Conceção de um robô virtual programável para ensino e treino de soluções integradas de automação

Pedro Filipe da Rocha Seabra

Dissertação de Mestrado

Orientador na FEUP: Prof. António Pessoa de Magalhães

Coorientador na Real Games: Eng. Bruno Vigário

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Mecânica

Janeiro 2020

Resumo

O propósito desta dissertação é apoiar a empresa Real Games no desenvolvimento de uma aplicação visual de vanguarda capaz de apresentar de forma sucinta e fácil de entender os conceitos básicos da robótica a estudantes de diversas áreas. Para tal, é especificado o desenvolvimento de um ambiente computacional 3D centrado num robô virtual programável capaz de executar as funções mais importantes para uma introdução ao tema da robótica industrial – nomeadamente, os três tipos de movimentos básicos (juntas, linear e circular), a manipulação de referenciais e o controlo de velocidade – e é construído e testado um protótipo do mesmo.

O trabalho inicia-se com uma revisão do estado da arte dos ambientes virtuais vocacionados para o ensino da robótica de forma a identificar os requisitos funcionais e de engenharia a adotar no projeto. Depois, é referido o desenvolvimento por parte da Real Games do ambiente virtual e das funções robóticas básicas implementadas no respetivo controlador. No sentido de tornar o robô programável de uma forma convencional é depois criada uma biblioteca na linguagem *Python* com as várias funções necessárias para cumprir esses requisitos. O trabalho termina com a exemplificação e teste do uso dessas funções.

No seu estado atual, esta aplicação é capaz de permitir ao utilizador realizar vários programas simples para testar o robô, assim como alguns programas mais complexos, sendo apresentada uma operação de paletização a título de exemplo. Para além disso suporta o protocolo de comunicação MODBUS TCP/IP permitindo por essa via a interação com aplicações externas.

Futuramente, a ideia é poder incluir o robô em aplicações de treino de soluções integradas de automação.

Palavras-chave: Robótica industrial, Ambientes virtuais 3D, Programação em *Python*, Interação gráfica, Aplicações de software de apoio ao ensino.

Development of a programmable virtual robot for teaching and testing automation technologies

Abstract

The aim of this project is to support the company Real Games in the development of a visual application capable of easily presenting the basics of industrial robotics to students. Therefore, it's detailed the development and testing of a 3D computational environment with a programmable virtual robot capable of executing the main functions necessary for an introduction into industrial robotics. Namely, the movement functions (joints, linear and circular), the frame manipulation functions, and the process used to control the robot velocity.

Whence this paper starts by analysing the state of art of the main virtual environments used for teaching industrial robotics. Observing which are the most important functions they possess, how they present them to the user and which ones of those we want to replicate in this project, which results in the specification of the project requirements. Then, its present the contribution of the Real Games which uses several engineering concepts to create an environment capable of supporting those desired functions, as well as allow their implementation. With that basic environment ready, it will then be developed a library of functions in *Python*, capable of performing all the movements and control methods specified in the project requirements. The project ends with the demonstration and testing of that library and all the functions it possesses.

In its current state, this application allows the user to develop several basic programs for testing as well as some complex ones, which may appear in real world applications. As an example of the robot capabilities it is shown a palletizing operation. Furthermore, the application also supports the communication protocol MODBUS TCP/IP, allowing it to interact with other applications. Hereafter it is desired to further develop this application and use it, in conjunction with others, for testing of automation technologies.

Agradecimentos

Gostaria de agradecer ao Professor António Pessoa de Magalhães pela sua orientação ao longo do projeto, e, em especial, pela ajuda disponibilizada na escrita da presente dissertação.

Aos Engenheiros Bruno Vigário e Gonçalo Vasconcelos, representantes da empresa Real Games, pela orientação fornecida e ajuda na especificação dos algoritmos desenvolvidos.

E agradeço ainda, de forma especial, aos meus pais e irmãos, que sempre me ajudaram e apoiaram ao longo de cada etapa.

Índice de Conteúdos

Índice de Figuras

Índice de Tabelas

Siglas e Acrónimos

API - *Application Programming Interface*

CAD - *Computer-Aided Design* - Desenho Assistido por Computador

FPS - *Frames Per Second* - Frames por segundo

IDE - *Integrated Development Environment*

IRL - *Industrial Robotic Language*

PID (controlador) - Controlador proporcional integral derivativo

PLC - *Programmable Logic Controller* - Controlador Lógico Programável

RPL - *Robot Programming Language* - Linguagem de programação de robôs

RVP - Robô Virtual Programável

TCP - *Tool Center Point* - Ponto Central da Ferramenta

1 Introdução

A robótica tem vindo a assumir um papel cada vez mais importante na automatização do mundo atual, favorecendo o desenvolvimento de uma grande variedade de indústrias graças à versatilidade inerente aos robôs industriais. Estes, graças à sua capacidade de programação e trabalho em contínuo, podem e têm vindo a assumir a realização de várias tarefas perigosas e repetitivas, aumentando não apenas a produtividade destas indústrias, mas também a qualidade geral dos seus produtos graças a possuírem também elevadas precisões(Pires 2012).

Consequentemente, o ensino dos vários conceitos da robótica industrial também tem vindo a ganhar grande importância. Porém, a sua programação e experimentação em robôs reais apresenta vários inconvenientes como o seu custo/perigo para o utilizador e para o robô em caso de acidente, o que é provável em situações de ensino/teste, para além da necessidade de interromper a linha de produção em aplicações industriais o que é inconveniente. Daí o interesse de realizar a programação em sistemas virtuais, ou seja, remotamente num computador, através de um ambiente virtual capaz de emular o robô, permitindo o ensino da sua manipulação sem os problemas enunciados. No entanto, esta programação requer uma maior especialização do operador, para além de ferramentas de simulação poderosas, o que é obviamente dispendioso de implementar. Ainda assim, as vantagens de segurança, eficiência e até mesmo custo a longo termo justificam a tendência para utilizar cada vez mais estes sistemas(Couto 2000).

Assim, o objetivo deste projeto é desenvolver e aproveitar as vantagens dos sistemas virtuais num contexto de ensino de robótica, ou seja, fornecer ao utilizador a possibilidade de programar um simulador de um robô inserido num ambiente virtual de desenvolvimento e teste de programas, onde o utilizador poderá testar à sua vontade as funções básicas existentes no domínio da robótica, assim como realizar pequenos programas e executá-los num controlador virtual.

1.1 Enquadramento da empresa Real Games e da FEUP

Esta dissertação resulta de um projeto realizado pela empresa Real Games em conjunto com a Faculdade de Engenharia da Universidade do Porto com o objetivo de desenhar, conceber e testar um Robô Virtual Programável (denominado daqui por diante por RVP).

A Real Games é uma empresa que trabalha na área do desenvolvimento de programas educacionais e simulação em 3D de soluções de automação industrial para escolas, universidades e outras organizações, especializando-se nessa área há mais de doze anos e tendo várias relações de pesquisa e desenvolvimento com universidades e centros de pesquisa(Real Games 2019b).

Assim, já tendo histórico nesta área, existe um interesse no projeto, mas o principal motivo de desenvolvimento será a incorporação deste RVP num *software* já existente na empresa: *Factory I/O*. O *Factory I/O* consiste num simulador de um ambiente industrial em 3D projetado para ajudar na aprendizagem das tecnologias de automação industrial como se pode ver na Figura 1.1. Este permite a criação rápida de uma fábrica virtual através da seleção de componentes industriais comuns, assim como a sua operação através de um controlador. Sendo os *PLCs* (*Programmable Logic Controllers* – Controladores Lógico Programáveis) o controlador mais comum em ambientes industriais, o *Factory I/O* é mais conhecido por ser uma aplicação para treino do uso de *PLCs*, mas este também possui outros controladores como microcontroladores e *SoftPLCs* e a possibilidade de comunicar com outras aplicações através de MODBUS(Real Games 2019a).

Figura 1.1 - Exemplo de ambiente virtual do software *Factory I/O*(Real Games 2019a)

O interesse no projeto advém do *Factory I/O* já possuir um robô cartesiano operacional e um robô articulado, como o que se pretende desenvolver neste projeto (a laranja na Figura 1.1), capaz de realizar apenas movimentos predefinidos, sendo um dos objetivos substituir esta versão limitada de um robô por um capaz de realizar qualquer movimento consoante a vontade do utilizador, possibilitando a execução e teste de um variado número de situações industriais.

De notar que, tanto o *Factory I/O*, como o RVP aqui a desenvolver, são aplicações destinadas a situações de aprendizagem, sendo o público-alvo utilizadores com pouco ou nenhum conhecimento destes sistemas, de onde vem o objetivo do RVP ser uma aplicação capaz de introduzir de forma fácil os conceitos da robótica industrial. Por isso, são ignorados alguns temas avançados que não são necessários para aprender o básico da programação de robôs (por exemplo, diferentes composições de robôs e pós-processadores) e é apenas emulado um único tipo de robô de forma a focar essas bases como, por exemplo, os tipos de movimentos disponíveis e a manipulação de referenciais. Obviamente, para aplicações reais, esses temas ignorados terão de ser tidos em conta, mas acredita-se que será muito mais fácil compreendê-los após a perceção dos conceitos básicos introduzidos pelo RVP.

O interesse da FEUP advém também desta componente didática que se pretende fornecer à aplicação, sendo que esta poderá ser usada, por exemplo, pelos alunos de engenharia mecânica em disciplinas de introdução a esta área.

Convém ainda referir que este projeto também possui interesse científico devido ao método de cinemática inversa a aplicar. Nesta aplicação a ideia é aplicar um "*cyclic coordinate descent algorithm*" alternativamente aos métodos matemáticos normalmente usados, o que é muito mais fácil de aplicar e exige um poder de calculo muito menor ao computador, mas possui uma menor precisão nos resultados. Este método tem vindo a ganhar muito poder em áreas como estatística computacional e *machine learning*, e, pode ser que venha também a ganhar importância para aplicações robóticas em que o erro de precisão resultante seja aceitável, servindo esta aplicação como um dos primeiros testes deste algoritmo nesta área. No entanto, não possuindo uma aplicação similar para comparação e visto que o mecanismo de cinemática inversa não pertence e quase não influencia a área de trabalho do autor, este tema não será tratado nesta dissertação(Kenwright 2012).

1.2 Objetivos da dissertação

Como foi dito, o objetivo principal deste projeto é desenhar, conceber e testar um robô virtual programável (RVP). No entanto, este robô e o ambiente que o envolve devem ser capazes de respeitar alguns requisitos que lhe permitam rivalizar com aplicações robóticas congéneres de referência.

Assim, e como aspeto fundamental, o RVP deve ser capaz de representar visualmente o robô a operar, permitindo identificar facilmente os seus componentes, como, por exemplo, a ferramenta em uso e as posições das suas juntas e motores.

Em segundo lugar, deve ser capaz de permitir a movimentação desses componentes livremente no ambiente gráfico, definindo já aqui a diferença entre o robô a desenvolver e o existente de momento no *Factory I/O*.

Possuindo a capacidade de manipulação gráfica, o RVP deve também ser capaz, em terceiro lugar, de permitir ao utilizador comandar o robô através de funções previamente definidas. Estas funções podem ser executadas através de métodos textuais ou gráficos, mas entre elas, devem estar no mínimo as três funções de movimento básicas usadas em robótica industrial: movimento de juntas, movimento linear e movimento circular. De notar que, como um objetivo secundário, o RVP deve também incluir várias outras funções como pausa ou manipulação de referenciais. A utilização destas funções individualmente deve permitir ao utilizador testá-las à sua vontade de forma a conseguir estudá-las e compreendê-las.

Em quarto lugar, o RVP deve permitir a criação de pequenos programas através das funções existentes, permitindo ao utilizador realizar pequenas operações.

Por fim, o RVP deve ainda ser destinado a um publico alvo que não esteja familiarizado com a área da robótica industrial, por isso outro objetivo é possuir uma interface fácil de utilizar e que apresente sucintamente os conceitos associados à robótica industrial.

1.3 Abordagem do problema

De forma a conseguir atingir os objetivos propostos, é primeiro necessário ter uma ideia clara do produto final a obter. Daí o projeto começar por estudar as aplicações existentes no mercado, identificando as funcionalidades que qualquer ambiente virtual para robótica deve possuir, assim como, várias características adicionais.

Estas funcionalidades devem depois ser avaliadas no contexto deste projeto e da posição que se pretende obter no mercado, nomeadamente, ser uma das melhores aplicações destinadas a situações de ensino. Através desta avaliação será então possível escolher quais as funcionalidades a implementar no projeto a desenvolver e agrupá-las nos requisitos funcionais a cumprir.

Tendo os requisitos funcionais definidos, é necessário definir e construir um protótipo da aplicação capaz de permitir a implementação desses requisitos, tendo em conta a interface que será fornecida ao utilizador final. Assim, nesta fase, deve ser definida toda a estética da aplicação, criado o esqueleto desta e implementados meios capazes de permitir o seu desenvolvimento futuro. No caso em questão, através da programação de algoritmos que realizem as funcionalidades desejadas.

De seguida, deve-se então programar os algoritmos necessários e testar as funções resultantes, de forma a verificar se os requisitos funcionais foram cumpridos satisfatoriamente. Todo este processo encontra-se resumido no esquema da Figura 1.2.

