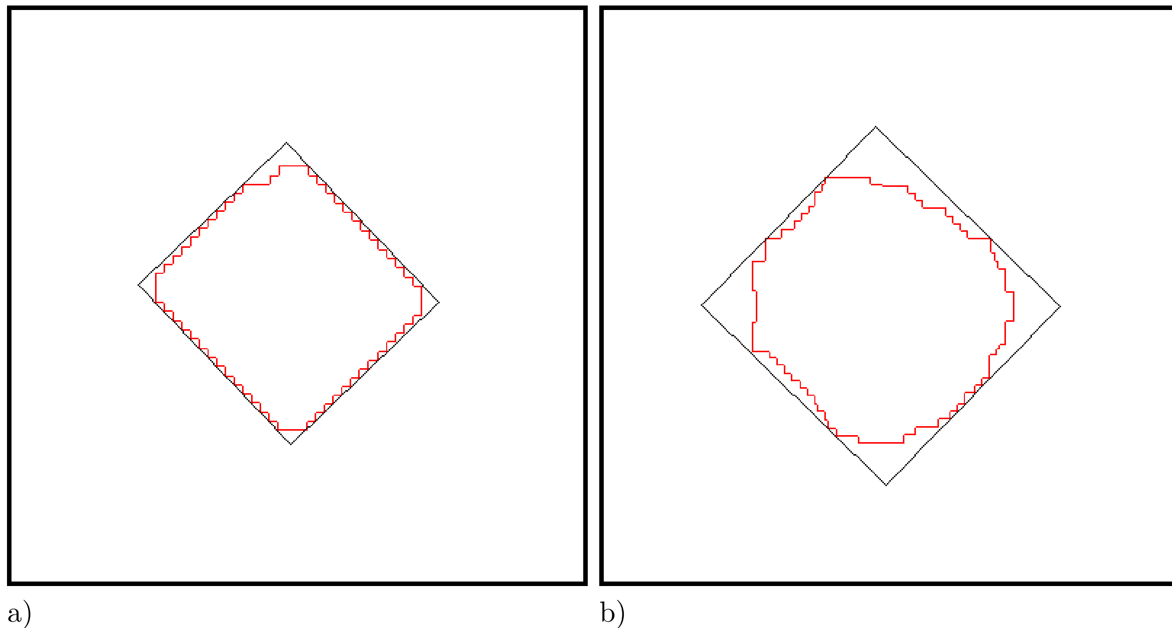


Figura 4.28: A vermelho o resultado da função *findContours()* quando aplicada à figura 4.27 e a preto o resultado da função *minAreaRect()* quando aplicada aos contornos a vermelho. a) contorno vermelho e retângulo preto estão sobrepostos devido à falta de ruído em simulação. b) contorno vermelho claramente com maior degradação devido à presença de ruído em ambiente real. As imagens foram alteradas para melhor visualização.

Finalmente, já se tem por cada peça, o nome do quadrado onde esta se localiza, a sua cor, a pose, ou seja, posição no mundo e ângulo, assim como a separação das peças normais das damas. No entanto, falta organizar esta informação para que os nós posteriores a possam usar.

A mensagem criada para agrupar estas variáveis está representada na listagem 4.5 onde *label* é o nome do quadrado do tabuleiro onde está a peça, *color* a cor da peça, *name* o nome da peça, no caso das damas as únicas opções são "none"para a peça normal ou "dama", mas outros jogos podem necessitar de outros nomes para as suas peças como por exemplo no jogo de xadrez, por fim *pose* corresponde à pose da peça.

Listagem 4.5: Protótipo da mensagem *PieceState*.

```
string label
string color
string name
geometry_msgs/Pose pose
```

Para que apenas uma mensagem seja publicada com a informação de todas as peças foi criada uma segunda mensagem, ver listagem 4.6, com um *array* do tipo da mensagem anterior, esta mensagem tem mais uma vantagem que é ter um *header* o que facilita na sincronização de mensagens pois contém a informação temporal de quando foi criada.

Listagem 4.6: Protótipo da mensagem *BoardState*.

```
std_msgs/Header header
PieceState[] boardstate
```

A mensagem *BoardState* é publicada no tópico */board_state*.

CAPÍTULO 5

Decisão

5.1 Introdução

A decisão consiste em determinar a jogada a ser efetuada pelo braço robótico, sem que para isso seja necessária intervenção humana, mas também receber e validar a jogada feita pelo humano, é por isso uma parte importante e integral deste sistema.

Este capítulo descreve a relação entre os motores de jogo (decisão) e a deteção do tabuleiro, das peças e do estado do tabuleiro(perceção), mais ainda, descreve a criação de um motor de jogo de raiz para o caso do jogo dos peões, uma vez que após alguma pesquisa não foi encontrado um motor para este jogo em concreto.

5.2 Regras dos jogos

Um jogo de tabuleiro é definido segundo um conjunto de regras. Uma vez que tanto na criação como na implementação de um motor de jogo as regras têm, obrigatoriamente, de ser respeitadas, serão de seguida apresentadas as regras dos dois jogos usados.

5.2.1 Damas Inglesas

O jogo das damas inglesas[12] é jogado num tabuleiro de 8x8 quadrados, cada jogador começa com 12 peças. O tabuleiro deve estar posicionado de forma a que o quadrado do canto à direita de cada jogador e mais perto de si seja branco. Os quadrados brancos do tabuleiro nunca são usados.

O objetivo do jogo é eliminar todas as peças do adversário, ou criar uma situação que deixe o adversário sem jogadas possíveis.

As regras são assim:

- As peças pretas começam sempre primeiro.
- À vez, cada jogador move uma das suas peças.
- As peças só se movem diagonalmente e na direção do adversário.
- As peças só podem avançar um quadrado de cada vez.

- Cada peça que chegue ao outro lado do tabuleiro é imediatamente transformada numa dama.
- As damas podem-se mover em qualquer direção um quadrado apenas.
- Só peças adversárias adjacentes à peça que captura podem ser capturadas.
- Capturar uma peça consiste em saltar por cima dela e poisar no quadrado imediatamente a seguir à peça capturada.
- Quando uma captura for possível, esta tem, obrigatoriamente, de ser feita.
- Não existe limite para o número de capturas consecutivas.
- Se existirem duas peças que possam capturar a mesma peça o jogador pode escolher qual usar.
- O jogador não é obrigado a escolher o caminho que capture mais peças adversárias.
- Uma peça capturada é removida do tabuleiro.
- Peças normais podem capturadas damas.

5.2.2 Jogo dos peões

O jogo dos peões[1]tem por base o jogo do xadrez, mas apenas se joga com peões, isto faz dele um bom exercício para treinar mecânicas de movimentação de peões. Cada jogador começa com 8 peões nas posições iniciais do xadrez, ou seja, na segunda linha a contar do lado de cada jogador, ver figura 5.1.

Os objetivos do jogo são: capturar todas as peças do adversário, deixar o adversário numa posição que não tenha mais jogadas possíveis ou chegar com um peão à linha mais próxima do adversário, ou seja, do outro lado do tabuleiro, a concretização de um destes objetivos é suficiente para um jogador ser vitorioso.

Se nenhum jogador tiver mais jogadas disponíveis o jogo empata.

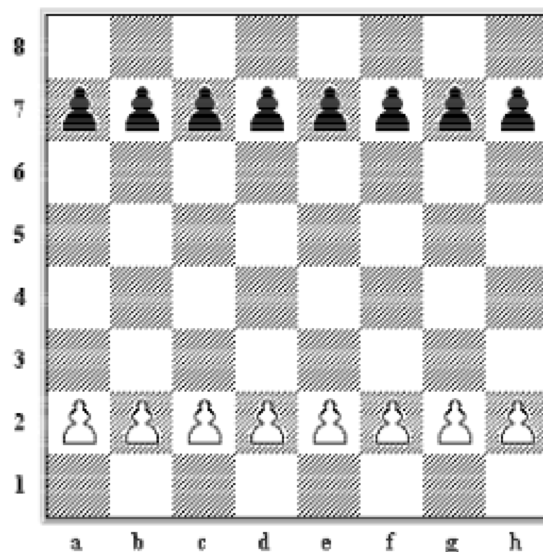


Figura 5.1: Posição inicial do jogo dos peões. Imagem retirada de Chess Corner[1].

[1]http://www.chesscorner.com/tutorial/basic/pawngame/pawngame.htm

As regras são assim:

- Os peões apenas se podem mover em frente (na direção do adversário).
- Na posição inicial os peões podem-se mover uma ou duas casas para a frente.
- Fora da posição inicial os peões apenas se podem mover um quadrado.
- Os peões apenas podem capturar em diagonal para a frente no quadrado adjacente àquele de onde partem.
- A captura de peões não é obrigatória.
- Um peão também pode capturar usando o modo de captura *en-passant*.

5.3 Motor do jogo das damas

A escolha de um motor de jogo é bastante importante e deve ser feita com algum cuidado e consideração para com a arquitetura geral, isto porque para além do motor dever ser capaz de decidir que jogada fazer deve também ser capaz de verificar as jogadas que o humano faz, avisando-o quando faz uma jogada ilegal. O motor deve também estar corretamente implementado segundo as regras do jogo e ser rápido na decisão de uma jogada forte.

Para além destas características a escolha do motor de jogo recaiu na necessidade de este ter o seu código fonte público, este é até um factor decisivo um vez que há necessidade de adaptar a comunicação entre o motor e o resto do sistema, por exemplo, ao enviar não só os quadrados de partida e chegada da peça a mover, mas também o quadrado da peça capturada, caso exista uma captura.

Esta mecânica não é à partida parte do motor, isto porque em software e na interface gráfica com o utilizador, uma peça capturada é simplesmente eliminada do estado do jogo, não existe necessidade de remover a peça capturada para fora do tabuleiro, pelo que o motor não retorna as peças capturadas quando uma jogada é feita por ele. Outra situação a ter em conta no caso das damas foi a necessidade de fazer com que o motor de jogo separe as jogadas compostas por vários saltos em jogadas simples para que o braço robótico as possa fazer.

Uma vez que é difícil encontrar um motor de jogo com estas características implementadas à partida daqui se justifica o acesso ao seu código, o código deve ser preferencialmente escrito em Python pois a curva de aprendizagem das mecânicas internas do motor é bastante baixa e a adaptabilidade do motor ao sistema é mais rápida.

O motor de jogo escolhido foi, assim, o **Samuel** [2] que é uma versão em Python do motor de jogo **GuiCheckers v1.05** [3], este motor tem por base uma função recursiva minimax com *alpha beta prunning* que diminui o número de jogadas que precisam de ser avaliadas para que chegue a uma jogada, aumentado a rapidez e o nível de profundidade que pode atingir na árvore de pesquisa de jogadas.

[2] `http://johncheetham.com/projects/samuel/`

[3] `https://www.3dkingdoms.com/checkers/oldCheckers.html`

5.4 Integração do motor de jogo das damas neste sistema

A integração de um motor de jogo consiste, essencialmente, em duas partes, na deteção da jogada do humano para ser inserido no motor de jogo e na determinação da jogada por parte do motor.

O nó */detect_human_move* trata de resolver a primeira parte. Para isso, subscreve os tópicos */board_state*, para obter o estado mais recente do tabuleiro e o */human_turn*, para que receba a sinalização de que é a vez do humano jogar e deve assim estar atento a mudanças do estado do tabuleiro.

A deteção do estado do tabuleiro é contínua, quer isto dizer que mesmo que não tenha sido feita nenhuma jogada é publicado no tópico */board_state* o estado atual, mesmo que este seja o mesmo estado publicado anteriormente, é assim da responsabilidade do nó que deteta jogadas verificar se o estado recebido corresponde a um novo estado do jogo ou se ainda é o mesmo.

Supondo um qualquer estado do jogo S_i o estado seguinte é S_{i+1}. O estado inicial é S_0, uma vez que para detetar uma jogada é necessário proceder à comparação com o estado anterior esta só pode ocorrer do estado S_1 em diante.

Um estado tem, por cada peça, a informação do quadrado onde esta está, a sua cor e a sua localização no mundo. Para efeitos da deteção de uma jogada apenas o nome do quadrado e a sua cor são necessários pelo que daqui em diante apenas essa informação será apresentada.