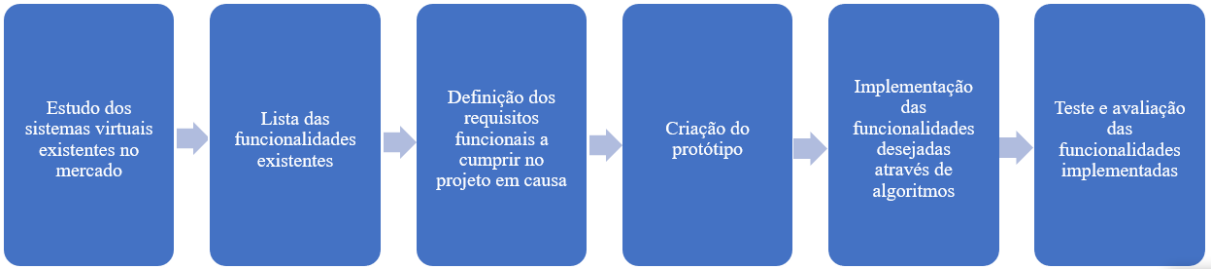

Figura 1.2 - Esquema resumo da abordagem aplicada

1.4 Organização da dissertação

De forma a seguir a planificação apresentada na secção anterior esta dissertação foi dividida em seis capítulos que são sumarizados de seguida:

No Capítulo 2 é apresentado o projeto a desenvolver, seguindo-se um estudo das aplicações existentes no mercado similares à que se pretende e um levantamento da lista das funcionalidades mais comuns neste tipo de aplicações. Após isso, são então definidos os requisitos funcionais, ao avaliar cada uma dessas funcionalidades quanto à sua utilidade para o projeto a desenvolver e culminando numa lista filtrada das que devem ser incluídas neste.

No Capítulo 3 começa-se por apresentar o esqueleto da aplicação desejada, apresentando depois as tecnologias que a empresa Real Games usou para a criar. Dessas tecnologias resultará o primeiro protótipo da aplicação desejada, também apresentado neste capítulo, assim como o método a usar para o desenvolver ainda mais e juntar-lhe as funcionalidades

especificadas no Capítulo 2, nomeadamente, através da criação de algoritmos capazes de realizar essas funcionalidades.

No Capítulo 4 são apresentados todos os algoritmos (métodos) desenvolvidos pelo autor, explicando detalhadamente como cada uma das funcionalidades desejadas foi implementada.

No Capítulo 5 são apresentados várias programas responsáveis pelo teste e demonstração dos métodos criados no Capítulo 4 e é desenvolvido um programa mais complexo de paletização para demonstrar as capacidades do RVP, tirando, ao mesmo tempo, conclusões sobre a sua eficiência.

No Capítulo 6 é recordado o trabalho realizado ao longo da dissertação apresentando e justificando as principais conclusões e os benefícios deste projeto para o autor e para a empresa, para além das várias possibilidades de continuação do trabalho desenvolvido no futuro.

2 Definição do problema e do projeto

Os robôs industriais são máquinas de grande versatilidade e amplamente usadas, podendo realizar tarefas como pintura, soldadura, manipulação de objetos, montagem de componentes, entre outras, sendo a sua existência praticamente insubstituível na indústria atual e o seu melhoramento sempre palco de atenção. Consequentemente, a sua apresentação de forma sucinta e o mais cedo possível aos estudantes de diversas áreas é de extrema importância(Pires 2012).

Atualmente, esta apresentação e, de forma similar, a utilização dos robôs em meios industriais pode ser feita de duas formas: através da manipulação do robô real por um operador, complementada com o auxílio de um computador ou de uma consola de comando (normalmente chamada de *Teach Pendant*) capaz de registar e reproduzir os movimentos executados pelo operador ou de ordenar comandos à distância. Ou alternativamente, através de um ambiente virtual capaz de simular o robô desejado e permitir a um utilizador manipulá-lo livremente(Henriques 2002).

Obviamente, destas duas opções, a utilização do robô real será sempre a mais apelativa a um novo utilizador; no entanto ela possui diversas desvantagens relativas à dificuldade de manipular diretamente um robô real, assim como o risco de acidente tanto para o utilizador como para o robô em si, que é obviamente maior durante aplicações de ensino e pode resultar em acidentes e prejuízos monetários. Para além disso, em contexto industrial, a utilização de um robô real obriga a parar a linha de produção, o que pode resultar num elevado prejuízo monetário. No entanto, como foi dito, este é o método mais apelativo de apresentar os conceitos da robótica industrial e, como tal, existem pesquisas para o seu desenvolvimento, assim como, vários fabricantes que fornecem produtos nesta área. Um exemplo destes produtos são os robôs colaborativos fornecidos pela *OMRON*(OMRON 2019).

Os robôs colaborativos começaram a ser desenvolvidos recentemente e são robôs destinados a trabalhar em conjunto com humanos, ao contrário dos robôs tradicionais, que necessitam de isolamento e outras medidas de segurança. Os robôs colaborativos não apresentam movimentos bruscos, nem demasiado rápidos (por exemplo, no caso dos da *OMRON* o utilizador deve informar que áreas do seu corpo podem entrar em contacto com o robô durante o seu funcionamento e, este vai automaticamente limitar a velocidade consoante exista um risco de colidir com uma mão ou em algo mais sensível, como o pescoço, como se pode ver na Figura 2.1, entre outras pequenas características destinadas a reduzir o perigo para o utilizador ao mínimo. Assim, estes são perfeitos para operações industriais que obrigam a mão humana, assim como, são extremamente interessantes para operações de ensino(OMRON 2019).

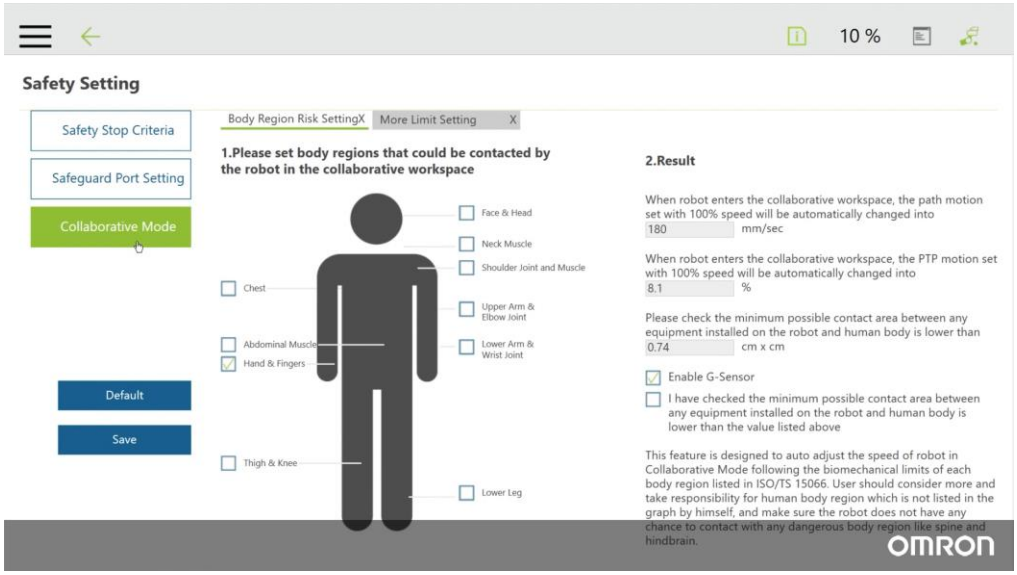

Figura 2.1 - Limitação da velocidade consoante o nível de segurança desejado nos robôs da *OMRON* (OMRON 2019)

No entanto, até estes robôs colaborativos continuam a ser apenas um robô, o que resulta numa outra desvantagem em contexto de ensino: as escolas e universidades geralmente não têm condições, nem recursos, para recriar complexos industriais nos seus laboratórios, pois a criação desses ambientes de treino exige custos elevados, espaço, manutenções regulares e um risco acrescido para os utilizadores inexperientes. Daí, a utilização dos robôs reais em contexto de ensino, apenas pode expor o básico da manipulação de um robô e exemplificar pequenos problemas introdutórios, resultando numa falta de conhecimento de como resolver problemas de maior dimensão e que exijam comunicação entre diferentes máquinas quando os estudantes entram no mercado de trabalho(Couto 2000).

Destes problemas, advém a elevada importância de apresentar estes conceitos através de sistemas virtuais, ou seja, num computador, através de um ambiente gráfico de treino capaz de emular um robô num cenário real, permitindo a sua manipulação sem os problemas enunciados. No entanto, este método também possui algumas desvantagens pois requer uma maior especialização do operador, para além de ferramentas de simulação poderosas, o que é obviamente dispendioso. Ainda assim, as vantagens de segurança, eficiência e até mesmo custo a longo termo, que advém de não necessitar de parar a fábrica e poder simular todo o sistema *à priori*, justificam a tendência para utilizar cada vez mais estes sistemas, assim como, o interesse desta dissertação(Magalhães, Riera e Vigário 2010).

2.1 Projeto a desenvolver

Tendo em conta as vantagens que os sistemas virtuais podem fornecer na área do ensino da robótica industrial, existe, obviamente, um grande interesse da FEUP em realizar um projeto nesta área, o que em conjunto com a experiência da Real Games em aplicações de simulação 3D como o *Factory I/O*(Real Games 2019a), assim como a vontade desta em explorar a área da robótica, resultou na oportunidade de desenvolver um sistema virtual destinado ao ensino da robótica industrial a estudantes.

Assim, o objetivo deste projeto será aproveitar as vantagens dos sistemas virtuais num contexto de ensino, ou seja, fornecer ao utilizador a possibilidade de programar um simulador de um robô através de um sistema virtual de desenvolvimento e teste de programas de robôs, onde o utilizador poderá testar à sua vontade as funções básicas existentes no domínio da robótica, assim como realizar pequenos programas e executá-los num controlador virtual.

Para além disso, este projeto abre também opções de trabalhos futuros de incorporação do robô a desenvolver na aplicação *Factory I/O* da Real Games, onde será possível colocar o robô num ambiente industrial como o da Figura 2.2 e permitir aos estudantes observar a sua operação e comunicação com o resto do sistema, o que torna esta aplicação ainda mais interessante.

Figura 2.2 - Exemplo de simulação de uma fábrica no FactoryIO(Real Games 2019a)

Para tal, de seguida vai-se projetar, conceber e testar o robô virtual programável a desenvolver (a denominar por RVP), começando pela observação de alguns sistemas robóticos virtuais parecidos com o que pretende construir.

2.2 Apreciação dos sistemas robóticos virtuais mais relevantes

Uma das maiores motivações para desenvolver um sistema virtual é criar um ambiente que torne a programação dos robôs fácil. Daí, a definição da interface com o utilizador é de extrema importância. No entanto, a própria definição de sistema virtual consiste em remover os equipamentos físicos, entrando em conflito com o objetivo de tornar a programação fácil: Se esta já é extremamente complicada quando se têm acesso ao robô real, como pode ser mais fácil sem ele?

Esta questão toca a essência do problema da definição de um sistema virtual para robótica, e, com o passar do tempo, os fabricantes destes sistemas perceberam que as linguagem de programação de robôs (RPLs – *robot programming languages*) fornecidas não conseguem ser facilmente utilizadas pela maioria dos utilizadores, por isso a maioria dos sistemas virtuais possuem uma interface com duas camadas: uma interface de programação textual para programadores versados na RPL do robô e uma interface gráfica com várias opções típicas de comando/controlo facilmente percetíveis por qualquer operador, mas com uma flexibilidade de programação muito menor(Craig 2009).

Para além disso, é de notar que ao longo dessa evolução devido a questões de sigilo cada fabricante acabou por desenvolver o seu próprio ambiente e linguagem de programação, normalmente válido apenas para os seus próprios robôs, o que culminou na existência de dezenas de sistemas virtuais e linguagens de robôs disponíveis comercialmente(Henriques 2002).

A dada altura também houve uma tentativa de normalizar estes sistemas, sendo criadas normas para regularizar algumas partes como a linguagem a usar (IRL – *Industrial Robotic Language* na norma DIN 66312)(Anton et al. 2003); no entanto estas normas raramente são respeitadas devido à resistência dos fabricantes a mudar os seus produtos, e, em alguns casos, como o da IRL apresentada acima, nem o seu uso é recomendável, devido à sua complexidade e incompatibilidade com os sistemas existentes.

Assim, ao longo desta sub-secção, irá ser feito um estudo das características de alguns dos sistemas virtuais mais parecidos com a aplicação a construir e usados na área do ensino da robótica industrial, de forma a observar a variedade dos métodos aplicados e posteriormente decidir quais desses métodos implementar no projeto a desenvolver.

2.2.1 A plataforma *RoboDK*

O *RoboDK* é uma aplicação de simulação de robôs industriais, desenvolvida de forma a possibilitar a elaboração de várias simulações sem quase nenhum conhecimento de programação. Entre as suas principais vantagens encontra-se o facto de possuir uma biblioteca com mais de 300 robôs de mais de 30 marcas diferentes, assim como pós-processadores para todos eles (programas capazes de traduzir o código especificado numa linguagem para outra)(RoboDK 2019a), permitindo, se desejado, programar os robôs em linguagens de programação vulgarmente usadas como *Python*, C#, C++, *Visual Basic* e *Matlab*.

Para tal, o *RoboDK* fornece uma interface gráfica extremamente poderosa da qual convém distinguir os quatro grupos de funcionalidades apresentados na Tabela 2.1.