Desta forma um estado inicial pode ser representado da seguinte forma:

$$\begin{aligned} S_0 = \{&(a1, blue), (c1, blue), (e1, blue), (g1, blue), (b2, blue), (d2, blue), (f2, blue), (h2, blue), \\ &(a3, blue), (c3, blue), (e3, blue), (g3, blue), (b6, red), (d6, red), (f6, red), (h6, red), \\ &(a7, red), (c7, red), (e7, red), (g7, red), (b8, red), (d8, red), (f8, red), (h8, red)\} \end{aligned} \tag{5.1}$$

A equação 5.1 corresponde a um estado onde só existem peças no tabuleiro, mas na verdade todas as peças são monitorizadas, mesmo as que estão fora do tabuleiro, isto porque caso haja necessidade de promover uma peça a dama, esta tem de ser encontrada fora do tabuleiro, assim, um estado mais realista conteria, para além das peças já descritas, quatro damas que começam o jogo fora do tabuleiro e são representadas por (outside,red_dama) no caso da dama vermelha e (outside,blue_dama) no caso da dama azul.

Sempre que um estado do tabuleiro é recebido é armazenado em *current_board_state*, este é por sua vez comparado com o estado anterior que está armazenado em *previous_board_state* para determinar a jogada feita pelo humano.

A comparação consiste em percorrer cada peça de um estado e procurar se essa peça existe no outro, se tiver sido detetada uma diferença entre os dois, ou seja, num dos estados não existe uma peça que existe no outro, estes seguem para a deteção da jogada, se não tiver sido detetada nenhuma diferença não é feito nada e o nó fica a aguardar pela receção do próximo estado.

Como dito anteriormente, o estado do jogo só é detetado quando não existe movimentação sobre o tabuleiro que obstrue a visão da câmara. Assim, é possível que quando o jogador humano pegue numa peça, a remova do tabuleiro e posicione os braços fora da área que tapa a visão da câmara, enquanto pensa na jogada a fazer, um novo estado seja detetado onde esta peça já não se inclui.

Este novo estado quando comparado com o anterior originaria uma deteção de jogada, uma vez que existe uma diferença entre os estados, pois uma peça do estado anterior não foi encontrada no novo estado, no entanto esta jogada é inválida pois só é possível verificar que uma peça desapareceu de um quadrado mas não é possível saber para que quadrado foi, é preciso então garantir que uma peça removida de um quadrado foi posta noutro antes de se detetar a jogada feita, para isso basta obrigar a que todas as peças presentes no estado no jogo anterior estejam também no atual.

Se o número de peças no estado anterior é igual ao do estado atual e existe uma diferença na localização de uma peça é porque pode ser procurada uma jogada.

A partir de um qualquer estado S_i podem ser obtidos n estados S_{j_n}, isto porque num qualquer estado podem existir n jogadas diferentes possíveis de serem executadas, ver figura 5.2, destes estados todos o estado S_{i+1} corresponde a apenas um deles não se sabendo à partida qual.

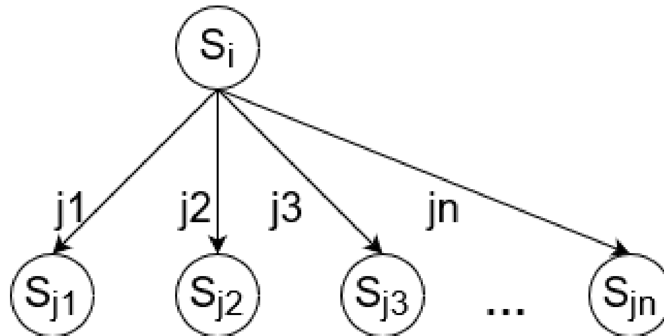


Figura 5.2: Um estado de jogo qualquer S_i pode originar n estados S_{i+1}. O estado S_{i+1} será apenas um dos estados S_{j_n}.

Assim, a função:

$$j_n = f(S_i, S_{i+1}) \tag{5.2}$$

deverá retornar qual a jogada j_n que originou o estado S_{i+1}. Esta é a função de deteção de jogada e devem-lhe ser fornecidos os estados atual S_{i+1} e o anterior S_i.

A deteção de uma jogada consiste em primeiro encontrar os quadrados de partida e de chegada da peça movida, para encontrar o quadrado de partida procuram-se no estado atual todos os elementos do estado anterior, que sejam da cor do jogador humano, aquele que não tiver sido encontrado corresponde ao desaparecimento da peça desse quadrado pelo que esse é o quadrado de partida. Pela mesma lógica, procuram-se no estado anterior todos os

elementos do estado atual, que sejam da cor do jogador humano, sendo que aquele que não for encontrado corresponde ao quadrado de chegada, pois significa que passou a existir uma peça nesse quadrado.

No entanto, saber o quadrado de partida e chegada de uma peça não é suficiente para que a jogada possa ser inserida no motor de jogo, isto advém das particularidades do jogo das damas onde numa mesma jogada várias peças adversárias podem ser capturadas, mas para o motor de jogo cada captura tem de ser feita individualmente.

Um solução possível seria obrigar o humano a fazer uma captura de cada vez, ou seja, pegar na peça, poisá-la no quadrado de destino, retirar a peça capturada, se existir captura, e após estes passos retirar os braços da zona do tabuleiro para que esta jogada fosse detetada, de seguida o humano poderia fazer o segundo salto, caso tivesse possibilidade para o fazer. Teria de repetir este procedimento para todos os saltos que tivesse de fazer, ora esta não é uma solução prática.

Uma solução mais atraente seria conseguir decifrar os saltos dados pelo humano apenas pelo quadrado de partida, chegada e pelas peças capturadas.

Suponhamos a situação da figura 5.5, a peça vermelha parte do quadrado **e3**, salta para o quadrado **c5** para capturar a peça adversária em **d4**, de seguida faz o segundo salto, uma vez que tem possibilidade de capturar uma segunda peça, e salta para o quadrado **e7**. Nesta jogada dois saltos foram realizados, o primeiro de **e3** para **c5** e o segundo de **c5** para **e7**, no entanto, se compararmos os estados do jogo antes da jogada ser feita e o estado depois da jogada estar terminada apenas é detetada a jogada **e3-e7**. Esta jogada não pode ser enviada para o motor de jogo pois não é uma jogada válida. Faltam então descobrir os saltos intermédios, neste caso faltava descobrir o quadrado **c5**.

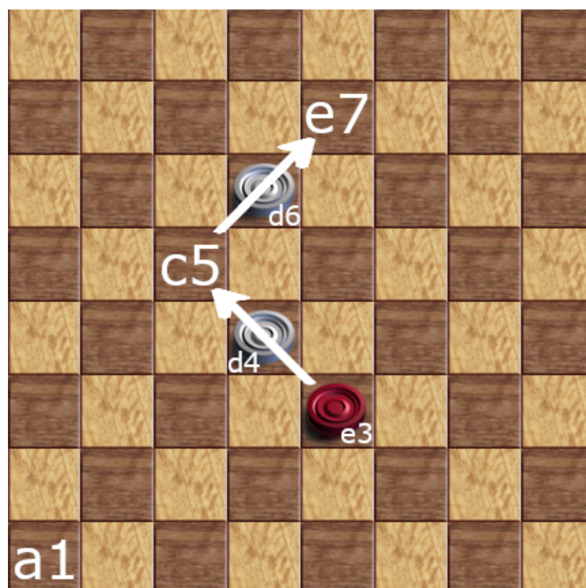


Figura 5.3: Representação de uma jogada composta por dois saltos e consequentemente duas capturas.

Na verdade o problema que se procura resolver é o de descobrir os quadrados intermédios no caminho seguido pelo peça jogada pelo humano, ou seja, sabendo o quadrado de partida, o

quadrado de destino e os quadrados das peças capturadas descobrir os quadrados em falta.

Este problema pode ser resolvido ao ser transposto para um grafo, desta forma cada quadrado onde uma peça pode estar é um vértice e os quadrados imediatamente adjacentes em diagonal estão conectados por ramos, assim pode ser aplicada teoria de grafos para encontrar caminhos entre os vértices.

O grafo resultante pode ser visualizado na figura 5.4, nesta figura é fácil constatar como foi construída a numeração representante de cada quadrado possível de ser usado no jogo das damas, bem como as relações que os quadrados partilham entre si.

Embora as peças de uma determinada cor apenas se possam movimentar numa direção, com exceção da dama, o grafo possuí ramos não direcionais para acomodar este último caso fazendo uso de lógica adicional para garantir o cumprimento da regras para as peças normais que será mais à frente explicado. Para a escolha de um caminho entre vértices apenas importa o número de ramos, ou seja, é como se estes tivessem peso 1.

Nesta nova representação a peça vermelha da figura 5.5 ocuparia a posição inicial no vértice **9**, a posição final seria no vértice **25**, as peças adversárias estariam nos vértices **14** e **22**, a jogada detetada seria **9-25** e o quadrado que falta descobrir para encontrar as duas jogadas é o **18**.

Figura 5.4: Grafo representativo dos quadrados onde as peças podem existir sobreposto no tabuleiro. Cada número dos vértices tem uma correspondência com o nome do mesmo quadrado na notação algébrica de xadrez, assim, o vértice **3** corresponde ao quadrado **a1**, o vértice **0** ao quadrado **g1** e assim sucessivamente.

O grafo foi construído usando o módulo **graph-tool**[4] para Python que fornece métodos para rapidamente encontrar os caminhos entre vértices.

O método *graph-tool.all_paths()* com o protótipo de função na listagem 5.1 retorna todos os caminhos do grafo **graph** que comecem no vértice **src** e terminem no vértice **dst** e cujo tamanho da distância seja inferior a **maxPathSize**.

[4] `https://graph-tool.skewed.de/`

Listagem 5.1: Protótipo da função *graph-tool.all_paths()*

```
paths = graph−tool.all_paths(graph, src, dst, maxPathSize)
```

Para o tamanho máximo de caminho contam todos os vértices que estejam entre o vértice incial e final, pelo que os vértices das peças adversárias também contam. Assim, no caso da figura 5.5 o tamanho máximo de caminho é 5, sendo dois vértices o inicial e o final, dois vértices das peças adversárias e um vértice para o quadrado intermédio.

Este valor pode ser obtido para qualquer número de saltos pela fórmula generalizada 5.3, mais ainda, o número de quadrados intermédios pode ser facilmente obtido através de $capturedPieces - 1$, visto que existe apenas um quadrado intermédio entre cada par de peças adversárias. Desta forma para uma qualquer jogada apenas é preciso saber o número de peças capturadas.