Tabela 2.1 - Grupos de funcionalidades da interface gráfica do *RoboDK*

Funcionalidade	Descrição
Representação do ambiente virtual	Permite a visualização e manipulação dos diferentes componentes do robô e do ambiente de forma realista
Funções de comando	Conjunto de botões que permite comandar o robô diretamente na interface gráfica, como por exemplo definir o tipo de movimento, criar pontos-alvo/trajetórias ou especificar a velocidade/referencial a usar
Representação da sequência de movimentos e ações a realizar	Permite visualizar e editar a sequência de movimentos e ações que se pretende realizar, possibilitando a criação dos programas desejados e a sua execução
Funções de controlo	Conjunto de botões que permite emular algumas funções de controlo de ambiente, que, embora não sejam obrigatórias/necessárias à execução do programa tornam muito mais fácil a manipulação do ambiente, como por exemplo inserir comentários ou mostrar/esconder algum objeto.

Na Figura 2.3 é possível observar a distribuição destes quatro grupos na interface disponibilizada.

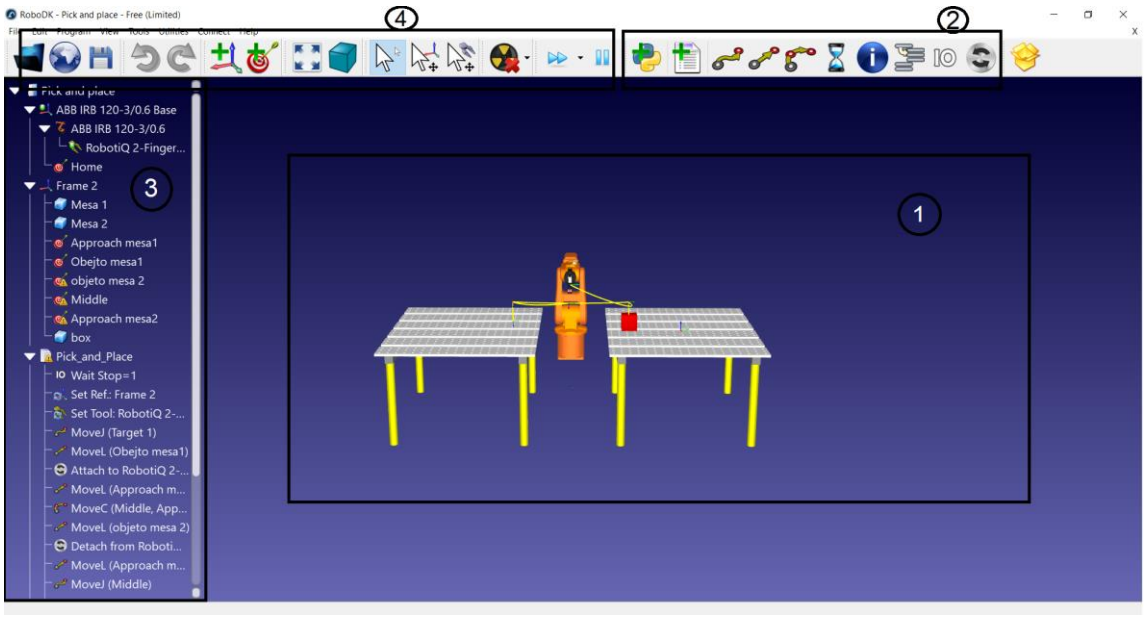

Figura 2.3 - Pontos de interesse da interface gráfica do *RoboDK*
1-Visulação gráfica do robô e ambiente; 2-Funções de comando; 3-Programas/sequência de instruções construída; 4-Funções de controlo

Quanto às funções disponibilizadas em cada uma dessas partes, são demonstradas de seguida ao longo do processo de criação de um qualquer programa.

2.2.1.1 Criação do ambiente e posicionamento dos objetos

O desenvolvimento de qualquer programa começa obviamente pela escolha do robô e dos objetos a usar. Para tal, o *RoboDK* disponibiliza uma biblioteca com mais de 300 robôs, ferramentas e objetos que são mais do que suficientes para aplicações básicas. Estes objetos podem ser facilmente importados para o projeto a construir e após a sua colocação no ambiente podem ser posicionados na posição pretendida através da movimentação com o cursor de cada um dos 6 graus de liberdade do seu referencial ou dos três planos de movimento principais, como se pode ver na Figura 2.4.

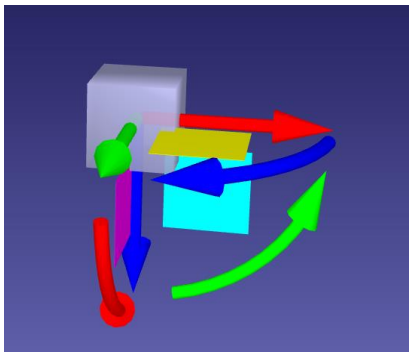

Figura 2.4 - Possibilidades de manipulação dos objetos no *RoboDK*

Alternativamente, este posicionamento pode também ser feito através das coordenadas cartesianas do objeto em relação a qualquer referencial existente. Este posicionamento por coordenadas é, só por si, obviamente interessante, mas, neste caso, convêm realçar que o facto de ele poder ser feito em relação a qualquer referencial torna o posicionamento muito mais fácil em relação a outras aplicações, pois, permite, por exemplo, posicionar um objeto em cima de uma mesa através do referencial da mesa em vez do global. Para além disso, o *RoboDK* também permite copiar/colar as coordenadas de um objeto, o que pode ser útil nestas operações de posicionamento ou durante a programação textual do robô.

No entanto, no *RoboDK* é difícil posicionar os objetos com elevada precisão pois não existem mecanismos adicionais de deteção de colisão ou de relação (distância, paralelismo, entre outros) com outros objetos. Por exemplo, quando se tenta colocar um objeto em cima de uma mesa, é praticamente impossível fazê-lo diretamente no ambiente gráfico pois irá haver sobreposição ou folga e, através das coordenadas é necessário saber as orientações e a largura exata da mesa e do objeto, o que pode não ser conhecido, para além de ser trabalhoso.

2.2.1.2 Desenvolvimento de programas

Para criar um programa é comum começar por especificar os pontos pelos quais o TCP do robô deve passar (*Tool Center Point* – O ponto do elemento terminal do robô onde são definidos todos os movimentos e posições por onde a ferramenta deve passar como se pode ver na Figura 2.5, o objetivo primário do robô é sempre manipular o seu TCP). Para tal, o *RoboDK* permite criar um ponto-alvo na localização atual do robô ou numa superfície qualquer (mas apenas com a precisão manual do operador, o que é pouco eficiente). No entanto, após a criação desses pontos-alvo, este permite alterá-los ao especificar as suas coordenadas em relação a qualquer referencial existente, o que é extremamente útil. Para além disso, o *RoboDK* possui também uma função para colocar o TCP do robô diretamente no referencial de um objeto, permitindo a criação perfeita do ponto-alvo neste, e uma função para observar diferentes combinações de juntas capazes de posicionar o TCP do robô na mesma posição cartesiana.

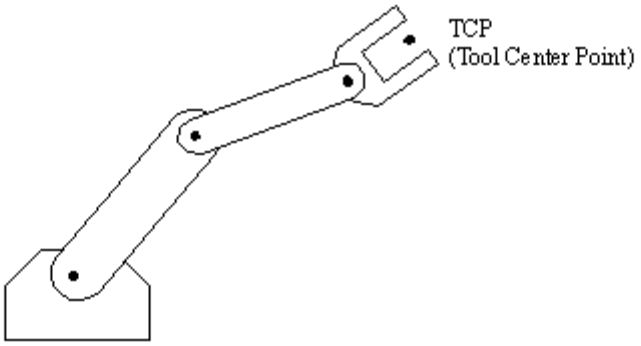

Figura 2.5 - Posicionamento do TCP do Robô(Engineer On A Disk 2010)

De notar a importância da função “alterar a configuração das juntas para uma mesma posição do TCP” do *RoboDK* pois ao definir os pontos-alvo as suas orientações muitas vezes não coincidem ou são demasiado diferentes o que torna impossível o movimento linear entre eles (por exemplo se o eixo dos zz do ponto inicial for oposto ao do ponto final, será necessário realizar uma rotação ao mesmo tempo do movimento, o que pode ser cinematicamente impossível para a trajetória desejada).

Após a definição dos pontos-alvo, segue-se então a especificação do movimento entre eles. No *RoboDK* isto é feito de forma intuitiva através de quatro botões na barra 2 da Figura 2.1: Simplesmente clica-se no segundo símbolo para criar um novo programa e depois escolhe-se o tipo de movimento (de juntas, linear ou circular, apresentados na Figura 2.6) para atingir o alvo em questão um de cada vez. Como dados adicionais pode-se posteriormente acrescentar uma instrução no meio do programa para alterar a velocidade predefinida, o raio de suavização da trajetória, o referencial no qual os alvos estão definidos ou a ferramenta a usar.

Figura 2.6 - Exemplo dos movimentos básicos em robótica industrial, da esquerda para a direita: movimento de juntas, movimento linear e movimento circular(Abreu 2018)

De notar que ao especificar o movimento para o ponto-alvo o *RoboDK* acrescenta automaticamente a instrução de mudança de referencial e, se a localização atual não for um ponto-alvo ele cria automaticamente um ponto-alvo na sua localização. A título de exemplo, é demonstrada na Figura 2.7 uma sequência de instruções no *RoboDK*, com um comando adicional de alteração de velocidade.

Figura 2.7 - Exemplo da estruturação de um programa no *RoboDK*

2.2.1.3 Suporte de eventos e alarmes

A incorporação de eventos/alarmes, assim como de várias funções de controlo, é de extrema importância para adicionar lógica e possibilitar programas mais complexos. Nesta área, o *RoboDK* fornece apenas um pequeno leque de funções básicas, mas que se acredita incluir todas as funções necessárias à maioria dos programas:

- Pausa (podendo escolher o tempo de paragem em milissegundos);
- Apresentar uma mensagem no *teach pendant* do robô;
- Executar um dos programas existentes no projeto;
- Inserir um comentário;
- Criar e escrever uma variável de saída;
- Esperar que uma variável de entrada tenha um determinado valor;
- Unir um objeto ao TCP de uma ferramenta (*attach*) e posteriormente separá-lo (*detach*);
- Mostrar/esconder um objeto ou ferramenta;
- Recolocar um ou mais objetos numa posição guardada
- Detetar colisões.

Dentre estas, convêm denotar a importância das funções de *attach/detach* pois é através destes dois eventos que o robô irá pegar e largar objetos, ou, na prática simular estas ações, pois no *RoboDK* a ferramenta não possui capacidade de movimento. Na prática, é apenas usado o comando *attach* para unir o objeto ao TCP da ferramenta, fazendo com que este se movimente junto do TCP.

Isto pode fazer com que visualmente algumas partes da ferramenta se sobreponham ao objeto o que não é visualmente apelativo, mas pode ser corrigido ao emular também artificialmente os estados da ferramenta através da divisão desta em várias. Por exemplo, se tivermos duas ferramentas: uma correspondente à garra no estado fechado e outra à garra no estado aberto, podemos alterar a visibilidade delas (recorrendo ao evento mostrar/esconder um objeto) ao mesmo tempo que é executado o evento de *attach*, tornando, a garra aberta invisível e a fechada visível, donde resulta a ilusão de que ela efetivamente fechou.

Por outro lado, outra função muito importante em robótica é a deteção de colisões que no *RoboDK* é feita através de um simples mapa de colisões que pode ser alterado a qualquer momento. Este mapa permite especificar que objetos podem colidir, como se vê na Figura 2.8, o que torna fácil a abordagem a este tema, permitindo ao utilizador testar as várias combinações e perceber que algumas colisões devem ser permitidas, por exemplo entre uma ferramenta e o objeto a transportar, enquanto outras devem ser detetadas e provocar a paragem imediata do programa. Alternativamente o *RoboDK* também permite verificar *à priori* se existem colisões numa dada trajetória e caso existam permite criar trajetórias para as evitar automaticamente.

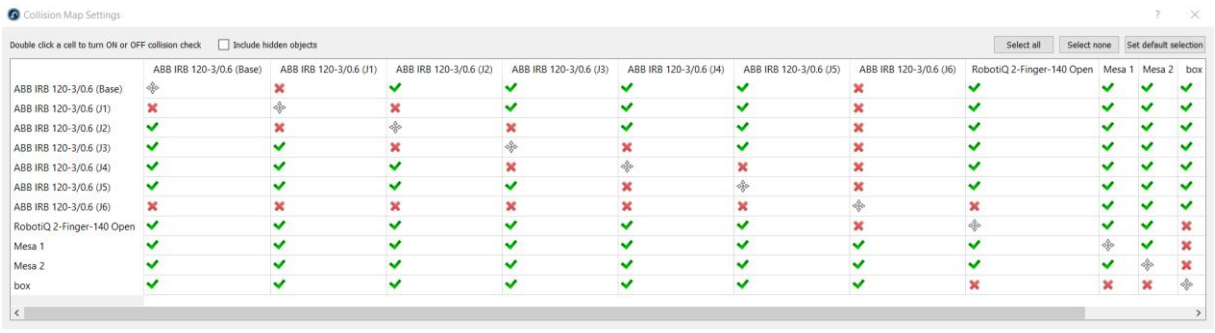

Figura 2.8 - Exemplo de um mapa de colisões no *RoboDK*

Por fim é de realçar também a função de colocar o objeto ou robô numa posição predefinida, o que é útil para definir várias posições de *reset* do sistema.

2.2.2 A plataforma *RobotStudio*

O *RobotStudio* é uma aplicação da *ABB* destinada a modelar, programar e simular células robóticas, construída sobre uma cópia exata do controlador usado pelos robôs da *ABB*: o *ABB VirtualController*, o que permite a realização de simulações extremamente realistas. Para além disso, o *RobotStudio* também possui funções CAD (*Computer-Aided Design* - Desenho Assistido por Computador) e permite a importação de ficheiros criados noutras aplicações de CAD, como por exemplo *SolidWorks*, permitindo a criação de robôs através da definição de relações entre os objetos importados. Isto, em conjunto com a sua biblioteca de objetos e ferramentas, possibilita a criação de várias simulações e ambientes de trabalho ao prazer do utilizador.