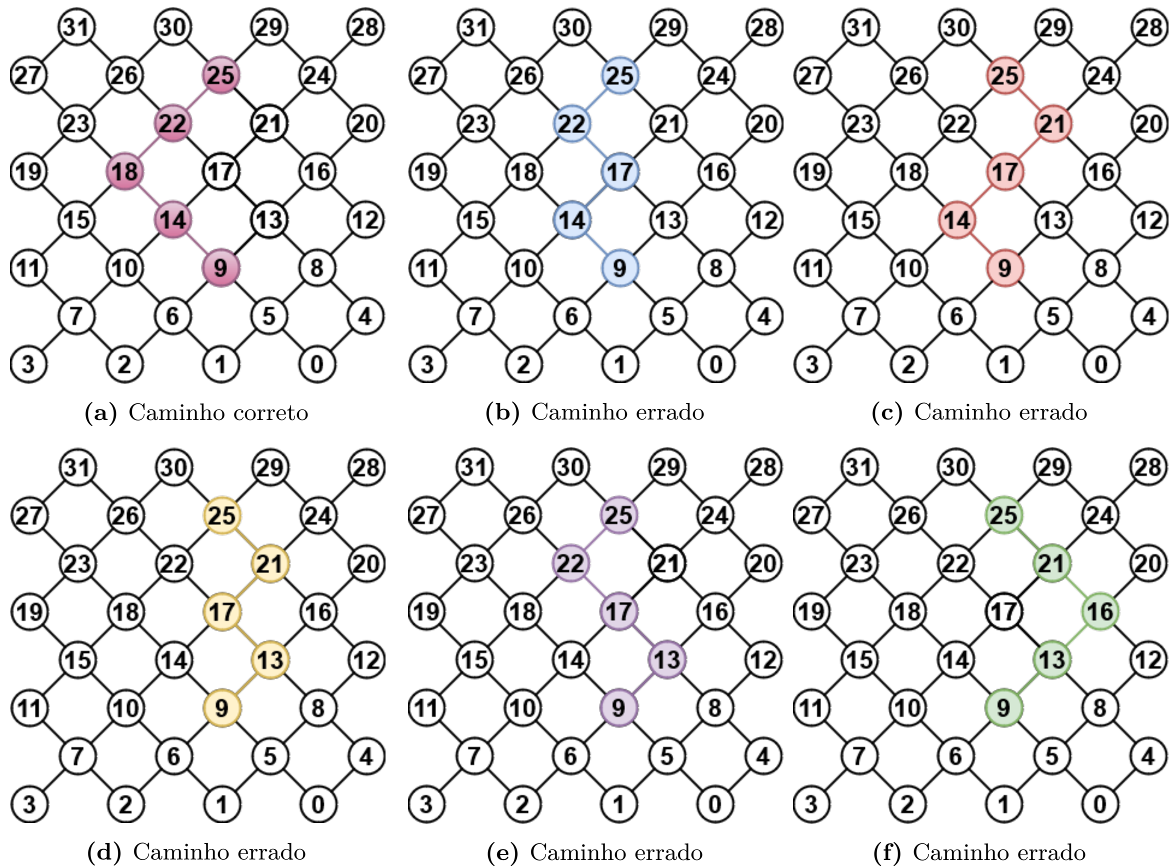
$$maxPathSize = capturedPieces + intermediateVertices + inicialVertex + finalVertex \tag{5.3}$$

Para o exemplo da figura 5.5 e inserindo os vértices inicial e de destino para um tamanho máximo de caminho igual a 5 o resultado esperado seria o caminho [9 14 18 22 25], caminho representado no grafo a) da figura 5.5. No entanto, os caminhos b), c), d), e) e f) também cumprem todos os requisitos para que sejam igualmente retornados por este método mesmo que do ponto de vista do jogo correspondam a jogadas impossíveis ou no caso de corresponderem a uma jogada possível não corresponde à jogada feita pelo humano. Há assim, necessidade de filtrar estes caminhos para que seja descoberto aquele que corresponde à jogada certa.

Para filtrar os caminhos começa-se por remover todos os caminhos que não contêm as peças capturadas, uma vez que estas têm obrigatoriamente de aparecer no caminho correto, daqui sobram os caminhos a) e b) da figura 5.5, todos os outros ou lhes falta uma das peças ou as duas.

Uma jogada elementar corresponde a um salto apenas, onde pode, ou não, existir captura de uma peça adversária. De seguida, o caminho correto é escolhido com base na observação de que cada uma destas jogadas elementares, onde existe captura, no jogo das damas, quando transpostas para o grafo, são válidas quando apresentam as seguintes regras: se partem de um vértice par chegam sempre a um vértice ímpar, se partem de um vértice ímpar chegam sempre a um vértice par.

Desta forma o caminho a) da figura 5.5 pode ser dividido em duas jogadas elementares, a primeira **9-18** e a segunda **18-25**, como se pode verificar, este caminho sendo o correto cumpre as regras acima mencionadas, se o vértice de partida é par o de chegada é ímpar e vice-versa. Já no caso de um dos caminhos errados, por exemplo o b) da figura 5.5, os caminhos elementares seriam **9-17** e **17-25** pelo não cumprem as mesmas regras e pode assim ser classificado como caminho errado. Para um caminho ser considerado errado basta que apenas uma das suas jogadas elementares seja considerada inválida, é o mesmo que dizer que um caminho para ser correto todas as suas jogadas elementares têm de ser consideradas válidas.



(a) Caminho correto (b) Caminho errado (c) Caminho errado

(d) Caminho errado (e) Caminho errado (f) Caminho errado

Figura 5.5: Estes são os caminhos retornados pela função *graph-tool.all_paths()* quando inseridos os parâmetros do exemplo da figura , todos estes caminhos começam no vértice **9**, terminam no vértice **25** e possuem tamanho igual a 5.

De referir ainda, que a diferença em módulo entre o vértice de partida e o de chegada, para uma jogada elementar, tem de ser superior a 1, isto porque no caso de ser a dama a fazer as capturas, esta pode-se movimentar para trás, exemplo presente na figura 5.6, isto permite que nos caminhos retornados existam dois caminhos onde a jogada elementar parte de um valor ímpar e chega a um valor par.

Ao se inserir o exemplo da figura 5.6, na função da listagem 5.1, onde **src** é o vértice **5**, **dst** é o vértice **7** e **maxPathSize** são 5 vértices, esta retorna vários caminhos, dois destes estão presentes na figura 5.7, o caminho correto é o verde, enquanto que o caminho vermelho está errado pois uma peça para capturar tem de saltar em linha reta, não pode fazer ângulos de 90º sobre a peça capturada, no entanto se na filtragem do caminho verde apenas se tivesse em conta que numa jogada elementar o vértice de destino tinha de ser par quando o vértice de partida fosse ímpar existia a possibilidade deste caminho ser avaliado como verdadeiro, ao se forçar que a diferença em módulo entre estes dois vértices seja superior a 1 o caminho vermelho é retirado enquanto que o verde é dado como o caminho verdadeiro.

Com o caminho correto descoberto resta apenas separá-lo nas suas jogadas elementares para que sejam enviadas uma a uma para o motor de jogo.

O motor de jogo corre no nó */checkers_engine*, este por sua vez recebe as jogadas do

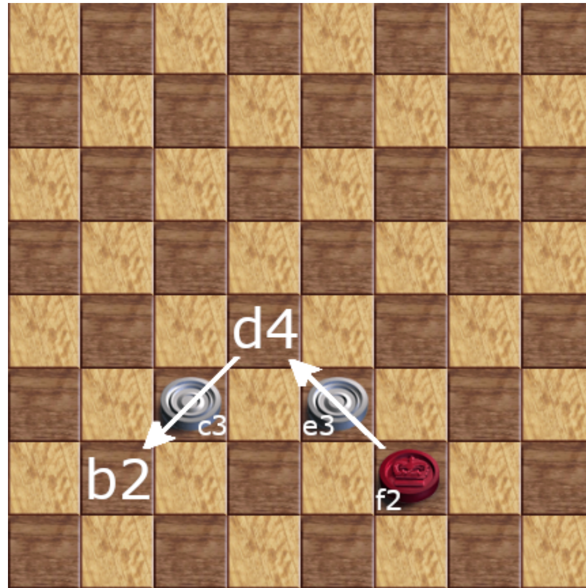
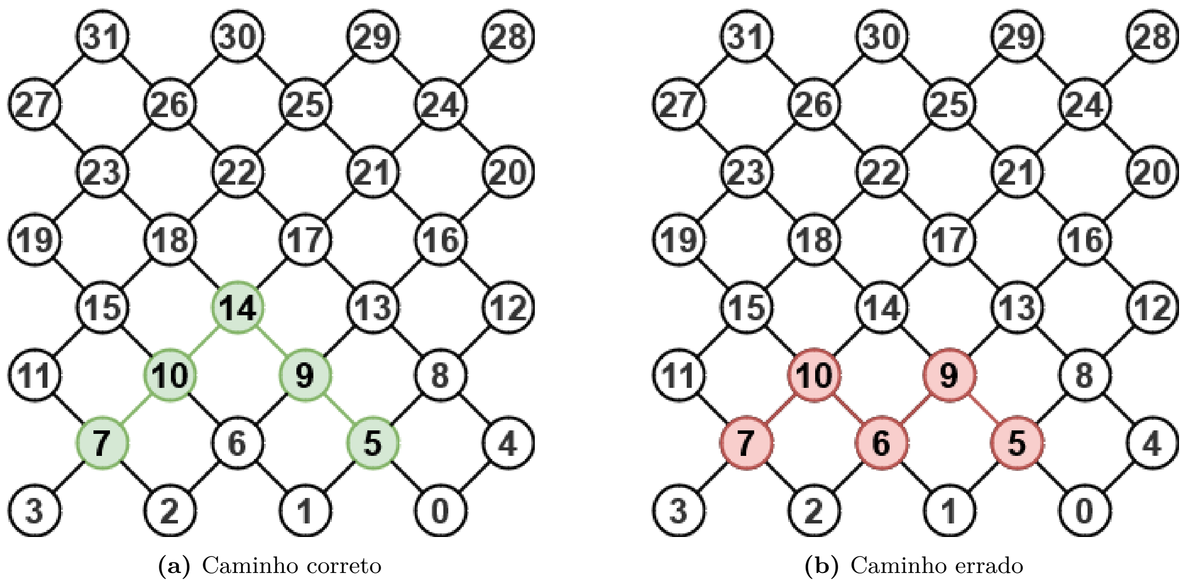


Figura 5.6: Nesta jogada a peça vermelha é uma dama pelo que pode capturar em qualquer direção, assim, após o primeiro salto pode capturar a segunda peça ao fazer um salto para trás.



(a) Caminho correto

(b) Caminho errado

Figura 5.7: Dois dos caminhos retornados pela função da listagem 5.1, ambos estes caminhos cumpriam a regra das suas jogadas elementares partirem de um vértice ímpar e chegarem a um vértice par, no entanto apenas o caminho verde está correto. Para eliminar o caminho vermelho basta verificar que a diferença entre estes vértices tem de ser, em módulo, superior a 1.

humano através de um serviço disponibilizado para o efeito com o nome de *play_human_move*, este serviço para além de fazer a jogada do humano retorna a validade da mesma. Se a jogada for válida o jogo pode prosseguir, para isso é preciso informar o nó do motor de jogo que pode criar a jogada do braço robótico, este processo é feito através do tópico */engine_turn*, sempre que o nó */detect_human_move* publicar uma mensagem neste tópico o nó */checkers_engine* faz a sua jogada e envia-a para ser executada pelo braço robótico.

Caso a jogada do humano tenha sido inválida, o nó */detect_human_move* não publica no tópico */engine_turn*, informa o humano que a jogada feita foi ilegal e espera que este a corrija. Quando for detetada a nova jogada, se esta for válida, é publicada uma mensagem no tópico */engine_turn* que permite que o motor de jogo faça a sua jogada.

O motor de jogo escolhido não estava preparado para fornecer a informação da jogada feita por ele, apenas atualizava internamente o seu estado de jogo para o estado após a sua jogada, desta forma houve necessidade de modificar o motor de jogo para que retornasse o quadrado de partida e chegada da sua jogada, bem como as peças capturadas por si.

Esta informação é por sua vez enviada para o nó */pick_and_place* através do serviço *play_engine_move*, este serviço aceita dois argumentos, o primeiro é a informação das peças capturadas, o segundo é um *array* com os quadrados por onde passa a peça jogada pelo motor de jogo, por exemplo, o *array* ['d6','c5'] significa que o motor de jogo moveu a peça que estava no quadrado **d6** para o quadrado **c5**, neste caso a jogada não corresponde a nenhuma captura pelo que o argumento das peças capturadas fica vazio. No caso do motor de jogo efetuar uma jogada composta por vários saltos é preciso enviar, para além do quadrado final, os quadrados intermédios, por exemplo o *array* ['e7','c5','e3'] significa que o braço robótico deve mover a peça em **e7** para o quadrado **c5**, mas não a deve poisar neste, de seguida deve movê-la para **e3** e aqui sim já a pode poisar. Desta forma o braço simula os pequenos saltos feitos por cima das peças.

Visto que no jogo das damas a peça que captura é posta sempre num quadrado livre, a remoção das peças capturadas pode ocorrer após o movimento da peça que captura. No caso do xadrez, por exemplo, uma peça quando captura outra substitui-a nesse quadrado, haveria assim necessidade de remover primeiro a peça e só depois mover a peça a jogar.

5.5 Motor do jogo dos peões

Para o motor de jogo dos peões houve a necessidade de criar um motor de raiz, para isso o método de decisão escolhido foi o *minimax*, este, determina a jogada que o computador deve fazer com base na avaliação do estado do jogo várias jogadas após o estado atual[5].

5.5.1 O algoritmo *minimax*

No algoritmo *minimax* a avaliação de um estado de jogo varia muito de jogo para jogo, no entanto, uma forma simples de o fazer é contar o número de peças que restam ao computador após um certo número de jogadas e contar as peças que sobram ao oponente, se o computador

[5] `https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1`

tiver mais peças é porque esse estado de jogo o beneficia, caso contrário esse estado de jogo é-lhe prejudicial, daqui se conclui que o objetivo do computador é MAXimizar a diferença entre as suas peças e as do adversário, enquanto que o objectivo do oponente é MINImizar esta diferença[6], esta característica dá nome ao algoritmo.

Para a avaliação dos estados do jogo seguintes começa-se por aplicar ao estado atual todas as jogadas possíveis, no caso de ser o computador a jogar, são feitas todas as suas jogadas possíveis, após cada uma destas jogadas chega-se aos estados do jogo sobre o qual o oponente deve jogar e a cada um desses estados são feitas as jogas possíveis do oponente, este processo repete-se até que se atinja a profundidade desejada na árvore de jogadas, de notar que quando maior esta profundidade o número de estados a avaliar aumenta exponencialmente, desta forma deve ser escolhido um limite razoável para que o algoritmo chegue a uma jogada forte num espaço de tempo reduzido.

Na figura 5.8 os nós a azul correspondem ao computador e os nós vermelhos ao oponente, que pode ser por exemplo o humano, no nível 0 está um nó azul, este é a raiz, o que significa que é a vez do computador jogador, neste estado o computador tem 3 jogadas possíveis a **a1**, a **a2** e a **a3** que transportam o jogo para os estados no nível 1, de momento ainda não são feitas quaisquer avaliações sobre os estados do jogo, para cada um destes estados de jogo o humano tem 2 jogadas com exceção do estado após a jogada **a3** onde o humano tem 3 jogadas possíveis que avançam o jogo para o nível 2, finalmente, é a vez do computador jogar de novo, este tem agora apenas duas jogadas para fazer por cada um dos estados de jogo atingidos, anteriormente, exceto no estado de jogo após a jogada **v7** onde o computador volta a ter 3 possíveis jogadas disponíveis.

Uma vez que o limite de profundidade foi atingido é agora necessário avaliar cada um dos estados atingidos no nível 3 para que o computador saiba qual a jogada a fazer no nível 0.

Após a jogada **a4** o jogo atinge um estado cuja avaliação resulta numa pontuação de -3, favorável ao humano, no entanto, se o computador fizer a jogada **a5** a avaliação é agora de 4, favorável ao computador, pelo que o algoritmo escolhe esta como sendo a jogada a ser feita pelo computador.

A avaliação neste nó do nível 2 não corresponde à avaliação do seu estado do jogo mas sim à avaliação do estado do jogo após a jogada que maximiza esta avaliação, isto significa que apenas os estados de jogo do nível 3 são avaliados e com base nessa avaliação são escolhidas as jogadas anteriores que resultam no melhor estado final para o computador, tendo em conta que o humano também faz a melhor jogada possível para si. No mesmo nível da árvore, mas no nó seguinte, o computador tem à sua disposição duas outras jogadas, a **a6** e a **a7**, destas a mais favorável para si é a **a6** pelo que é escolhida pelo algoritmo.

É agora a vez do humano decidir qual a sua jogada entre **v1** e **v2**, naturalmente, este escolhe aquela que minimiza a pontuação, opta assim pela jogada **v2** e a pontuação do nó filho passa para o nó pai. Finalmente, o mesmo processo é aplicado a todos os restantes nós até que no nível 0 o algoritmo decide a jogada que deve realmente ser feita pelo computador, a melhor jogada é a jogada **a2** pelas razões já descritas.

[6] `https://www.youtube.com/watch?v=l-hh51ncgDI`

Fazer esta jogada significa que mesmo que o humano jogue as suas melhores jogadas o computador terá uma vantagem em relação ao humano na avaliação do jogo daí a 3 jogadas, ou seja, no nível 3.

De notar que no nível 3 existem estados do jogo melhores para o computador do que aqueles atingidos pela jogada **a2**, como é o caso do estado após a jogada **a12**, no entanto, uma vez que os estados após as jogadas **a14** e **a15** são mais benéficas para o humano este nunca fará a jogada **v5**, que resultaria numa má pontuação para si, e se o humano nunca faria a jogada **v5** então o computador não deve nunca fazer a jogada **a3**.

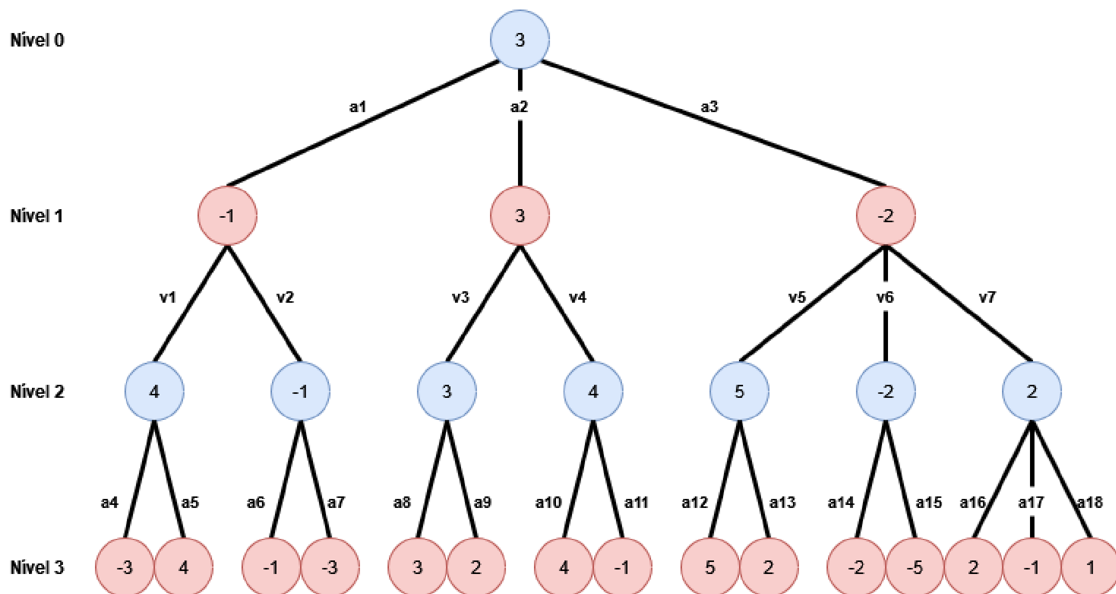


Figura 5.8: Árvore representativa do método de pesquisa *minimax*. Os nós a azul correspondem ao computador e ao jogador que prefere maximizar a avaliação, os nós a vermelho correspondem ao oponente, que pode ser um humano, e que prefere minimizar a avaliação. Esta árvore apresenta uma profundidade de 3 pois podem ser feitas no máximo 3 jogadas seguidas. A avaliação começa a ser feita das folhas para a raiz ou seja do nível 3 para o nível 0. Os ramos correspondem às jogadas que cada jogador pode fazer num determinado estado de jogo e estas podem variar conforme o estado.

Este método é eficaz na obtenção de uma jogada mas não é eficiente se a profundidade, e assim o número de jogadas que antecipa, aumentar. Uma forma de tornar o algoritmo mais rápido é através da implementação de *alpha-beta prunning*[13]. Aplicar *alpha-beta prunning* consiste em ignorar certos caminhos da árvore não sendo necessário avaliá-los a todos. Na visualização em árvore da figura 5.8 a aplicação de *alpha-beta prunning* consiste realmente em aparar os ramos da mesma, resultando numa árvore com menos folhas e assim menos estados para avaliar.

Nesta árvore e tendo em conta que a avaliação dos estados é feita da esquerda para a direita, podia-se aplicar *alpha-beta prunning* após a avaliação da jogada **a10**, isto porque, quando se estiver a avaliar a jogada **a10** já se sabe que o humano não vai escolher a jogada **v4**, independentemente do resultado da avaliação da jogada **a11**, uma vez que nesse nível é o computador a escolher e este escolherá sempre a maior pontuação, ora, se na jogada **a10** já

se obtém uma pontuação de 4, para que o computador escolha **a11** a pontuação terá de ser superior a 4, mas 4 já é maior que a pontuação obtida após **v3**, pelo que é garantida ser esta a melhor jogada para o humano após a avaliação da jogada **a10**.

Neste caso a poupança em termos de avaliações de estados de jogo foi de apenas um, mas se a profundidade fosse maior podia-se logo no nível 2 decidir não avaliar mais nenhum estado após a jogada **a11** o que podia, efetivamente, corresponder a um grande número de avaliações.

5.5.2 Implementação do motor de jogo dos peões

Um motor de um jogo precisa, acima de tudo, de manter internamente o estado atual do jogo, para poder obter a partir deste as suas jogadas. Assim, uma vez que jogo dos peões usa o tabuleiro de 8x8 quadrados, foi criada uma matriz de 8x8 células para representar a posição das peças no tabuleiro num determinado momento do jogo.

Às peças brancas corresponde o valor 2 e às pretas o 1, sendo que uma célula vazia tem o valor 0. Define-se que a célula (7,0) corresponde ao quadrado **a1** na notação algébrica de xadrez e o quadrado **b1** fica na célula (7,1), assim as letras são constantes nas colunas e os números são constantes nas linhas da matriz. Uma representação desta matriz pode ser vista na figura 5.9.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	2	2	2	2	2	2	2	2
7	0	0	0	0	0	0	0	0

Figura 5.9: Matriz do estado de jogo do motor de jogo dos peões. O estado representado nesta matriz corresponde ao início de um jogo, os valores a 1 representam as peças pretas e os valores a 2 as peças brancas.

Na inicialização do jogo a matriz fica com o valor 1 nas células todas da linha 1 e com o valor 2 nas células da linha 6, isto se o humano jogar com as peças brancas, caso o humano jogue com as peças pretas então o valor da linha 1 passa a ser 2 e da linha 6 para a ser 1.

De seguida, tanto no caso de ser o humano a jogar ou de ser o computador é necessário saber quais a jogadas possíveis no estado atual do jogo, no primeiro caso para validar a jogada do humano ao compará-la com todas as possibilidades, se a jogada do humano não estiver na lista de jogadas possíveis então é porque a jogada não é válida, no segundo caso para que o computador consiga avaliar as várias jogadas para que seja escolhida uma para ser feita por ele.

A matriz é percorrida célula a célula, se o seu valor for 0 não há necessidade de determinar uma jogada a partir dessa célula e passa-se à célula seguinte. Se o valor corresponder ao jogador que deve jogar nesta vez então devem ser determinadas as jogadas para essa peça.

Na situação da figura 5.9 onde o humano joga com as peças brancas estas só se podem mover no sentido decrescente das linhas, ou seja, uma peça na linha 6 só se pode mover para a linha 5 ou 4. No entanto, só se podem mover dois quadrados apenas quando partem do quadrado inicial, nos restantes só se podem mover um quadrado de cada vez.

Uma jogada é representada no motor da seguinte forma:

$$Move = (src_row, src_col, dst_row, dst_col) \tag{5.4}$$

Desta forma duas jogadas possíveis para o humano seriam (6,0,5,0) ou (6,0,4,0) no caso de avançar dois quadrados. Para que uma jogada seja possível é ainda necessário verificar se o quadrado de destino se encontra vazio, visto que no jogo dos peões as peças apenas capturam na diagonal.

Para adicionar uma jogada que corresponde a uma captura basta verificar os quadrados na linha imediatamente em frente e nas duas colunas adjacentes ao quadrado de onde se encontra a peça que captura, se existir nesse quadrado uma peça da cor oposta então é possível fazer-se uma captura caso contrário não se pode adicionar essa jogada.

Estes casos já cobrem a quase totalidade das jogadas no tabuleiro todo. Falta no entanto adicionar a jogada *en passant*, esta requer um cuidado adicional visto que só pode ser feita em certas condições muito específicas, são elas: a peça do humano estar na linha 3, o computador jogar umas das suas peças a partir da linha inicial para a linha 3 numa das colunas adjacentes à peça do humano e a jogada *en passant* só está disponível para o humano quando esta jogada do computador tiver sido a última jogada feita, para isto é necessário guardar sempre a jogada feita anteriormente, desta forma pode-se saber se a jogada *en passant* está disponível num qualquer momento do jogo. Na figura 5.10 podem-se ver estas condições e os vários passos da jogada.