Quanto à interface fornecida ao utilizador pode-se identificar facilmente os quatro grupos de funcionalidades da Tabela 2.1, sendo a organização da interface muito parecida com a do *RoboDK*, como se observa na Figura 2.9.

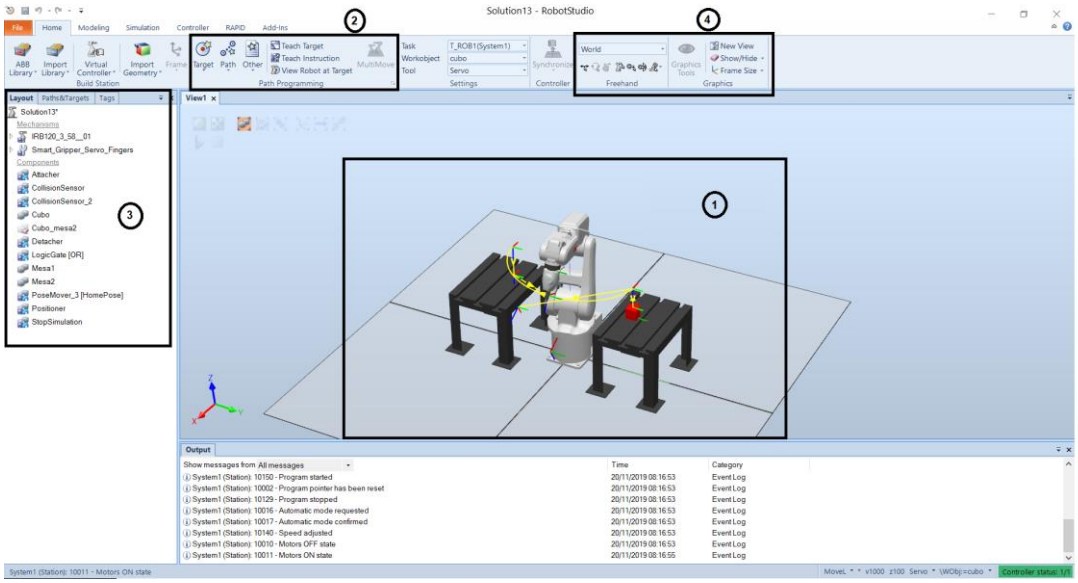

Figura 2.9 - Pontos de interesse da interface gráfica do RoboStudio
1-Visulação gráfica do robô e ambiente; 2-Funções de comando; 3-Programas/sequência de instruções construída; 4-Funções de controlo

2.2.2.1 Criação do ambiente e posicionamento dos objetos

Para criar o ambiente de trabalho, o *RobotStudio* possui, tal como o *RoboDK*, uma biblioteca de robôs, objetos e ferramentas mais do que suficiente para aplicações básicas; no entanto, este também fornece várias funções de CAD, permitindo criar diretamente vários objetos geométricos ou mesmo criar objetos a partir da extrusão de planos, assim como a possibilidade de importar ficheiros CAD de outras aplicações como o SolidWorks, o que permite ao utilizador uma grande liberdade na criação do ambiente.

Por outro lado, quanto ao posicionamento desses objetos, o *RobotStudio* permite, de forma similar ao *RoboDK*, manipular o objeto diretamente no ambiente gráfico (através da movimentação de um dos seus 6 graus de liberdade), especificar as coordenadas cartesianas em relação a um dos 4 referenciais principais (o global, do robô, da tarefa ou do objeto em causa) ou posicioná-lo diretamente num ponto de interesse (canto de um objeto, ponto central, origem de um eixo, interseções, centro de gravidade, entre outros) o que é extremamente útil permitindo o posicionamento de objetos com elevada precisão instantaneamente.

Para além disso o *RobotStudio* também permite alterar o referencial de cada objeto, o que é útil em vários casos: por exemplo, ao colocar um objeto num ponto de interesse, na prática é o centro do seu referencial que é colocado nesse ponto, logo se o ponto de interesse for a superfície de uma mesa e o referencial estiver no centro do objeto haverá sobreposição. Isto pode ser facilmente evitado ao movimentar, temporariamente, o referencial para um dos cantos inferiores. De notar ainda que a alteração do referencial pode também facilitar o posicionamento através das coordenadas cartesianas, pelos mesmos motivos.

2.2.2.2 Desenvolvimento de programas

Para criar os pontos-alvo o *RobotStudio* possui as mesmas funções que para posicionar objetos, ou seja, especificá-los em relação a um dos referenciais principais ou obter um certo ponto de interesse através de vários métodos, o que permite criar todos os pontos desejados com elevada precisão. No entanto, ele possui também as funções de "copiar orientação" e "aplicar orientação" para responder ao problema apresentado na sub-secção 2.2.1.2 de pontos com referenciais demasiado diferentes. Estas funções permitem copiar as orientações de uns pontos-alvo para outros facilmente, o que é muito mais prático e fácil do que apresentar todas as alternativas de conjuntos de juntas existentes.

Por outro lado, para criar o programa no *RobotStudio* apenas é necessário criar uma trajetória (que será o programa), arrastar os pontos alvo para dentro dela e ordená-los pela sequência desejada. Após apenas estas ações, o programa já fica operacional com todos os movimentos definidos como movimentos lineares; no entanto, se desejável eles podem ser editados clicando do lado direito e *edit instruction* para alterar o tipo de movimento, a velocidade, a ferramenta e outros dados mais específicos que já exigem conhecimentos avançados sobre o assunto. Daí, editando esses movimentos acabamos por ficar com algo parecido com o *RoboDK* como se pode ver na Figura 2.10, mas sem poder observar os dados adicionais diretamente, o que pode não ser tão interessante de um ponto de vista académico, mas permite ao programa ser muito mais compacto.

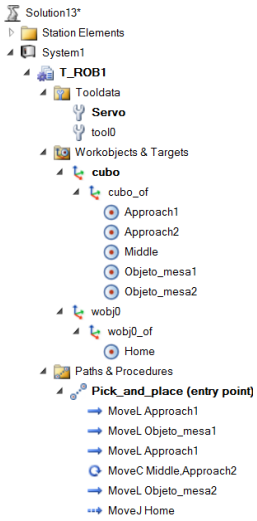

Figura 2.10 - Exemplo da estruturação de um programa no *RobotStudio*

2.2.2.3 Suporte de eventos e alarmes

Quanto à incorporação de eventos/alarmes o *RobotStudio* possui as mesmas funções apresentadas na sub-secção 2.2.1.3 para o *RoboDK* e ainda outras mais avançadas como alterar variáveis analógicas, criar saídas pulsantes e outras que se acredita já não serem interessantes para este estudo. No entanto o método de as incluir no programa é completamente diferente, o *RobotStudio* apenas permite criar diretamente no programa algumas ações (por exemplo alterar/ler variáveis e criar uma pausa) enquanto que os outros eventos, como a realização do *attach/detach* do objeto, devem ser programados num painel à parte.

Nesse painel à parte podem-se juntar vários componentes chamados de *smart components* ao projeto a realizar, sendo que cada um desses componentes representa uma função inteligente ou um evento a realizar no sistema. Para os usar, o utilizador têm então, de começar por escolher quais os *smart components* que pretende através de uma biblioteca que possui dezenas deles (Figura 2.11), entre os quais funções matemáticas, blocos lógicos, funções como as de attach/detach a aplicar à simulação, movimentos do robô, alterar posições de objetos, sensores de colisão, medidores de distância entre objetos, entre muitas outras.

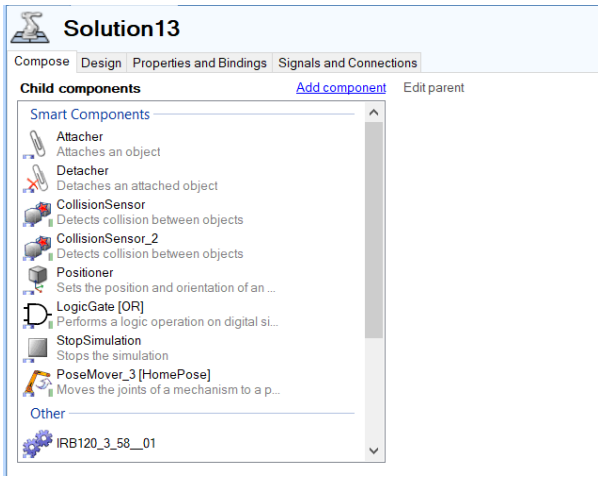

Figura 2.11 - Exemplo de *Smart Components*

Após a escolha dos componentes a usar, o utilizador deve especificar a lógica entre eles através de uma interface interativa que representa cada um desses *smart components* como um bloco que pode ser ligado aos outros consoante a vontade do utilizador e que apenas é executado quando a sua entrada vai a 1 (Figura 2.12). De notar que também se podem incluir variáveis de entrada/saída internas nesta interface ou externas (as criadas no programa do robô) para simular sensores e possibilitar a ligação desta logica ao programa criado.

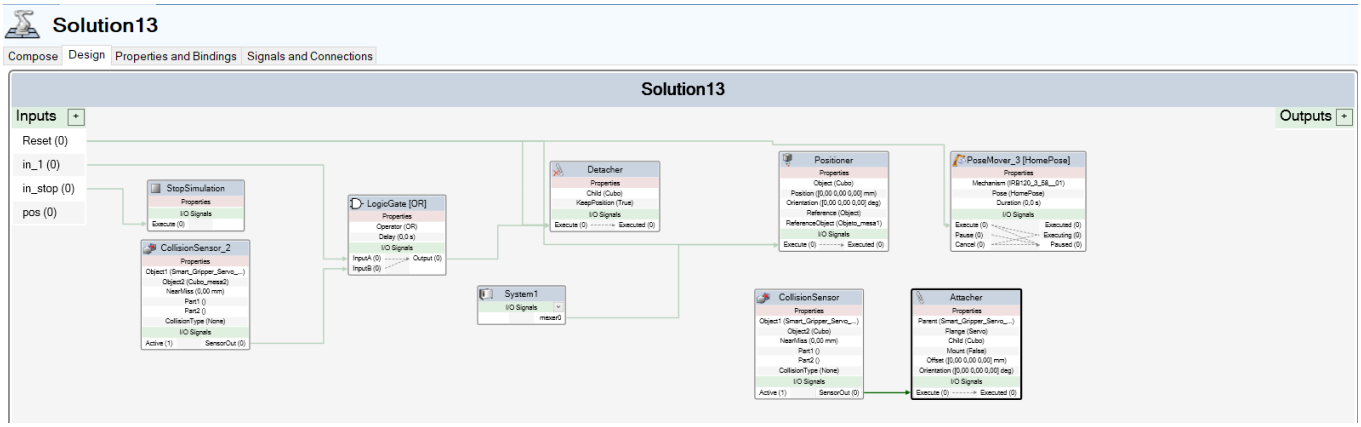

Figura 2.12 - Logica definida entre os *smart components*

Na Figura 2.12 podem-se ver essas variáveis internas nos *inputs* de toda a simulação (chamada de *Solution13*) e uma variável usada no robô (*System1*) chamada mexer0 que irá ativar o *Positioner* responsável por recolocar o sistema na posição inicial no fim da simulação como se pode ver na Figura 2.13, para além disso também se pode ver que foi usada uma ação de chamar um programa para colocar este em *loop*.

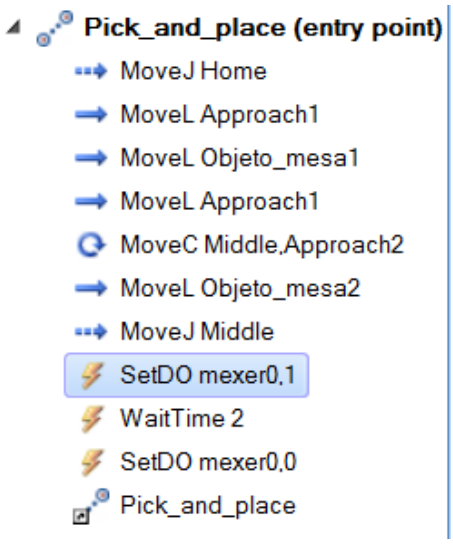

Figura 2.13 - Exemplo da utilização de variáveis analógica para conectar o programa com o painel lógico

Concluindo, pode-se observar que o *RobotStudio* possibilita assim a criação de diversas ações extra possibilitando interrupções e execuções simultâneas de diferentes funções ao passo que no *RoboDK* isso é muito mais difícil ou até mesmo impossível de fazer devido ao leque limitado de funções. No entanto a existência de todas estas funções, junto com as de posicionamento e CAD apresentadas anteriormente, tornam a interface muito mais confusa para um novo utilizador.

Por fim, convêm ainda notar que para correr o programa no *RoboDK* basta executar um duplo clique no seu nome, enquanto no *RobotStudio* é necessário criar um controlador para o robô, sincronizar o ambiente virtual com esse controlador, definir o programa a executar e, por fim, correr a simulação, o que parece desnecessariamente complicado.

2.2.3 A plataforma *Visual Components*

O *Visual Components* é uma aplicação de simulação 3D de ambientes industriais, usado para projetar e simular várias opções de *layouts* para diversos ambientes de produção, aplicações de programação *off-line* e realizar a conexão destes ambientes com sistemas de controlo(Visual Components 2019).