Para as jogadas do computador basta aplicar a mesma lógica no sentido oposto. Com as jogadas determinadas é possível, agora, fazer as jogadas do humano para isso e como dito anteriormente basta verificar primeiro se a jogada existe na lista de jogadas legais, caso exista basta mudar o valor da célula de partida para 0 e o valor da célula de chegada para o valor da peça do humano. Já no caso do computador a jogada não é tão trivial, visto ser preciso implementar o método *minimax* descrito anteriormente.

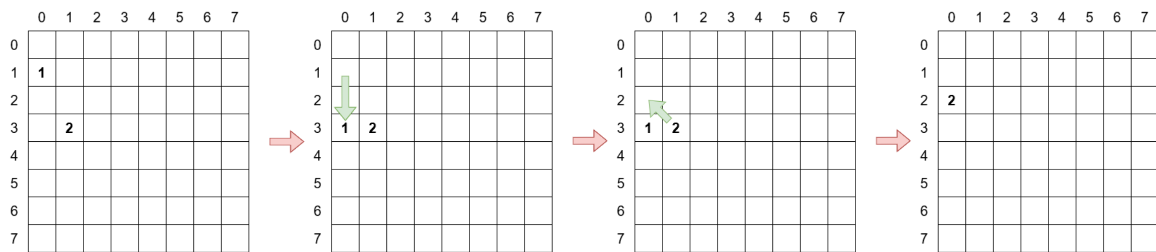


Figura 5.10: Representação da jogada *en passant*, os valores das células vazias foram removidos para melhorar a visualização. Apenas imediatamente após ao movimento da peça 1 é que a jogada *en passant* está disponível para o jogar da peça 2.

A função que implementa o algoritmo *minimax* recebe, inicialmente, o estado do jogo sobre o qual deve fazer uma nova jogada, a partir deste estado determina todas as jogadas que o computador pode fazer e calcula o estado do jogo se cada uma destas jogadas fosse efetuada no estado atual.

A função chama-se a si própria para cada um destes estados no entanto agora alterando um parâmetro para informar que as jogadas a serem criadas na próxima iteração devem ser as do humano, após ter determinado as jogadas deste e de ter criado os novos estados de jogo a função chama-se de novo para estes novos estados modificando de novo o parâmetro para avaliar agora a resposta do computador a estes estados.

Esta ação recursiva termina quando a profundidade definida for atingida, ou ter sido atingido um estado onde se pode obter um vencedor. Em qualquer dos casos o estado é avaliado e o seu valor retornado pela função para a instância que a chamou.

A avaliação de um estado de jogo segue os valores da tabela 5.1, considerando que o computador joga com as peças pretas, desta forma por cada peça preta a avaliação é incrementada em uma unidade, por cada peça branca, do humano, esta avaliação é decrementada também numa unidade, daqui se retira que um estado de jogo onde existam mais peças pretas que brancas favorece o jogador das peças pretas, por outras palavras, avaliações positivas favorecem o computador e avaliações negativas favorecem o humano.

Piece	Black	White	Win for black	Loss for black	Depth
Value	1	-1	100	-100	maxDepth - currentDepth

Tabela 5.1: Valores usados para determinar a avaliação de um estado do jogo.

No entanto, no caso de existirem mais peças pretas que brancas mas o estado de jogo corresponder a uma vitória das peças brancas é necessário que esta vitória seja refletida na avaliação, desta forma o valor, em módulo, dado a uma vitória é bastante superior à maior diferença entre peças para garantir que o facto de existir uma vitória é mais importante do que um jogador ter mais peças que o outro, o valor escolhido para uma vitória do jogador das peças pretas foi 100 e para o jogador das peças brancas -100.

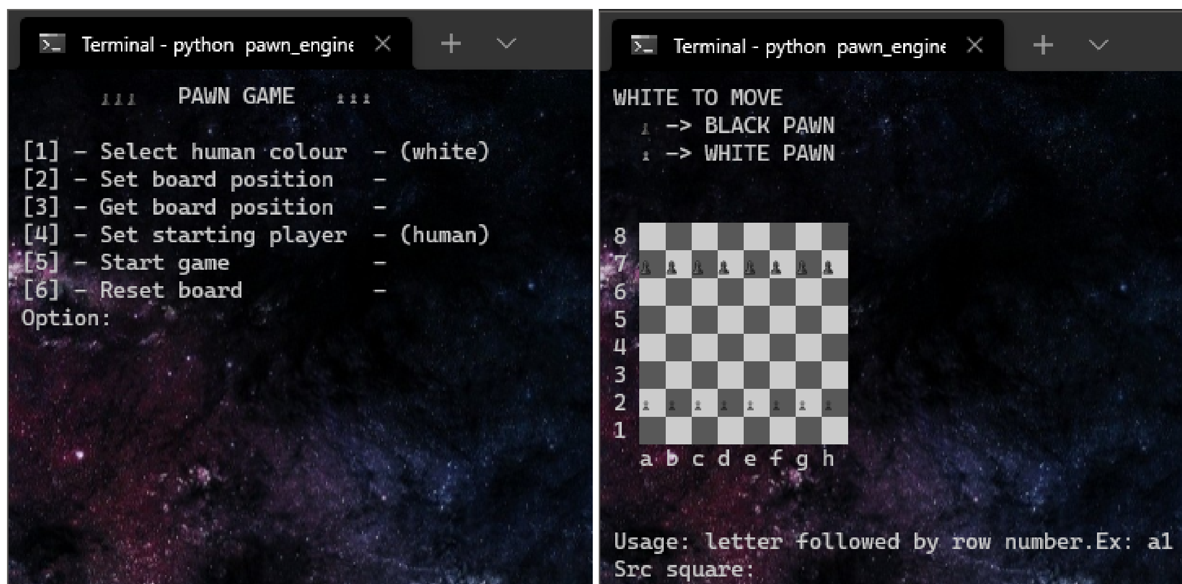
Mais ainda, caso existam duas formas de alcançar uma vitória para o computador mas um delas ocorre em menos jogadas tem de existir uma forma de as distinguir. Esta distinção

é feita através da profundidade necessária para atingir essa vitória, assim, quanto menos profundidade for necessária mais rapidamente se atinge essa mesma vitória pelo que devem ser feitas essas jogadas.

Para a incorporação desta mecânica basta somar à avaliação calculada anterior a diferença entre a profundidade máxima definida para o motor de jogo e a profundidade atual, desta forma se a profundidade atual for 0 o valor a somar é a profundidade máxima, no caso de a vitória só acontecer quando se chegar ao último nível da árvore o valor a somar é 0.

Com base nas avaliações destes estados todos esta função retorna a melhor jogada seguindo os princípios descritos na secção 5.5.1.

Este motor de jogo embora tenha sido criado para iteragir com o sistema geral é também capaz de ser utilizado de forma independente, para isso foi criada uma *Graphical User Interface (GUI)* no terminal de comandos, ver a figura 5.11, aqui para além de ser possível receber as instruções do humano, este pode ver o estado do jogo atual, mudar a cor com que joga, quem começa a jogar, definir o tabuleiro numa posição qualquer, obter o estado do tabuleiro num momento qualquer e repor o estado inicial do jogo.



(a) Menu para interação com o motor de jogo dos peões **(b)** Tabuleiro para visualização do estado do jogo

Figura 5.11: Interfaces gráficas para interação direta com o motor de jogo dos peões.

No caso de apenas integrar o motor de jogo com o sistema esta pode ser feita através de funções para enviar a jogada do humano e obter a jogada do computador.

CAPÍTULO 6

Resultados

6.1 Introdução

Este capítulo mostra através da realização de experiências a robustez e a viabilidade dos vários módulos que constroem esta arquitetura, desde a deteção do tabuleiro até jogar contra um humano. Para isso foram feitas várias experiências tanto em ambiente real como simulado.

6.2 Experiências e Resultados

De seguida serão apresentadas as experiências realizadas e os seus resultados.

6.2.1 Deteção do tabuleiro

Detetar o tabuleiro consiste em saber a sua localização em relação a um dos componentens do sistema, seja o braço robótico seja a câmara. Para se saber a localização do tabuleiro em relação ao outro basta que se saiba também a relação entre estes dois.

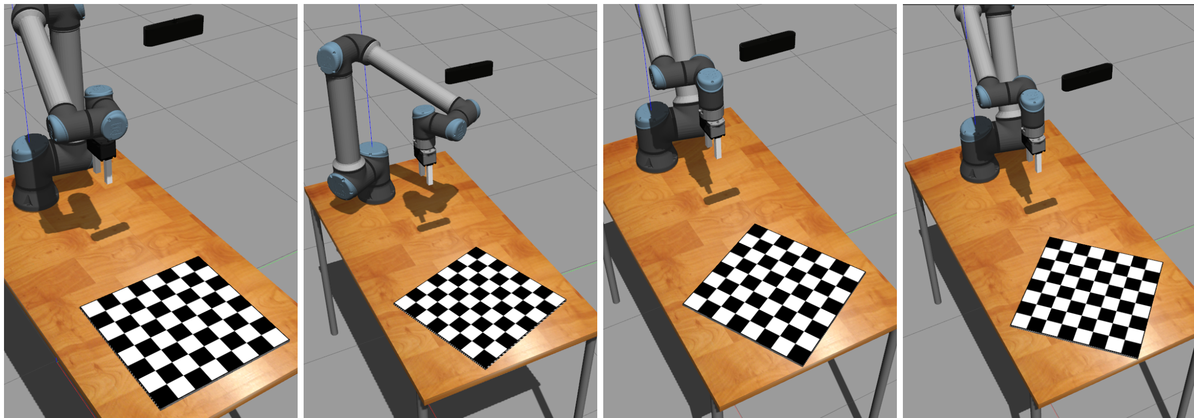
Ambiente Simulado

Para a deteção do tabuleiro em ambiente simulado considera-se que os três principais componentes do sistema estão perfeitamente calibrados entre si, ou seja, são conhecidas à partida as transformações que devem ser feitas entre cada um destes para que se obtenha uma determinada pose em qualquer uma das suas *frames*.

O tabuleiro foi posicionado na mesa com um série de ângulos, apresentados na figura 6.1, a câmara foi posta imediatamente acima deste e paralela ao tampo da mesa com o mesmo ângulo desta.

Para que o tabuleiro seja detetado apenas são necessárias descobrir as componentes x e y do centro deste, uma vez que o tabuleiro está sobre o tampo da mesa a componente z não é relevante. Em termos de ângulo já ocorre o oposto, apenas é necessário conhecer o ângulo segundo o eixo z, pois corresponde a uma rotação segundo um eixo perpendicular ao tampo da mesa.

Como o tabuleiro está sempre sobre a mesa e as peças sobre o tabuleiro ou sobre o tampo as restantes medidas não precisam de serem descobertas pois não afetam a determinação do estado do tabuleiro.



(a) Ângulo de 0 graus **(b)** Ângulo de 20 graus **(c)** Ângulo de 30 graus **(d)** Ângulo de 45 graus

Figura 6.1: Estas figuras apresentam a configuração utilizada para a realização das experiências. O tabuleiro foi posto com vários ângulos, a câmara estava posicionada sempre no mesmo sítio diretamente acima do tabuleiro.

Assim o tabuleiro foi posicionado para cada uma destas experiências sempre nas coordenadas da tabela 6.2.

Coordenadas	x	y	
Valor	0.92	0.1	m

Tabela 6.1: Posição do tabuleiro, em relação à base do braço robótico, em todas as experiências.

A tabela seguinte apresenta os resultados após média de 3 medições para cada um dos ângulos de tabuleiro.

0 graus = 0 radianos					
$x(m)$	$\varepsilon_x(\%)$	$y(m)$	$\varepsilon_y(\%)$	$yaw(rad)$	$\varepsilon_{yaw}(\%)$
0.919	0.1	0.098	2	0.003	-
20 graus ≈ 0.35 radianos					
$x(m)$	$\varepsilon_x(\%)$	$y(m)$	$\varepsilon_y(\%)$	$yaw(rad)$	$\varepsilon_{yaw}(\%)$
0.92	0	0.1	0	0.351	0.2
30 graus ≈ 0.52 radianos					
$x(m)$	$\varepsilon_x(\%)$	$y(m)$	$\varepsilon_y(\%)$	$yaw(rad)$	$\varepsilon_{yaw}(\%)$
0.922	0.2	0.1	0	0.521	0.19
45 graus ≈ 0.79 radianos					
$x(m)$	$\varepsilon_x(\%)$	$y(m)$	$\varepsilon_y(\%)$	$yaw(rad)$	$\varepsilon_{yaw}(\%)$
0.914	0.6	0.099	1	0.776	1.77

Tabela 6.2: Posição do tabuleiro, em relação à base do braço robótico, em todas as experiências.