Quanto à parte da robótica, esta é muito parecida com a do *RoboDK* e *RobotStudio*, possuindo também uma biblioteca com mais de 2300 componentes das marcas mais presentes no mercado, funções de CAD e as mesmas opções de posicionamento destes componentes no ambiente. No entanto, quanto à especificação de pontos-alvo, trajetórias e programas no ambiente virtual, o *Visual Components* apresenta uma solução muito mais fácil e inovadora.

Esta aplicação, após uma fase de criação do ambiente e posicionamento dos objetos similar ao *RoboDK* e *RobotStudio*, permite definir diretamente a tarefa a realizar no ambiente, sendo que a criação dos pontos-alvo e da trajetória é feita automaticamente pelo *Visual Components*. Para se compreender melhor este processo de programação, apresenta-se a Figura 2.14, onde se pode observar um ambiente fabril construído no *Visual Componentes*, onde foi colocado um robô capaz de se movimentar ao longo de uma correia, uma plataforma inicial (identificada na figura pelo número 1) que recebe peças de um tapete transportador (cilindros aos quais se chamou “222”) e uma plataforma 2 para onde se pretende transportar as peças. Para executar esta tarefa, o utilizador apenas precisa criar uma tarefa *feed* da peça “222” no robô, e uma tarefa de *need* no painel das propriedades da plataforma 2, como se vê no espaço realçado na Figura 2.14. Tendo estas informações, o robô vai automaticamente procurar pela peça “222”, pegar nela e transportá-la até plataforma 2 onde é necessária. Isto é algo completamente revolucionário, quando comparado com o processo apresentado anteriormente e extremamente útil, mas, no entanto, também é igualmente complicado de programar exigindo o fornecimento de inúmeras informações sobre o ambiente, as peças e as tarefas ao controlador.

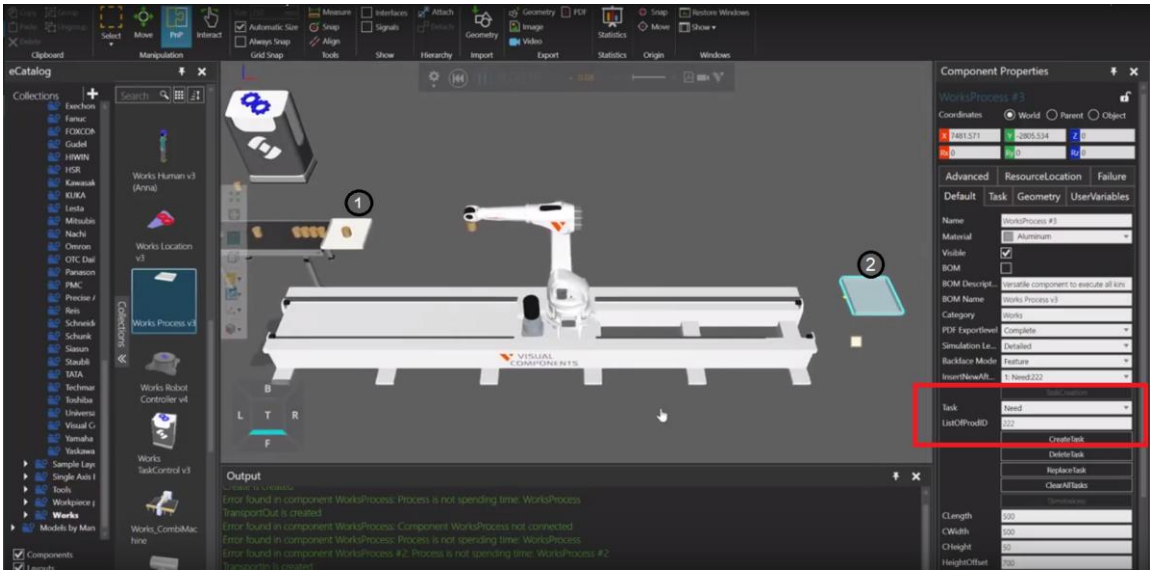

Figura 2.14 - Exemplo de programa no *Visual Components*

Por outro lado, também convêm notar que o *Visual Components* permite a aplicação de lógica a programas (às tarefas criadas) de uma forma fácil usando conhecimentos básicos de programação num dos painéis disponíveis, como se pode ver na Figura 2.15. Neste painel é possível adicionar um leque de ações predefinidas similar às apresentadas para o *RoboDK* (pausa, comentários, alterar variáveis, entre outras), mas, como se pode ver, ele permite também adicionar ciclos lógicos como *if* ou *while* e preencher as lacunas dos ciclos com as ações predefinidas ou programas criados anteriormente, apresentando uma outra forma interessante de programar diretamente na interface virtual.

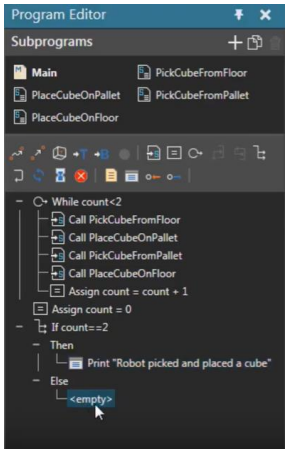

Figura 2.15 - Painel de programação no ambiente gráfico no *Visual Components*

2.3 Análise comparativa da interface gráfica dos programas estudados

De forma a sumariar os conhecimentos obtidos na secção anterior, apresenta-se de seguida a Tabela 2.2 onde se encontram as funções que se consideraram mais necessárias/interessantes entre as observadas nas aplicações *RoboDK*, *RobotStudio* e *Visual Components* (identificadas por R, S e V respetivamente na tabela) e que serão, posteriormente, filtradas de forma a decidir quais incluir no projeto a desenvolver. Desta tabela também se pode concluir que as aplicações possuem funcionalidades extremamente parecidas, sendo a abordagem realizada para resolver a maioria das etapas de construção de um programa a mesma.

No entanto, observaram-se duas lacunas nesta lista de funções que convém referir. Primeiro nenhuma das aplicações possui um motor físico para emular os efeitos do ambiente como, por exemplo, gravidade ou interação entre os objetos, donde resulta que o pegar/largar dos objetos não acontece efetivamente no programa, na prática são apenas usados mecanismos para mover o objeto a transportar ao mesmo tempo do TCP, o que nem sempre parece visualmente correto, para além de ser obviamente mais interessante poder emular as conexões reais.

Tabela 2.2 - Funções encontradas nas aplicações estudadas

Funcionalidades	Funções	Uso
Manipulação de Objetos	-Biblioteca com vários robôs, ferramentas e objetos	R S V
	-Manipulação dos 6 graus de liberdade com o cursor	R S V
	-Manipulação dos planos de movimento com o cursor	R - V
	-Posicionamento por coordenadas	R S V
	-Posicionamento em pontos de interesse	- S V
	-Copiar/colar coordenadas	R S V
	-Alterar posição do referencial de um objeto	- S -
Criação do programa	-Colocar TCP num objeto	R S V
	-Alterar configuração de juntas para mesma posição	R - -
	-Copiar/colar orientação	- S -
	-Movimentos de juntas, linear e circular	R S V
	-Velocidade e raio de suavização da trajetória são variáveis globais	R - -
	-Velocidade e raio de suavização da trajetória fazem parte da definição de cada movimento	- S V
Eventos/alarmes	-Pausa, imprimir mensagem, inserir comentário	R S V
	-Chamar programa	R S V
	-Criar e escrever uma variável de saída, ler entrada	R S V
	-*Attach/detach*	R S -
	-Mostrar/esconder um objeto	R S V
	-Recolocar objetos numa posição guardada	R S V
	-Detetar colisões aquando do movimento e à priori	R S V
	-Programação através de blocos lógicos	- S -
	-*If/Whiles* no painel de execução	- - V
Interface de programação	-Movimentos incrementais, junta-a-junta e de apenas uma junta	R S V

Em segundo lugar nenhuma das aplicações aborda o movimento de juntas assíncrono, todas elas fazem apenas o movimento síncrono, que na prática é melhor pois torna a trajetória do objeto mais uniforme, mas em contexto de aprendizagem também é interessante o assíncrono, que é o movimento mais rápido e fácil de fazer entre dois pontos. A diferença entre os dois encontra-se na velocidade das juntas que no assíncrono é a máxima para todas, fazendo com que as juntas que tem de percorrer uma distância menor acabem o movimento primeiro e insiram uma descontinuidade na trajetória quando o fazem. No síncrono isto é corrigido ao diminuir a velocidade das juntas que percorrem um espaço menor de forma a acabarem todas o movimento ao mesmo tempo como se pode ver na Figura 2.16(Almeida 2016).

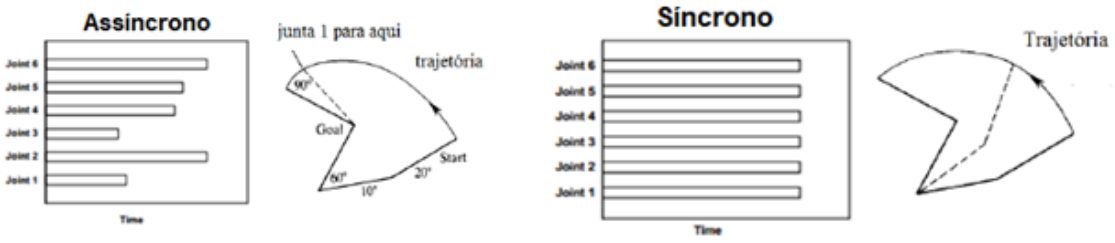

Figura 2.16 - Comparação entre a velocidade e trajetória nos movimentos de juntas síncrono e assíncrono

2.4 Análise comparativa da interface de programação dos programas estudados

Na interface de programação o utilizador usa uma RPL (*Robotic Programming Language*) capaz de permitir ao controlador do robô executar todas as funções existentes na interface gráfica como movimentos, reagir ao ambiente, enviar sinais de saída, entre outras tarefas através de uma sequência de comandos.

Como dito anteriormente a RPL usada varia entre os fabricantes, pois no decorrer da evolução destes sistemas vários fabricantes criaram a sua RPL totalmente original que só funciona nos seus robôs, ou desenvolveram uma biblioteca robótica ao adicionar extensões com comandos usados para a programação de robôs a linguagens previamente existentes e largamente usadas como C++, C# ou *Python*(Lapham 1999). De notar que, nesta segunda hipótese, a utilização dos comandos é muito parecida com a programação tradicionalmente usada nas linguagens de origem e, sendo linguagens comuns, existe uma maior disposição para outros fabricantes aderirem a elas e/ou criarem pós-processadores (programas capazes de traduzir o código especificado numa linguagem para outra)(RoboDK 2019a).

A título de exemplo desta variedade pode-se usar os três programas apresentados em que o *RobotStudio* possui uma linguagem própria inteiramente desenvolvida pela *ABB* chamada *RAPID*, o *Visual Components* usa uma biblioteca criada a partir da linguagem *Python* e o *RoboDK* possui bibliotecas para C++, C#, *Python*, *Visual Basic* e *Matlab* permitindo ao utilizador escrever no estilo de linguagem que está mais familiarizado.

No entanto, convêm denotar que estas linguagens acabam por ser parecidas possuindo quase todas as mesmas funções principais (as apresentadas ao longo da secção anterior), mas com nomes ligeiramente diferentes. Na Figura 2.17 e Figura 2.18 pode-se ver um exemplo de duas destas linguagens: *RAPID* da *ABB* e *Karel* da *Fanuc* para uma aplicação de soldadura, e facilmente se repara que, embora a terminologia seja um pouco diferente, existe claramente uma etapa de definição dos pontos-alvo, seguida da especificação de vários movimentos de juntas/lineares com uma certa velocidade e um certo raio de suavização das arestas (neste caso *fine* – inexistente).

```
%%%
VERSION:1 LANGUAGE:ENGLISH
%%%
MODULE LIVRO_RAPID
VAR robtarget TP1:=[[1144.53,- 0.00,1287.05],[0.70701,0.00000,0.70719,0.00000],[0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget WELD_BAS0GP2:=[[686.71,-100.00,630.08],[0.24991,0.06698,0.93303,-0.25000],[0,- 1,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget WELD_POLGP1:=[[786.71,-100.00,530.08],[0.24993,0.06698,0.93303,-0.24999],[0,- 1,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget TP2:=[[786.71,-100.00,530.08],[0.24994,0.06698,0.93302,-0.24999],[0,- 1,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget WELD_POLGP2:=[[786.71,100.00,530.08],[0.24995,0.06698,0.93302,-0.24999],[0,- 1,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget WELD_POLGP4:=[[986.71,100.00,530.08],[0.24995,-0.93302,0.06698,0.25000],[- 1,0,1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget WELD_POLGP3:=[[886.71,200.00,530.08],[0.00000,0.70707,0.61239,-0.35356],[- 1,0,1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget WELD_POLGP5:=[[986.71,-100.00,530.08],[0.24998,-0.93301,0.06698,0.24999],[- 1,0,1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget TP3:=[[986.71,-100.00,530.08],[0.24999,-0.93301,0.06698,0.24999],[- 1,0,1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
VAR robtarget WELD_BAS0GP3:=[[1086.71,-100.00,630.08],[0.25000,-0.93301,0.06698,0.24999],[- 1,0,1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
!! LANGUAGE RAPID
!! MEMORY 32768
!! TEACHPOINTFILE LIVRO_RAPID.PR#
!! ROBOT IRB1400
PERS tooldata TOCHA:=[TRUE,[[-92.05,- 0.00,189.53],[0.99999,0.00000,0.00000,0.00000]],[0,[0,0,0],[1,0,0,0],0,0,0]];
PROC main()
MoveJ [[1144.53,-0.00,1287.05],[0.70710,0.00000,0.70710,0.00000],[-1,0,- 1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]],\V:=1000,fine\Z:=100,TOCHA;
MoveJ WELD_BAS0GP2,vmax\V:=220,fine\Z:=0,TOCHA; MoveL WELD_POLGP1,vmax\V:=220,fine\Z:=0,TOCHA;
!! ARCWELDON 100,10
MoveL TP2,vmax\V:=220,fine\Z:=0,TOCHA;
MoveL WELD_POLGP2,vmax\V:=220,fine\Z:=200,TOCHA;
MoveC WELD_POLGP3,WELD_POLGP4,vmax\V:=220,fine\Z:=200,TOCHA;
MoveL WELD_POLGP5,vmax\V:=220,fine\Z:=0,TOCHA;
!! ARCWELDOFF
MoveL TP3,vmax\V:=220,fine\Z:=0,TOCHA;
MoveL WELD_BAS0GP3,vmax\V:=220,fine\Z:=0,TOCHA;
MoveJ [[1144.53,-0.00,1287.05],[0.70710,0.00000,0.70710,0.00000],[-1,0,- 1,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]],\V:=1000,fine\ Z:=100,TOCHA;
ENDPROC ENDMODULE
```

Figura 2.17 - Exemplo de programação de uma operação de soldadura na linguagem RAPID(Henriques 2002)

```
1   PROGRAM LIVRO_KAREL
2   -- ! LANGUAGE KAREL 2
3   -- ! MEMORY 8192
4   -- ! ROBOT IRB1400
5   -- TEACHPOINT DECLARATIONS VAR
6   WELD_BAS0GP3: POSITION WELD_POLGP5: POSITION WELD_POLGP4: POSITION WELD_POLGP3: POSITION WELD_POLGP2: POSITION WELD_POLGP1: POSITION WELD_BAS0GP2: POSITION TP1 : POSITION
7
8   BEGIN
9   $UTOOL=POS(154.8749,-0,67.6648,90,0,90,'')
10  $USEMAXACCEL=TRUE
11  %INCLUDE LIVRO# WITH $MOTYPE=JOINT
12  MOVE TO $HOME:$UTOOL
13  WITH $MOTYPE=Joint, $TERMTYPE=FINE, $SPEED=100 MOVE TO WELD_BAS0GP2
14  WITH $MOTYPE=Joint, $TERMTYPE=FINE, $SPEED=100 MOVE TO WELD_POLGP1
15  -- ! ARCWELDON 100.0,30
16  WITH $MOTYPE=Linear, $TERMTYPE=FINE, $SPEED=100 MOVE TO WELD_POLGP2
17  WITH $MOTYPE=Linear, $TERMTYPE=FINE, $SPEED=100 MOVE TO WELD_POLGP3
18  WITH $MOTYPE=Linear, $TERMTYPE=FINE, $SPEED=100 MOVE TO WELD_POLGP4
19  WITH $MOTYPE=Linear, $TERMTYPE=FINE, $SPEED=100 MOVE TO WELD_POLGP5
20  -- ! ARCWELDOFF
21  WITH $MOTYPE=Linear, $TERMTYPE=FINE, $SPEED=100 MOVE TO WELD_BAS0GP3
22  WITH $MOTYPE=JOINT MOVE TO $HOME:$UTOOL
23  END LIVRO_KAREL
```

Figura 2.18 - Exemplo de programação de uma operação de soldadura na linguagem Karel(Henriques 2002)

Para além disso as RPLs possuem normalmente outras duas características. Primeiro, são linguagens interpretadas, ou seja, não é necessário compilar o programa, as instruções são traduzidas para linguagem máquina aquando da sua execução. Isto advém do facto da programação de robôs tender a ser interativa e aperfeiçoada por tentativa e erro exigindo a repetição do ciclo "editar-compilar-correr", normal noutros tipos de linguagens, inúmeras vezes, o que é aborrecido para o utilizador e pode ser uma perda de tempo. Para além disso, sendo interpretada, é possível executar apenas parte das instruções de um programa sozinha e a qualquer momento durante o desenvolvimento ou depuração do programa. No entanto, isto é sempre uma decisão do fabricante por isso algumas linguagens, como a Karel, continuam a ser compiladas(Craig 2009).

Em segundo lugar, as RPLs devem possuir as variáveis básicas necessárias à modelação de um espaço robótico virtual, apresentadas na Tabela 2.3. Estas variáveis são de extrema importância pois são as bases para representar um ambiente tridimensional e manipular textualmente pontos-alvo e movimentos, fornecendo a capacidade de mover objetos ao robô, ou seja, a sua função principal(Craig 2009).

Tabela 2.3 - Variáveis necessárias à modelação de um espaço robótico virtual

Variável	Significado
Conjuntos de posições de juntas	Representação da posição alvo pela combinação da deslocação das juntas do robô
Pontos-alvo	Representação da posição alvo em coordenadas cartesianas
Orientações	Representação da orientação desejada em função dos ângulos de rotação em relação a um referencial
Referenciais	Facilita a especificação de pontos e permite a alteração da posição de vários pontos-alvo ao mesmo tempo quando o seu referencial é alterado (por exemplo mover todos os pontos especificados numa mesa quando esta é movimentada)

Por fim, as RPLs também costumam suportar, para além das funções apresentadas na interface gráfica, várias outras funções parecidas, mas mais especificas para certas situações resultando em muitas funções extra com pequenas diferenças entre si, por exemplo o *RobotStudio* possui mais de 5 funções destinadas a alteração de referenciais, que variam consoante se pretende obter os dados em quartiniões, ângulos de Euler ou num objeto *frame*. Este tipo de funções pode ser interessante em algumas situações, mas muitas delas acabam por quase nunca ser usadas pois essas situações especificas são raras e solúveis com as funções principais a que o utilizador está mais habituado. No entanto, três funções deste tipo recorrentes e importantes para o caso em estudo são a possibilidade de movimentar apenas uma das seis juntas do robô, observar a movimentação sequencial das juntas ou realizar movimentos incrementais no robô, que embora não sejam movimentos tão importantes, nem tão necessários em aplicações reais como os três principais, são interessantes num contexto de aprendizagem.

2.5 Requisitos funcionais do projeto a desenvolver

Tendo em mente as várias características das aplicações existentes no mercado e o objetivo a atingir no robô Real Games, segue-se a seleção de quais as características a incluir no projeto a desenvolver. Assim, esta secção apresenta e justifica as funções/funcionalidades mais importantes das aplicações existentes, assim como uma primeira planificação do RVP a construir.

Nesse sentido, primeiramente decidiu-se que o ambiente devia ser o mais simples possível, visto ser destinado a aplicações de aprendizagem, e como tal, devia evitar funcionalidades e detalhes desnecessários que apenas subcarregam o utilizador. Daí, ignora-se já a partida a inclusão de funcionalidades CAD e bibliotecas de objetos, assim como a presença de vários robôs. A aplicação a desenvolver deverá possuir apenas um único robô genérico capaz de apresentar todos os conceitos desejados. Da mesma forma, vários outros temas ignorados na maioria das aplicações existentes no mercado, como pós-processadores e controladores, serão também ignorados aqui.

Assim, o robô a usar não precisa de respeitar grandes requisitos, bastando ser, no mínimo, um robô de 5 eixos, de forma a representar os robôs mais comuns usados nesta área, normalmente de 5 ou 6 eixos. Para além disso, definiu-se também que este devia ser suficientemente grande para conseguir realizar operações de paletização entre tapetes transportadores e possuir uma

capacidade de carga capaz de carregar os componentes existentes no *Factory I/O* (3-20 Kg), planeando já futuras interações com esse ambiente.

De seguida, para ser possível ao utilizador comandar o robô este tem de possuir um certo número de funções capazes de emular movimentos. Por isso, definiu-se que este seria, obviamente, um dos tópicos principais a abordar, sendo que a aplicação deve ser capaz de apresentar sucintamente os principais tipos de movimentos existentes e as características de cada um. Entres estes estão obviamente os três movimentos principais (de juntas, linear e circular), mas também a possibilidades de manipular juntas individualmente e os movimentos incrementais, devido ao seu interesse extra para aplicações de aprendizagem. Para além disso, também é interessante fazer a distinção entre o movimento de juntas síncrono e assíncrono pelo mesmo motivo, mas, quanto ao movimento paramétrico, considera-se que ele é demasiado complexo e desnecessário para esta aplicação.

O utilizador também deve ter controlo sobre a velocidade e conseguir suavizar os pontos de descontinuidade(arestas) da trajetória em cada um desses movimentos. Para tal, decidiu-se adotar o método de *RoboDK* em que ambos são tratados como variáveis globais e cada movimento adapta-se aos valores predefinidos. Na Tabela 2.4 apresenta-se um resumo de todas as funções a implementar relativas aos movimentos.

Tabela 2.4 - Movimentos a incluir no projeto

Funcionalidades	Importância
Movimento de juntas	Movimento mais rápido e fácil de realizar entre dois pontos, pode ser assíncrono se todas as juntas se movimentarem à mesma velocidade ou síncrono se todas durarem o mesmo tempo a realizar o movimento.
Movimento linear	O robô realiza uma trajetória retilínea entre o ponto inicial e final
Movimento circular	O robô realiza um círculo entre o ponto inicial e final passando num certo ponto intermédio
Controlo da velocidade dos movimentos	É um requisito extremamente importante em muitas aplicações, também é interessante poder controlar o perfil de velocidade ao longo do movimento; no entanto, devido às alterações de perfil de velocidade serem difíceis de implementar decidiu-se implementar apenas o mecanismo de velocidade constante.
Suavização de descontinuidades da velocidade	Permite evitar acelerações/desacelerações consecutivas assim como tornar o movimento mais rápido
Movimentos junta-a-junta e incrementais	Movimentação de apenas uma das juntas de cada vez, movimentação individual das juntas e movimento relativo das juntas/TCP. Não possuem grande interesse industrial, mas são uteis num contexto de ensino.
Movimento paramétrico	Realiza um movimento segundo uma equação paramétrica. Elevada complexidade de implementação e raramente usado, não justificando o trabalho necessário à sua implementação

Ainda no contexto de movimento, decidiu-se também acrescentar mais duas funções que parecem uteis para contextos didáticos e de depuração. Uma função que regista o trajeto do TCP do robô e outra que é capaz de desenhar os movimentos a realizar à priori (desenhar linhas na interface gráfica), o que permite avaliar a precisão do robô pela comparação das duas.

Após o robô mover-se para a posição desejada, ele necessita, obrigatoriamente, de executar alguma ação nessa posição, sendo a ferramenta e as suas capacidades de manipulação outro tópico importante a abordar. No entanto, sendo esta aplicação apenas algo introdutório aos conceitos da robótica industrial decidiu-se, por questões de tempo e volume de trabalho emular apenas uma ferramenta, tal como no caso dos robôs, o que é suficiente para perceber os conceitos associados à manipulação geral de ferramentas. No entanto os TCPs são algo a ser estudado claramente devido à sua elevada importância e como tal será útil permitir a definição de vários TCPs, assim como a sua manipulação e alteração.

Tendo este robô funcional (capaz de se movimentar e realizar operações), deve-se ter em conta a sua comunicação com outras máquinas, algo extremamente necessário não só em robótica, mas em qualquer contexto industrial. Daí o projeto a desenvolver deve também possuir a capacidade de emular eventos e alarmes que permitam essa comunicação.

Pretendendo esta aplicação ser algo introdutório e simples, apenas se acha necessário incorporar variáveis binárias e, se tal for possível, mostrá-las e permitir ao utilizador alterá-las no modo de execução, deixando a incorporação de variáveis analógicas e de supervisão para desenvolvimentos futuros. Quanto aos alarmes, e mais precisamente às colisões, é de realçar a sua extrema importância na robótica industrial. No entanto, devido à complexidade exigida na sua implementação, assim como por causa do tempo limitado para desenvolver a aplicação decidiu-se não as aplicar.

Ainda no tópico dos eventos, o robô também deve possuir as funções de pausa, comentários, *print*, *multi-thread* e chamar programas. Estas funções são extremamente uteis na programação das tarefas, assim como, na sua depuração.

Estando o robô definido e capaz de comunicar com o ambiente é necessário criar também esse ambiente, assim como a interface de comunicação com o utilizador. Esta deve ser dividida em duas partes: a interface gráfica e a de programação. Começando pela de programação devem ser criados um IDE (*Integrated Development Environment*) de edição textual e uma RPL capaz de especificar todas as funções necessárias, que, por uma questão de simplicidade tanto para o autor como para um futuro utilizador do programa, devem ser desenvolvidos a partir de uma linguagem existente em vez de algo inteiramente original. Esta obviamente deverá possuir todas as variáveis de modelação de ambientes robóticos presentes na Tabela 2.3 (pontos-alvo, referenciais, orientações e conjuntos de posições de juntas).

Por fim quanto a interface gráfica a fornecer ao utilizador onde normalmente estão as funções de comando e controlo, considera-se que numa primeira fase de desenvolvimento da aplicação é desnecessária a criação dos botões responsáveis pelos comandos de programação pois, na prática, todos estes comandos são primeiramente especificados através da programação textual, onde podem ser diretamente usados e não é lógico criar os botões antes das funções que eles devem ativar. No futuro, após possuir todas as funções completamente definidas e testadas talvez se decidia efetuar este aperfeiçoamento. Por outro lado, quanto às funções de controlo, estas devem ser apenas as mínimas necessárias pois, apesar de serem um pouco interessantes, a maioria delas, como por exemplo as funções de controlo da visibilidade dos componentes, de observação do espaço de trabalho e de calculo de configurações de juntas alternativas, não justificam o esforço e tempo necessários à sua implementação, enquanto outras como as de CAD adicionam novos níveis de complexidade que não se pretendem numa aplicação de introdução ao tema. Daí pretende-se apenas incluir a capacidade de mostrar a posição atual do TCP em coordenadas cartesianas e das suas juntas, assim como permitir alterá-las diretamente na interface gráfica.