Ambiente Real

Em ambiente real a localização verdadeira do tabuleiro em relação ao referencial *world* é muito díficil de ser medida, pois este é um ponto imaginário não existente na realidade. Fica assim complicado comparar a posição do tabuleiro determinada pelo algoritmo em teste com a real posição do tabuleiro.

Uma forma, no entanto, de se poder comprovar que o tabuleiro realmente foi detetado corretamente, é aplicar às imagens recolhidas pela câmara os eixos na origem do referencial do tabuleiro. Isto é, o sistema começa por determinar a transformação da câmara para o centro do tabuleiro, de seguida, usando a transformação inversa, transformam-se os pontos: origem, um ponto no eixo x e outro no eixo y, todos da *frame* do tabuleiro para a *frame* da câmara.

Com estes pontos é possível criar os eixos da *frame* do tabuleiro na *frame* da câmara e com auxílio da classe *image_geometry::Pinhole-CameraModel* podem ser projetados na imagem recolhida pela câmara.

Se a interseção destes eixos corresponder com o centro do tabuleiro e se os eixos estiverem coincidentes com as arestas dos quadrados centrais do tabuleiro é porque este foi bem detetado.

Tendo isto em conta os tabuleiros testados foram os da figura 6.2, o tabuleiro esquerdo corresponde ao tabuleiro criado para favorecer a sua deteção, sendo composto por quadrados pretos e brancos com cantos bem definidos para que estes fossem mais facilmente detetados, o tabuleiro da direita corresponde a um tabuleiro comum, mais tradicional para se jogarem damas ou outro tipo de jogo.

Figura 6.2: Tabuleiros usados para teste da deteção do tabuleiro. O tabuleiro da esquerda foi impresso em papel e apresenta melhor contraste entre os quadrados, enquanto que o da direita representa o tabuleiro tradicional feito de madeira.

Os tabuleiros foram dispostos da mesma forma que os testes em ambiente de simulação, sendo a câmara montada no topo destes e aproximadamente paralela ao tampo da mesa. A

figura 6.3 apresenta a posição dos vários elementos, câmara, tabuleiro e mesa, usados nesta experiência.

(a) Tabuleiro impresso em papel (b) Tabuleiro de madeira

Figura 6.3: Estas figuras apresentam a configuração utilizada na deteção dos tabuleiros em ambiente real.

O resultado da deteção do tabuleiro de madeira pode ser visto na figura 6.4, na imagem 6.4a o eixo x representado a vermelho e o y a verde, como é possível verificar tanto o centro do tabuleiro como o ângulo deste foram encontrados com bastante precisão, isto porque a interseção dos eixos está no centro do tabuleiro e os eixos coincidem com as arestas dos quadrados centrais do tabuleiro.

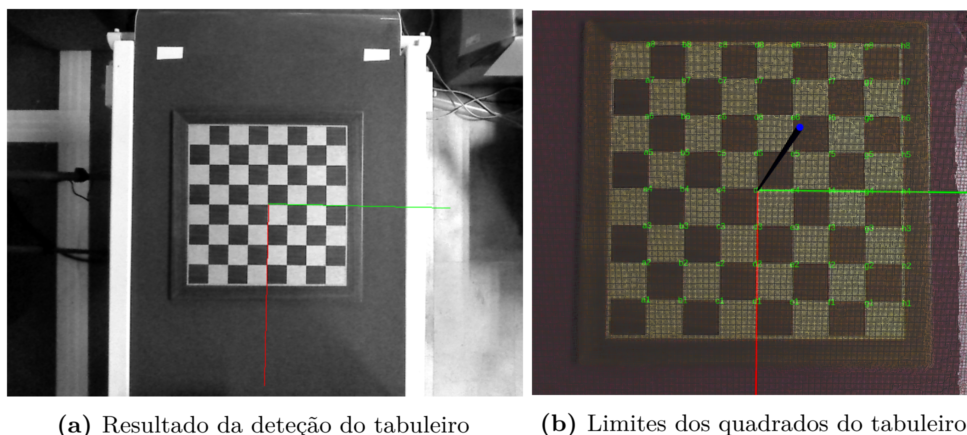
Já na imagem 6.4b está presente a *point cloud* recolhida pela câmara, nela foram sobrepostas as *labels*, a verde, correspondentes a cada quadrado, como se pode observar estão perfeitamente sobre o canto superior direito do quadrado a que pertencem.

Este é um resultado importante pois é com esta informação que as peças serão identificadas e subsequentemente o estado do tabuleiro será construído. Mais uma vez podem-se observar os eixos da *frame* do tabuleiro agora a partir da sua publicação na árvore de transformações **tf** do ROS.

Na figura 6.5 pode-se observar o mesmo resultado agora para o tabuleiro impresso em papel, que apresenta melhor contraste nos cantos, este contraste é bastante percetível na imagem 6.5a onde os quadrados brancos são mais claros do que na imagem 6.4a.

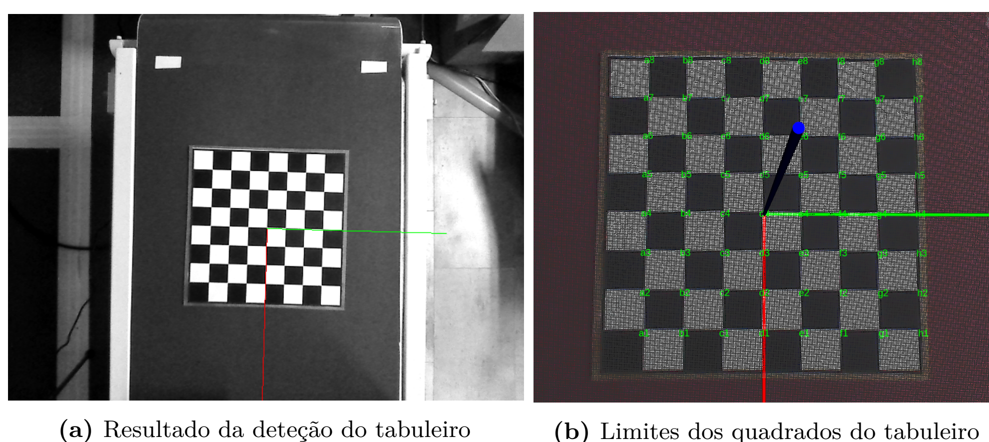
O software usado para visualizar estas *point clouds*, *labels* e eixos da **tf** foi o programa rviz[1] do ROS.

[1] https://github.com/ros-visualization/rviz/tree/melodic-devel



(a) Resultado da deteção do tabuleiro (b) Limites dos quadrados do tabuleiro

Figura 6.4: A figura da esquerda apresenta o resultado da sobreposição dos eixos da *frame* do tabuleiro na imagem recolhida pela câmara, a imagem da direita mostra em cada canto superior direito dos vários quadrados a *label* dada a esses quadrados.



(a) Resultado da deteção do tabuleiro (b) Limites dos quadrados do tabuleiro

Figura 6.5: A figura da esquerda apresenta o resultado da sobreposição dos eixos da *frame* do tabuleiro na imagem recolhida pela câmara, a imagem da direita mostra em cada canto superior direito dos vários quadrados a *label* dada a esses quadrados.

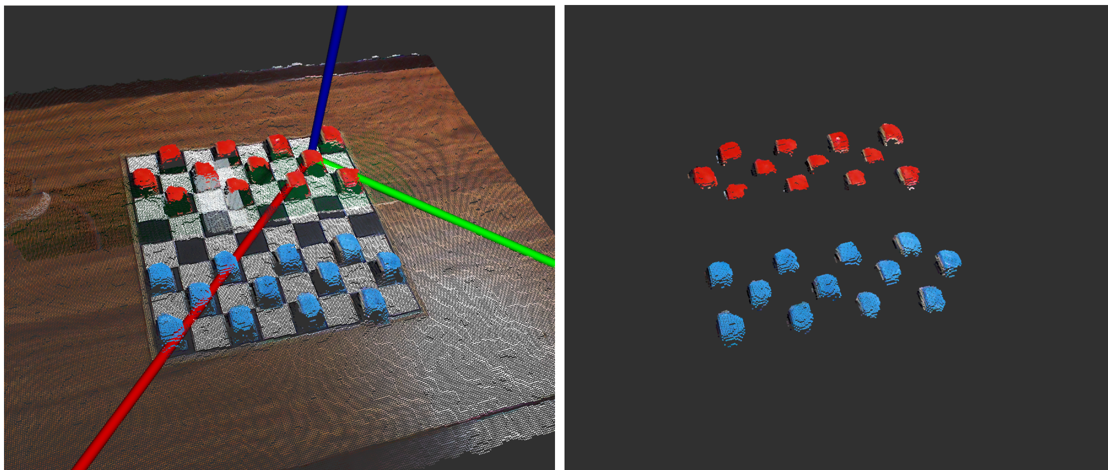
6.2.2 Segmentação das peças e estado do tabuleiro

Para que o estado do tabuleiro seja corretamente determinado é necessário que as peças sejam segmentadas da *point cloud*, para que isto aconteça o plano da mesa deve ser encontrado para assim serem feitas as filtragens com base neste.

Para esta experiência o tabuleiro foi colocado numa mesa com um tampo o mais planar possível, a câmara foi posicionada imediatamente acima de modo a que toda a zona de jogo estivesse no seu campo de visão e o tabuleiro foi populado com 24 peças nas posições inciais do jogo das damas, este será o pior caso para a deteção pois é quando a densidade de peças é maior no tabuleiro.

Na imagem 6.6a é possível verificar, pela presença dos eixos da *frame* da mesa, que esta foi bem detetada, de notar que o facto dos eixos não estarem paralelos nem à borda da mesa nem às arestas do tabuleiro significa que estejam errados, isto porque a parte crítica é apenas que a origem esteja no tampo da mesa e que os eixos x e y a vermelho e verde, respetivamente,

sejam paralelos ao tampo, como é, efetivamente, o caso.



(a) Resultado da deteção do tampo da mesa **(b)** *Point cloud* apenas com pontos das peças

Figura 6.6: A figura da esquerda apresenta o resultado da sobreposição dos eixos da *frame* da mesa *point cloud* visualizada no programa rviz, a imagem da direita mostra o resultado depois de ser removido o tampo da mesa e todos os pontos abaixo deste ficando apenas os pontos das peças.

Embora o resultado anterior mostre que as peças foram corretamente segmentadas ainda não é possível concluir sobre a qualidade da identificação das mesmas, ou seja, se a segmentação foi suficiente para separar as peças em *clusters* e se o estado do tabuleiro foi bem detetado, para isso na imagem 6.7 foi posicionado sobre cada peça uma etiqueta que representa, sucintamente, o resultado do estado do tabuleiro.

Esta etiqueta começa por identificar o quadrado onde a peça reside, a sua cor e a posição do centroide da peça usado para que o robô no futuro agarre as peças. A posição determinada para a peça foi usada para posicionar as etiquetas, pelo que, se estas estiverem sobre a peça correta é porque foi bem detetada.

A confirmação da validade da identificação pode ser feita com base nas etiquetas a verde, na mesma imagem, que identificam os quadrados com os seus nomes segundo a notação algébrica de xadrez.

Estas etiquetas verdes estão posicionadas sobre os cantos superiores direitos dos quadrados a que pertencem para mostrarem uma das posições determinadas de um dos limites de um quadrado do tabuleiro, ou seja, não representam o centro do quadrado identificado.

No vídeo[2] é possível visualizar estes resultados de forma mais interativa pois é mais fácil ter a perceção da tridimensionalidade dos dados que aqui são apresentados. Neste começa-se por mostrar, ao centro, a captação em *point cloud* da câmara com os eixos da *frame* da mesa presentes sobre ela, de seguida é apresentada a *point cloud* após a filtragem, ou seja, as peças segmentadas.

Finalmente, a captação inicial é reposta para que as etiquetas das peças possam estar contextualizadas com o tabuleiro assim como os nomes dos quadrados deste.