Para além disso e da mesma forma, os métodos de programação lógica usados diretamente na interface gráfica como a programação de blocos ou a listagem e edição da sequência de ações a realizar, também devem ser realizados apenas em versões futuras. Sendo que nesta primeira versão, o objetivo é permitir à interface gráfica apenas apresentar o robô, enquanto a programação deve ser realizada numa interface textual que possua todas as funções necessárias à sua manipulação.

A Tabela 2.5 sumaria as funcionalidades a incluir no projeto.

Tabela 2.5 - Resumo das funcionalidades/funções a incluir no projeto

Funcionalidade	Funções a Implementar
Especificação de movimentos	-Movimento de juntas síncrono/assíncrono, linear, circular, de apenas uma junta, junta-a-junta e incremental -Controlo da velocidade e suavização das descontinuidades da trajetória
Incorporação de ferramentas/TCPs	-Apenas 1 ferramenta simples -Definir, mover e alterar vários TCPs
Interação com o ambiente	-Apenas variáveis binárias -Funções de pausa, comentários, *print*, *multi-thread* e capacidade de chamar outros programas
Definição da RPL	-Criar uma biblioteca para uma linguagem existente -Especificar pontos-alvo, referenciais, orientações e conjuntos de posições de juntas
Definição da interface gráfica	-Visualização e edição da posição atual do TCP -Visualização e edição do estado das juntas -Manipulação da juntas e TCP graficamente

2.6 Síntese

Este capítulo apresentou o problema da introdução da robótica industrial a estudantes, apresentando como solução os sistemas virtuais, após a avaliação das suas vantagens em relação aos sistemas reais. Daí, decidiu-se que o projeto a desenvolver seria a construção de um sistema virtual onde os estudantes podem operar um robô, criar programas para ele e testar as diversas funções do domínio da robótica industrial.

No entanto já existem várias aplicações de sistemas virtuais de robótica no mercado por isso o desenvolvimento deste projeto começou obrigatoriamente pelo estudo desses sistemas. Assim, apresentou-se as características existentes em três das aplicações mais usadas nesta área: *RoboDK*, *RobotStudio* e *Visual Components* donde se pode concluir sobre as estratégias de construção de programas para robótica usadas (através da criação de pontos-alvo, trajetórias e eventos no *RoboDK/RobotStudio* e através da definição de tarefas no *Visual Components*),

para além de obter a lista das principais funções necessárias para desenvolver cada parte do programa.

Tendo observado os diferentes métodos de desenvolver os programas, assim como as várias opções de funções a incluir pode-se então especificar uma primeira lista de requisitos para o projeto a desenvolver, avaliando quais dessas funções seriam mais importantes num contexto de aprendizagem e o mínimo de complexidade necessário para tornar o nosso projeto competitivo neste mercado.

Daqui, resultaram cinco requisitos funcionais a cumprir: executar movimentos que permitam ao robô chegar aos pontos-alvo; incorporar ferramentas e usá-las de forma a realizar as mais diversificadas tarefas que são requeridas nesses pontos-alvo; comunicar com o exterior para saber quando operar ou não e conseguir responder prontamente a alterações do ambiente; definir uma linguagem de programação que seja capaz de comandar o robô e implementar as funções necessárias para a correta implementação dos três temas anteriores. E, por fim, definir uma interface gráfica capaz de permitir ao utilizador interagir com o robô. Para além disso também se apresentou o mínimo de funções necessárias para tornar cada um destes requisitos funcionais, obtendo uma primeira ficha técnica das funções a implementar pelo autor.

No capítulo seguinte, serão estudados os recursos de engenharia a aplicar pela empresa para tornar possível a implementação destas funções, assim como, para criar o sistema virtual em si.

3 Aspetos de engenharia subjacentes ao projeto

Delineadas no Capítulo 2 as funcionalidades a atingir no presente projeto, é chegada a altura de as definir em concreto e traçar o caminho para as obter.

Para tal, e à imagem do que se tem noutros ambientes de programação de robôs virtuais, decidiu-se que a aplicação RVP a atingir deverá consistir num ambiente virtual onde coexistem o robô virtual e quatro painéis, correspondendo estes a:

- O painel "*Robot*" – um sinótico de leitura e escrita de variáveis de trabalho e do estado atual do robô;
- O painel "*Scripts*" – para listagem e seleção de programas pré-desenvolvidos e, portanto, em condições de ser executados pelo robô;
- O painel "*Console*" – uma janela de comandos e mensagens para troca de informações com o utilizador – como erros e o início/fim de programas.
- O painel "*MODBUS*" - para troca de sinais de I/O entre o ambiente virtual e o exterior

Também se decidiu que, pelo menos nesta primeira fase, o ambiente RVP não incluiria meios de desenvolvimento de programas do robô. Em alternativa, esses desenvolvimentos serão feitos num editor de texto externo e à escolha do utilizador, sendo os ficheiros daí resultantes transferidos depois para o diretório de programas acessíveis através do painel "*Script*". Sendo também que a escrita de programas por parte do utilizador final terá de basear-se em instruções reconhecidas e executáveis pelo robô, é essencial disponibilizar ao programador o conjunto de instruções de programação aceitáveis e as regras (semânticas e outras) a que têm de obedecer. Esta informação é disponibilizada na forma de uma biblioteca devidamente comentada acessível através da aplicação RVP. Os conceitos enunciados estão ilustrados na Figura 3.1.

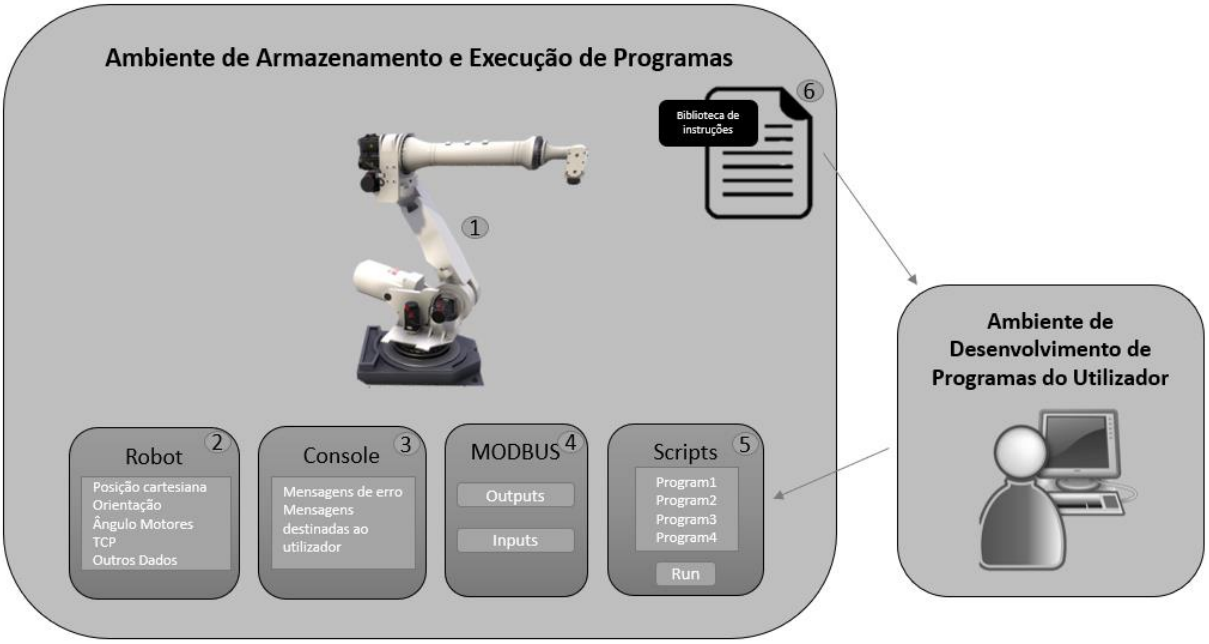

Figura 3.1 - Componentes do ambiente destinado ao utilizador final
1 - Robô Virtual, 2 - Painel "Robot", 3 – Painel "Console", 4 – Painel "MODBUS", 5 – Painel "Scripts", 6 – Biblioteca de instruções

Assim, coube à empresa Real Games desenhar e criar o embrião daquela que será a futura aplicação RVP. Nesta forma ainda básica, a aplicação RVP inclui:

- O robô virtual propriamente dito e a interface gráfica que o representa,
- O seu controlador embebido (que não tem uma representação visual e cujos detalhes de implementação passam despercebidos ao utilizador final)
- As quatro janelas antes referidas
- Uma primeira biblioteca com os métodos base necessários ao comando do robô na dita interface.

A missão do autor é expandir esta biblioteca através da criação de novos métodos capazes de fornecer ao utilizador final as funcionalidades enunciadas no Capítulo 2.

A Real Games interveio, portanto, ativamente nas decisões sobre as funcionalidades a usar, desenvolvendo umas e especificando outras. Daí, ser importante apresentar neste capítulo as tecnologias usadas para a criação do ambiente final desejado.

3.1 Definição e suporte tecnológico da aplicação a criar

Para criar um sistema virtual capaz de representar da forma mais realista possível um robô e o seu ambiente de funcionamento, a Real Games decidiu criar três modelos ou partes que representam, de certa forma, as etapas necessárias à emulação do robô RVP: um modelo gráfico capaz de apresentar o robô num ambiente 3D, um motor físico capaz de executar movimentos segundo as leis da física reais, e um algoritmo capaz de decidir quais os movimentos das juntas a fazer consoante o movimento desejado no TPC, como se pode ver na Figura 3.2.

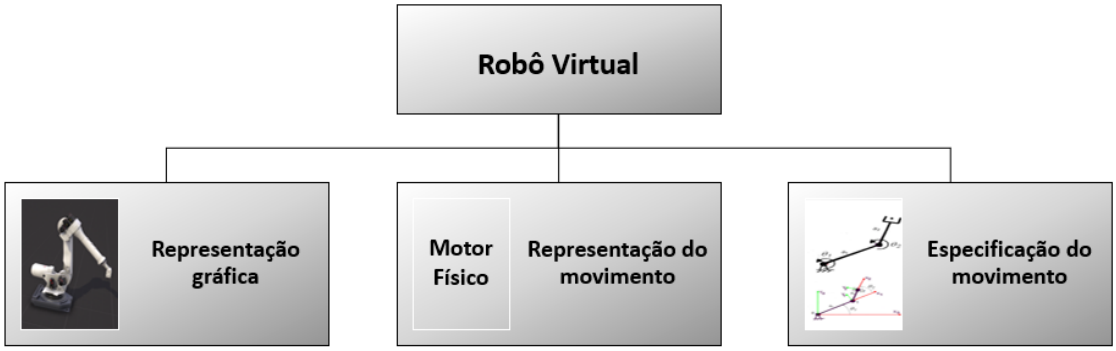

Figura 3.2 - Componentes necessários ao desenvolvimento de um robô virtual

Estes três modelos obviamente complementam-se resultando numa representação gráfica de um robô programável de grande qualidade e altamente realista, mas resultam também em três problemas que precisam de ser abordados separadamente como se descreve de seguida.

3.1.1 Modelo gráfico

Tendo uma ideia do tipo de robô a construir e das funcionalidades a incluir começou-se então pelo desenvolvimento do ambiente gráfico, sendo para tal usado o *software Unity 3D* pela Real Games.

Este software foi escolhido devido à Real Games já possuir bastante experiência com ele e por ser um dos melhores do mercado na simulação de ambientes virtuais, possuindo várias ferramentas de animação, design, construção de objetos, um motor de renderização em tempo real, APIs gráficas, entre várias outras funcionalidades(Unity Technologies 2019).

Definido o ambiente gráfico de desenvolvimento a usar, decidiu-se depois o robô a incluir. Para esta escolha procurou-se um robô usualmente usado na indústria que tivesse no mínimo cinco eixos, um espaço de trabalho suficientemente grande para conseguir realizar operações de paletização entre tapetes transportadores e uma capacidade de carga capaz de carregar os componentes existentes no *Factory I/O* (3-20 Kg). Daí, escolheu-se emular o robô R-2000iB/100H da *Fanuc*, que possui 5 eixos, um erro de repetibilidade de ±0,2 mm e o espaço de trabalho apresentado na Figura 3.3.