[2]Deteção do estado do tabuleiro: `https://youtu.be/9eWs7Fo2hz0`

Figura 6.7: Visualização do estado do tabuleiro. A branco as etiquetas que identificam as peças individualmente com base na sua posição no tabuleiro, a verde a identificação dos quadrados determinada anteriormente.

No canto superior direito é possível visualizar a captação de vídeo da mesma experiência e imediatamente em baixo pode ser vista a publicação da mensagem do tipo *BoardState* onde estão reunidas as informações sobre todas as peças presentes na área de jogo, a primeira peça nesta mensagem encontra-se expandida pelo que se pode ver a sua *color*, *label*,*name* e *pose*.

6.2.3 Deteção das jogadas

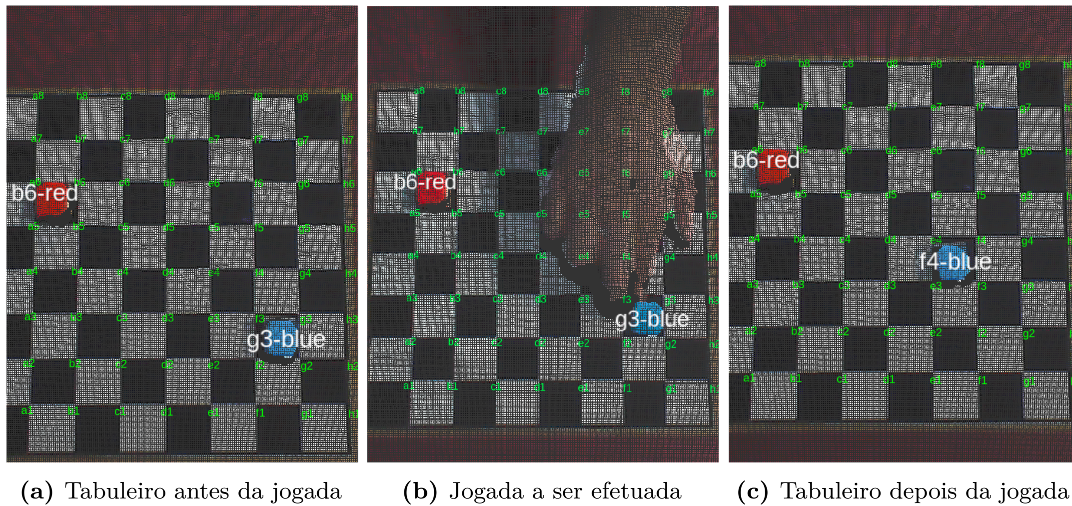
Para testar a deteção de jogada foram posicionadas sobre o tabuleiro duas peças, uma de cada cor. O tabuleiro e a câmara encontravam-se na mesma situação da figura 6.3a.

A razão de serem necessárias pelo menos duas peças, uma de cada cor, deve-se ao facto de que quando o humano informa o sistema do começo de um novo jogo este inicia o motor de jogo com o estado atual das peças, para que possa começar a jogar no estado atual, ora se apenas existisse uma peça, para testar a deteção de uma jogada simples, o que aconteceria é que o motor de jogo iria terminar imediatamente, informando que o jogador da peça em questão tinha vencido, isto porque não existem mais peças em jogo.

Assim, foram necessárias duas peças, para que não exista um estado de vitória no tabuleiro e se possa mover uma delas de forma a verificar se o movimento foi corretamente detetado.

O teste ocorreu em ambiente real e os vários momentos da execução de uma jogada estão presentes na figura 6.8, como pode ser visto o movimento do braço obstrui a visão da câmara sobre o tabuleiro, se a deteção do estado do tabuleiro não fosse interrompida durante o movimento do humano, esta originaria estados do tabuleiro com peças em falta e dessa forma

seriam detetadas jogadas erradamente, assim com movimento sobre o tabuleiro o estado deste não é publicado, para que apenas os estados das imagens 6.8a e 6.8c sejam detetados.



(a) Tabuleiro antes da jogada **(b)** Jogada a ser efetuada **(c)** Tabuleiro depois da jogada

Figura 6.8: Figuras da *point cloud* recolhida pela câmara nos vários momentos de uma jogada, com as etiquetas identificativas do estado do tabuleiro sobrepostas nas peças.
Na figura a) consegue-se ver a peça vermelha no quadrado **b6** e a azul no **g3**, na figura b) é possível ver a perturbação causada pelo braço do jogador humano na *point cloud* e a obstrução que esta faz no tabuleiro, finalmente na figura c) está presente o estado do tabuleiro após a jogada com a peça azul já no quadrado **f4**.

Como dito anteriormente o humano configura o sistema através de uma HRI, esta pode ser em parte visualizada na figura 6.9, sendo a zona 1 da imagem o gestor de jogo onde o humano pode ver o estado de alguns dos parâmetros do sistema como o jogo atualmente a ser jogado, o jogador que começa a jogar e a cor das peças com que cada jogador joga.

Estes valores podem ser alterados pelo humano através desta interface, para além destes pode iniciar o jogo, terminar o jogo, imprimir o estado do tabuleiro atual, forçar a calibração do tabuleiro de jogo e definir a dificuldade do motor de jogo, sendo que esta característica não foi ainda implementada.

Na zona 2 da figura 6.9 o humano pode ver a evolução da deteção de jogada, sendo que de momento não houve ainda alteração no estado do tabuleiro.

Finalmente na zona 3 da mesma figura pode ser visto o estado do jogo interno do motor de jogo através de uma representação gráfica e reduzida do tabuleiro, os valores a 0 correspondem às posições do jogo das damas sem peças no tabuleiro. Os valores 1 e 2 correspondem às peças azul e vermelhas, respetivamente.

A jogada presente na figura 6.8 corresponde ao mover da peça azul do quadrado **g3** para **f4**. A comprovação que o sistema detetou bem esta jogada está na figura 6.10, onde na zona 2 se pode ver uma mensagem a informar que o estado do tabuleiro mudou e que a jogada detetada foi **g3-f4**, esta jogada foi enviada para o motor de jogo o qual validou a jogada como sendo legal e atualizou o seu estado de jogo interno, como se pode ver no segundo tabuleiro impresso na zona 3.

Com a validade da jogada assegurada é agora a vez do motor jogar, pelo que a deteção de

```
roscore ...  nacl@n...  nacl@n...  nacl@n...  /home/...  nacl@n...
-------- GAME MANAGER ---------
The current values are:
Game: checkers
Starting player: human
Human plays with: blue
Robot plays with: red

[1]-Choose game.
[2]-Define starting player.
[3]-Set players' colours.
[-]-Set engine difficulty.
[5]-Restore values.
[6]-Start game.
[7]-Finish game.
[8]-Print board PDN.
[9]-Calibrate board position.
Option:
```

1

```
File  Edit  View  Search  Terminal
Updating board state!

Board didn't change.
[INFO] [1634753159.730064]:
Updating board state!

Board didn't change.
[INFO] [1634753160.072631]:
Updating board state!

Board didn't change.
[INFO] [1634753160.433524]:
Updating board state!

Board didn't change.
[INFO] [1634753160.792532]:
Updating board state!

Board didn't change.
[INFO] [1634753161.140916]:
Updating board state!
```

2

```
Terminal
File  Edit  View  Search  Terminal  Help
Arguments received!!
using data path= /home/nacl/catkin_ws/src/chess_ur1
25993 Positions Loaded
Opening Book database loaded from /home/nacl/catkin_
eckers/data/opening.gbk
2pc database loaded from /home/nacl/catkin_ws/src/ch
ta/2pc.cdb
3pc database loaded from /home/nacl/catkin_ws/src/ch
ta/3pc.cdb
4pc database loaded from /home/nacl/catkin_ws/src/ch
ta/4pc.cdb
Endgame databases loaded
B:W24:B9.
  0   0   0   0
0   0   0   0
  2   0   0   0
0   0   0   0
  0   0   0   0
0   0   0   1
  0   0   0   0
0   0   0   0
```

3

Figura 6.9: HRI antes da jogada. Na zona 1 está o gestor de jogo onde o humano pode inserir as suas ordens para atualizar o sistema, iniciar ou terminar um jogo, entre outras, ou visualizar apenas o valor dos parâmetros atuais. Na zona 2 o humano pode ver como está a deteção da jogada e a validade desta quando for detetada, na zona 3 pode-se ver o estado do jogo interno ao motor de jogo. Os 0's correspondem aos quadrados vazios do jogo das damas, o 1 a uma peça azul e o 2 a uma peça vermelha.

jogada fica interrompida, informando o humano que é a altura do motor de jogo fazer a sua jogada através das mensagens na zona 2. O decorrer da deteção da jogada pode ainda ser visto no vídeo[3], em nota de rodapé.

[3]Deteção de uma jogada: `https://youtu.be/jG7hQEaYpkI`

```
File Edit View Search Terminal         File Edit View Search Terminal Help
Updating board state!                  3pc database loaded from /home/nacl/catkin_ws/src/ch
                                       ta/3pc.cdb
Board didn't change.                   4pc database loaded from /home/nacl/catkin_ws/src/ch
[INFO] [1634753174.294525]:            ta/4pc.cdb
Updating board state!          2       Endgame databases loaded
                                       B:W24:B9.
Board didn't change.                     0   0   0   0
[INFO] [1634753174.643466]:            0   0   0   0
Updating board state!                    2   0   0   0
                                       0   0   0   0
Board didn't change.                     0   0   0   0
[INFO] [1634753176.571269]:            0   0   0   1
Updating board state!                    0   0   0   0
                                       0   0   0   0
Board changed.
                                       Playing human move
The detected move: g3-f4               B:W24:B9.
Paths:                                   0   0   0   0
[ 8 13]                                0   0   0   0
Legal move!                              2   0   0   0
[INFO] [1634753176.942719]:            0   0   0   0
Engine turn                              0   0   1   0
[INFO] [1634753177.320825]:            0   0   0   0
Engine turn                              0   0   0   0        3
[INFO] [1634753177.691717]:            0   0   0   0
Engine turn
```

Figura 6.10: Após a jogada ter ocorrido há, agora, alteração em duas zonas da HRI, não assim necessidade de apresentar de novo a zona 1, na zona 2 é apresentada agora a jogada detetada, a validade da mesma e a indicação que passou a ser a vez do motor de jogo jogar. Na zona 3 o estado do jogo interno ao motor foi atualizado para refletir a jogada do humano uma vez que a mesma é válida.

6.2.4 Situação de jogo

Finalmente, é agora necessário testar uma situação normal de jogo, onde o braço robótico execute as jogadas feitas pelo motor de jogo, em resposta a jogadas feitas pelo humano.

Para isso a mesa, o tabuleiro e as peças foram posicionados como na figura 6.11, a câmara desta vez está posicionada um pouco mais perto do lado do humano e não diretamente acima do tabuleiro. A câmara está nesta posição para garantir um maior volume de movimentação por parte do robô.

Como se pode ver pela figura 6.12 as peças foram postas com vários ângulos no tabuleiro, para mostrar a capacidade de compensação do robô no agarrar da mesma.

O braço do humano é uma parte importante da componente de deteção da jogada pois é a sua interferência com o volume da zona de jogo por cima do tabuleiro que sinaliza o sistema para que este não detete estados de tabuleiro enquanto o humano joga.

Por forma a simular esta parte importante foi usado um bloco, com altura suficiente para que sempre que fosse feita uma jogada este pudesse criar a ilusão da presença de um braço na zona de jogo. A figura 6.13 apresenta um exemplo deste bloco e como foi usado.

No vídeo[4], pode ser visto o jogo dos peões jogado na íntegra de início ao fim, a figura 6.14

[4] Jogo completo dos peões: `https://youtu.be/-5qzjVpHvH8`

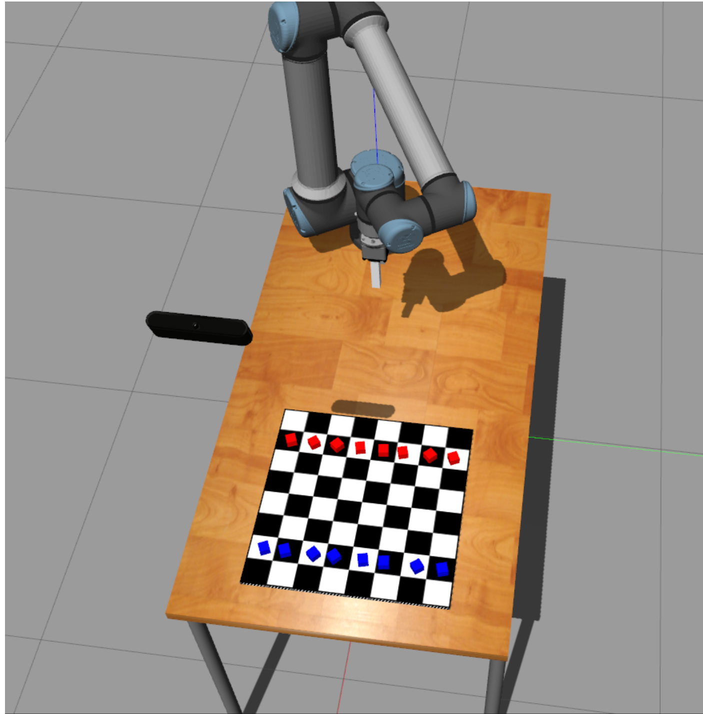


Figura 6.11: Posicionamento dos vários componentes usados para simular um jogo completo.

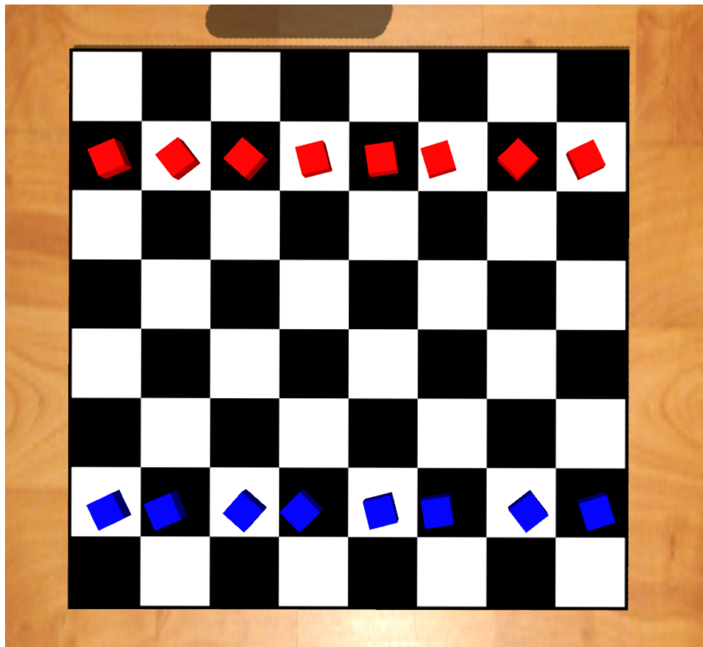


Figura 6.12: Vista de cima do tabuleiro onde se pode comprovar a disposição das peças com vários ângulos.

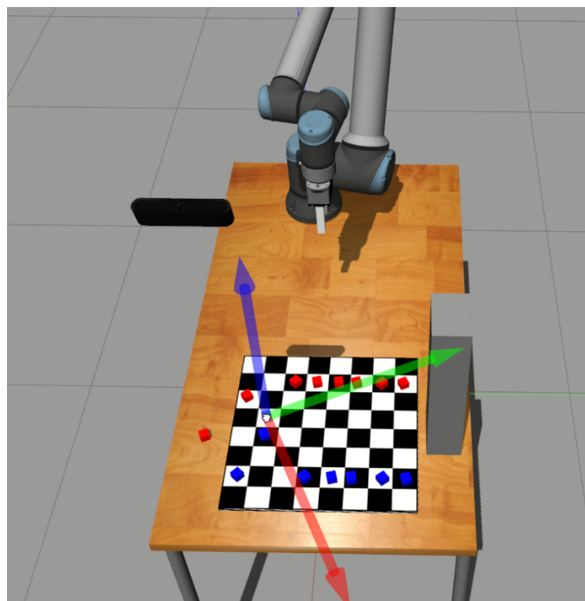


Figura 6.13: No lado direito do tabuleiro pode-se ver o bloco cinzento usado para simular a presença de um braço humano na altura de se fazer a jogada deste.

apresenta uma imagem retirada desse vídeo a meio de uma jogada do robô onde se pode notar que este posiciona as suas garras corretamente conforme a posição da peça a agarrar.

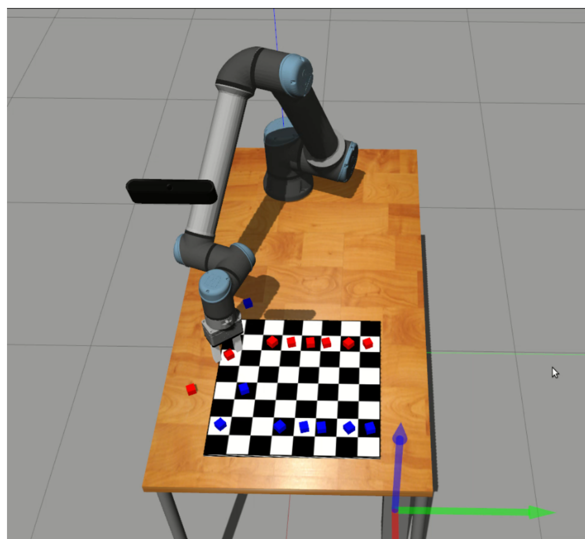


Figura 6.14: Imagem adaptada do vídeo do jogo dos peões.

De referir ainda que foi adicionado um atraso de 3 segundos entre o início da vez do braço jogar e este começar efetivamente a movimentar-se, para que numa situação real o humano tenha algum tempo para sair do alcance do robô em segurança caso seja necessário.

6.2.5 Execução de jogadas complexas

Embora já tenham sido, anteriormente, testadas várias partes deste sistema, não houve ainda oportunidade de verificar a execução de algumas jogadas dos jogos em questão, como são: a jogada *en passant* no jogo dos peões, a captura múltipla e promoção de uma peça a

dama no jogo das damas. Dado o grau de complexidade envolvido nestas jogadas, devido ao elevado número de peças manipuladas ou à especificidade da mesma como é o caso da jogada *en passant*, faz todo o sentido que sejam também elas testadas.

Assim, a jogada *en passant* pode ser vista no vídeo[5], em nota de rodapé. Neste, pode-se verificar que a peça capturada é removida antes de o robô fazer a sua jogada, embora neste caso específico não fosse necessário.

No entanto, no jogo dos peões esta não é a realidade para a maioria das jogadas, visto que quase todas as capturas acontecem com a peça a ir para o quadrado da peça capturada, por isso, na jogada *en passant* a peça capturada é também removida antes da jogada se efetuar.

Já no caso do jogo das damas, o vídeo[6] apresenta uma captura tripla seguida de uma promoção de peça a dama, de notar que todos estes passos acontecem de forma fluida e seguida e que todas as peças removidas para fora do tabuleiro pelo robô seguem as posições estipuladas na figura 3.3.

[5] Jogada *en passant*: `https://www.youtube.com/watch?v=Qh_4J5VoAtU`

[6] Captura tripla e promoção de peça a dama: `https://www.youtube.com/watch?v=KN_a59z1BH8`

CAPÍTULO 7

Conclusão

A partir das experiências e resultados é possível afirmar que a maioria das componentes críticas à validação desta arquitetura funcionam corretamente, tanto em ambiente simulado como real, sendo elas a deteção do tampo da mesa e remoção deste, para permitir segmentar as peças da *point cloud*, a deteção do tabuleiro com a identificação única e individual de todos os seus quadrados, a captação correta do estado do tabuleiro, através da posição que as peças ocupam neste, a deteção de jogadas feitas pelo humano e também a capacidade perceptual do sistema sobre a existência de movimento na zona de jogo.

Faltou apenas testar em ambiente real a aptidão por parte do robô verdadeiro na manipulação das peças, que não aconteceu pelo facto de não se ter conseguido atingir uma calibração correta, em tempo útil, da transformação entre a câmara e o braço robótico.

Não obstante, ficou provada a exequibilidade de uma arquitetura única poder ser capaz de dotar um braço robótico da habilidade de jogar diferentes jogos de tabuleiro contra humanos se todos os componentes nela presentes estiverem devidamente calibrados entre si.

Devido ao elevado ruído presente na captação da *point cloud* a determinação do ângulo das peças tornou-se mais complicada, uma solução melhor seria determinar este ângulo a partir da captação de vídeo. Para isso, pode-se usar a informação do centroide da peça para criar uma máscara na zona da imagem onde esta está, para a segmentar do resto da imagem, a partir daqui basta encontrar a zona do topo da peça dentro desta máscara e determinar o seu ângulo.

Como trabalho futuro, seria interessante implementar um maior número de tabuleiros, por exemplo o tabuleiro do jogo do galo, dotando a deteção do tabuleiro da capacidade de distinguir características próprias destes, como o número de quadrados, a cor destes, entre outros. Através da inserção, nesta arquitetura, de um nó, entre a deteção do estado do tabuleiro e a deteção de jogadas, com o intuito de identificar as peças pela sua forma, é possível facilmente adaptá-la para que consiga jogar xadrez, uma vez que esta arquitetura já está feita de modo a que cada peça possa ter uma identificação única, como acontece no caso da peça dama no jogo das damas.

Referências

[1] P. Kołosowski, A. Wolniakowski e K. Miatliuk, «Collaborative Robot System for Playing Chess,» em *2020 International Conference Mechatronic Systems and Materials (MSM)*, 2020, pp. 1–6. DOI: 10.1109/MSM49833.2020.9202398.

[2] C. Matuszek, B. Mayton, R. Aimi, M. P. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J. R. Smith e D. Fox, «Gambit: An autonomous chess-playing robotic system,» em *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 4291–4297. DOI: 10.1109/ICRA.2011.5980528.

[3] H. M. Luqman e M. Zaffar, «Chess Brain and Autonomous Chess Playing Robotic System,» em *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2016, pp. 211–216. DOI: 10.1109/ICARSC.2016.27.

[4] E. Kopets, A. Karimov, G. Kolev, L. Scalera e D. Butusov, «Interactive robot for playing russian checkers,» *Robotics*, vol. 9, n.º 4, pp. 1–15, 2020, cited By 0. DOI: 10.3390/robotics9040107. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85097807886&doi=10.3390%2frobotics9040107&partnerID=40&md5=6ccab284c690cb447b8b6acde3ea3090.

[5] T. Foote, E. Marder-Eppstein e W. Meeussen. (2017). «tf2::Matrix3x3 Class Reference,» URL: http://docs.ros.org/en/jade/api/tf2/html/classtf2_1_1Matrix3x3.html#details. (accessed: 17.06.2021).

[6] R. B. Rusu. (). «Filtering a PointCloud using a PassThrough filter,» URL: https://pcl.readthedocs.io/projects/tutorials/en/latest/passthrough.html. (accessed: 01.12.2020).

[7] G. Bradski, «The OpenCV Library,» *Dr. Dobb's Journal of Software Tools*, 2000.

[8] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart e T. Yu, «scikit-image: image processing in Python,» *PeerJ*, vol. 2, e453, 2014.

[9] R. Shaikh. (2020). «OpenCv Perspective Transformation,» URL: https://medium.com/analytics-vidhya/opencv-perspective-transformation-9edffefb2143. (accessed: 02.04.2021).

[10] P. Mihelich. (2013). «Pinhole camera model class reference,» URL: http://docs.ros.org/en/diamondback/api/image_geometry/html/c++/classimage__geometry_1_1PinholeCameraModel.html. (accessed: 07.04.2021).

[11] S. Tuerker. (). «Euclidean Cluster Extraction,» URL: https://pcl.readthedocs.io/projects/tutorials/en/latest/cluster_extraction.html#cluster-extraction. (accessed: 22.04.2021).

[12] M. T. Games. (). «The Rules of Draughts,» URL: https://www.mastersofgames.com/rules/draughts-rules.htm. (accessed: 16.04.2021).

[13] D. E. Knuth e R. W. Moore, «An analysis of alpha-beta pruning,» *Artificial Intelligence*, vol. 6, n.º 4, pp. 293–326, 1975, ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(75)90019-3. URL: https://www.sciencedirect.com/science/article/pii/0004370275900193.