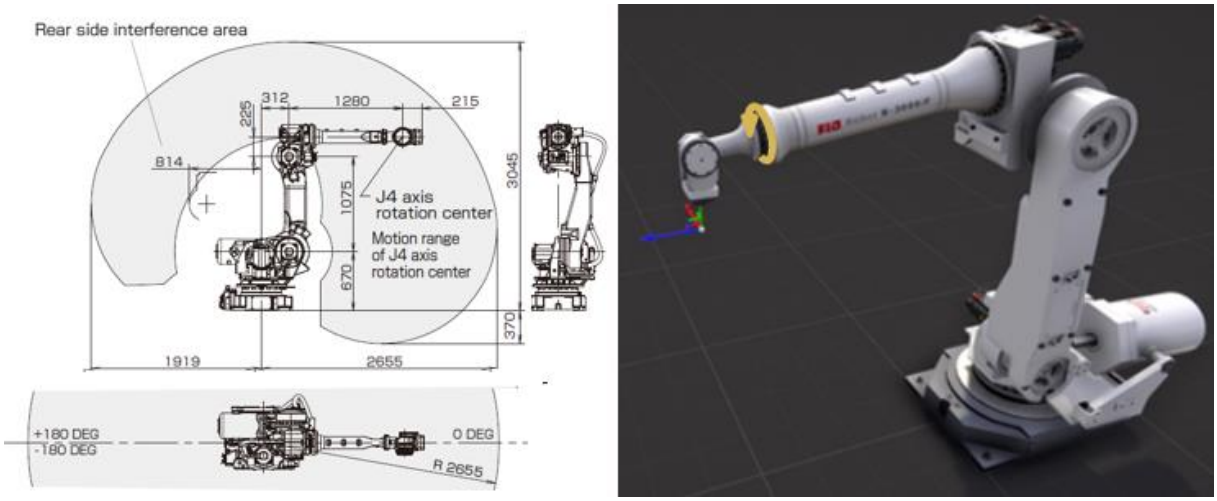

Figura 3.3 - Dimensões e representação do robô R-2000iB/100H da Fanuc no RVP

Quanto às capacidades de movimento do robô emulado, elas também são idênticas à do robô real, existindo apenas variâncias mínimas, como se vê na Tabela 3.1. No entanto, convêm reparar que foi adicionada a junta e motor representados a amarelo na Figura 3.3, de forma a adicionar o grau de liberdade que faltava e torná-lo um robô de 6 eixos.

Tabela 3.1 - Amplitude de movimento e velocidade de cada junta do robô

Junta	Amplitude de movimento do robô real	Amplitude de movimento do robô emulado	Velocidade de ambos os robôs
Junta 1	360 º	320 º	130 º/s
Junta 2	136 º	125 º	130 º/s
Junta 3	362 º	315 º	130 º/s
Junta 4	------	340 º	170 º/s
Junta 5	250 º	250 º	360 º/s
Junta 6	720 º	380 º	360 º/s

3.1.2 Controlador do robô

O controlador do robô é o componente responsável por conduzir os movimentos do robô. Daí, embora normalmente possua capacidades como armazenar a sequência de instruções, comunicar com outros equipamentos ou recolher dados do ambiente, a sua função fundamental é coordenar cada uma das juntas do robô de forma a realizar as operações/movimentos para o qual foi programado, assim como garantir a precisão destes(Couto 2000).

Para tal o controlador deve ser capaz de saber a cada momento o estado atual das juntas e o estado final desejado para as mesmas. Destes estados, o inicial pode ser facilmente obtido a partir de transdutores de posição, mas o final nem sempre é tão fácil, pois o objetivo do robô é colocar o TCP numa certa posição final, não as juntas, e essa posição final do TCP pode ser fornecida de várias maneiras. Daí, a existência da área de estudo da cinemática - estudo analítico da geometria e do movimento de um robô em relação a um sistema de coordenadas de referência e em função do tempo.

De forma simples, a cinemática pode ser dividida em duas vertentes, a cinemática direta responsável por transformar uma combinação das juntas numa posição do TCP (expressa num sistema de coordenadas), e a cinemática inversa responsável pelo processo inverso: transformar a posição do TCP fornecida em coordenadas cartesianas/esféricas numa combinação de juntas(Pires 2012) como se pode ver na Figura 3.4.

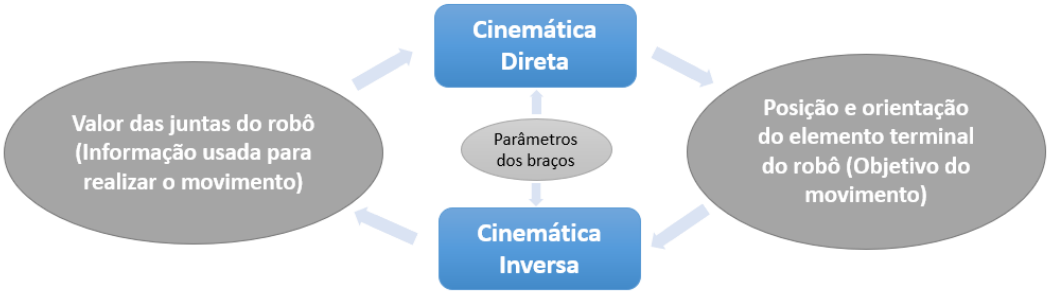

Figura 3.4 - Relação entre cinemática direta e inversa

De notar que em termos de cálculos a cinemática direta será relativamente fácil pois basta somar/multiplicar os ângulos e comprimento de cada junta para obter a posição final; no entanto o processo inverso é muito mais difícil exigindo algoritmos complexos.

3.1.2.1 Motor físico

Começando a definição do controlador pelo nível mais baixo (cinemática direta), decidiu-se utilizar um motor físico – *software* capaz de simular as leis da física num ambiente 3D usando variáveis como massa, velocidade e força, tornando possível simular condições reais.

Assim escolheu-se usar uma variante do motor físico PhysX da Nvidia para Unity3D, que suporta a emulação de articulações, permitindo uma simulação fiável das juntas do robô e realizar movimentos de cinemática direta como o que se vê na Figura 3.5(NVIDIA 2019).

Figura 3.5 - Movimento do 3º motor do robô

3.1.2.2 Modelo de cinemática inversa

Estando o mecanismo de cinemática direta funcional, pôde-se avançar para o nível mais sofisticado de controlo: a cinemática inversa, ou seja, a criação de modelos matemáticos capazes de calcular a posição das juntas desejadas usando apenas a especificação da posição num eixo cartesiano.

Para tal foi utilizado o algoritmo *cyclic coordinate descent*, que, em comparação com os outros métodos geralmente usados para resolver o problema da cinemática inversa, como a inversão da matriz Jacobiana, é muito mais fácil de implementar e exige uma capacidade de calculo muito menor, mas também produz resultados com uma menor precisão(Kenwright 2012; Mishra e Meruvia-Pastor 2014).

Para além disso, devido à forma como este método foi aplicado também convêm denotar que quando se move o robô de uma posição cartesiana para outra, não é possível saber o conjunto de juntas da posição final do movimento antes de o realizar. Isto vai complicar a realização de vários movimentos complexos como o linear e circular, e tornar impossível a realização do movimento de juntas síncrono, como se verá no Capítulo 4.

3.2 Aplicação RVP construída

Tendo o robô virtual completamente definido e operável, foi altura de criar os painéis enunciados anteriormente, nomeadamente, um painel de visualização de dados, um diretório de programas e uma janela de comunicação, através das funcionalidades fornecidas pelo *Unity 3D*.

De notar que, nesta etapa de desenvolvimento do RVP, decidiu-se também criar um quarto painel temporário, destinado a mostrar as entradas/saídas do ambiente.

Assim, vista a importância destes quatro painéis, estes são apresentados nesta secção juntamente com as funcionalidades que fornecem ao utilizador.

3.2.1 Painel de visualização de dados (Painel *Robot*)

O painel "Robot" permite ao utilizador observar e alterar o estado geral do robô, assim como outras informações do ambiente, estando divido em quatro partes que convêm apresentar individualmente, como se pode ver Figura 3.6.

Figura 3.6 - Painel "Robot" e as suas quatro partes: "Inverse Kinematics", "Tool", "Joints" e "Drives"

Inverse Kinematics: Permite observar e alterar a posição cartesiana e a orientação do TCP do robô, para além de mostrar os possíveis erros linear e angular existentes nessa posição. Para além disso, também permite observar o modelo de cinemática inversa do robô, através da *checkbox* "Debug", e ligar/desligar o controlo em cinemática inversa, através da *checkbox* "Enabled".

Tool: Permite observar e alterar a posição cartesiana e a orientação do TCP em relação ao ponto de contacto da ferramenta com o robô, assim como manipular a dita ferramenta (na versão em causa ligar/desligar a sucção da ventosa usada). Para além disso, permite ainda observar o movimento do TCP através da *checkbox* "Trail" que quando selecionada faz com que o TCP desenhe o seu percurso no ambiente.

Joints: Permite operar o robô através de cinemática direta manipulando o valor do ângulo de cada motor. De notar que para tal deve-se primeiro desligar o mecanismo de cinemática inversa no sub-painel "Inverse Kinematics".

Drives: Permite observar e alterar as características do controlador de cada motor, como por exemplo, os seus ganhos e a velocidade máxima permitida.

3.2.2 Diretório de programas (Painel *Scripts*)

O painel "Scripts" permite observar a lista de programas existentes e executar um programa pretendido, como se observa na Figura 3.7. De notar, que para tal, este painel apenas mostra os programas presentes na pasta em que está guardado nos documentos do computador ("c:/…/Documents/RVP"), sendo o método escolhido para importar os programas, simplesmente colocá-los nessa pasta.

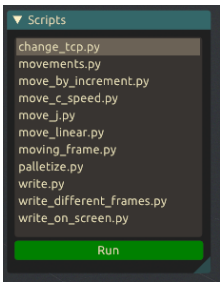

Figura 3.7 - Painel "Scripts"

3.2.3 Janela de comunicação com o utilizador (Painel *Console*)

O painel "Console" funciona como uma plataforma de comunicação entre o robô e o utilizador, onde vão ser impressas as mensagens destinadas ao utilizador, tanto as especificadas no programa como as mensagens de erro e outras informações uteis, como por exemplo, o início e finalização de programas, como se pode ver na Figura 3.8.

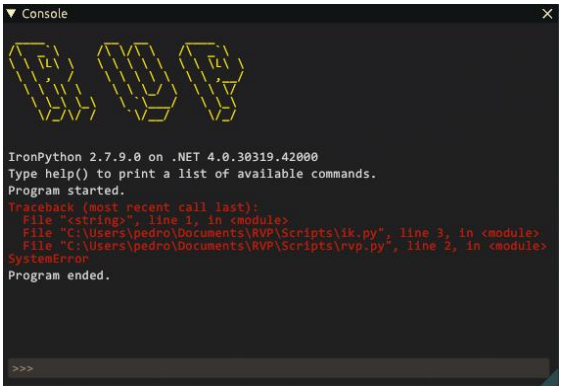

Figura 3.8 - Mensagens transmitidas pelo painel "Console"

Para além disso, este painel também permite executar linhas de código individualmente e aceder à lista de comandos capazes de manipular o robô, através da utilização do comando help().

3.2.4 Painel de comunicação MODBUS

O painel MODBUS é responsável pela comunicação com outros equipamentos, usando o protocolo de comunicação MODBUS TCP/IP. Mais precisamente, este permite estabelecer uma conexão *cliente/servidor* com outro equipamento, onde o RVP desempenha o papel de *servidor*. Para tal, é apenas necessário especificar os dados da comunicação (IP, porta e slave) no painel apresentado na Figura 3.9 e clicar no botão *Connect*.

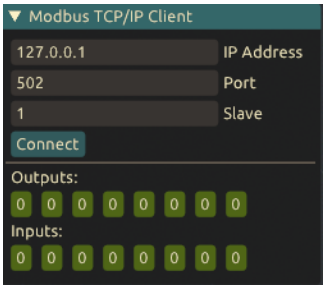

Figura 3.9 - Painel "MODBUS"

Este painel apresenta ainda as variáveis de entrada/saída do ambiente, que neste momento são apenas oito variáveis binárias de entrada e oito variáveis binárias de saída, permitindo manipular as variáveis de saída (“inputs” no painel da Figura 3.9) a qualquer momento, mesmo durante a execução de programas.

3.3 Métodos e regras de escrita fornecidas

Com a emulação do robô e a criação de todos os painéis foi então disponibilizado o primeiro protótipo da aplicação RVP apresentado na Figura 3.10. É esta versão embrionária que o autor terá de desenvolver passo a passo até atingir um patamar final que lhe confira o estatuto de “plataforma de treino de robótica virtual” de acordo com os requisitos funcionais enunciados no Capítulo 2. Para tal, esta versão já especifica o conjunto de métodos básicos necessários ao seu controlo e o conjunto de regras de escrita que o controlador do robô consegue ler, para que o autor possa escrever os métodos/programas desejados.

Figura 3.10 - Aplicação RVP fornecida e os seus painéis
1 - Janela de comunicação com o utilizador, 2 - Painel de visualização de dados, 3 - Diretório de programas, 4 - Painel de comunicação MODBUS

Começando pelas regras semânticas, importa desde já esclarecer que estas têm por base uma importante decisão tomada pela empresa: o controlador do robô é um interpretador de *Python* 2.7.16(Python Software Foundation 2020). Assim, toda e qualquer instrução ou ordem dada ao controlador terá de assentar nesta linguagem. Esta linguagem foi principalmente escolhida devido a tornar a integração do robô na aplicação mais fácil, mas também abre a possibilidade de programar o robô numa linguagem versátil, moderna e que alarga substancialmente as possibilidades da robótica programável.

Por outro lado, para controlar o robô e o ambiente em que opera, foi disponibilizado o acesso a várias propriedades destes, assim como a vários métodos capazes de as alterar. Destes os mais importantes são sem dúvida os que pertencem ao objeto robô, pois, apenas através da alteração das propriedades deste objeto, se torna possível executar os movimentos de cinemática direta e inversa desejados. Assim, começando pelo mecanismo de cinemática inversa, foi fornecido o acesso (leitura e escrita) às propriedades apresentadas na Tabela 3.2.

Tabela 3.2 - Propriedades destinadas a controlar o robô em cinemática inversa, (*) - Apenas é permitido ler a propriedade

Propriedades	*Data Type*	Descrição
robot.ik.enabled	Boolean	Alterna entre o controlo de cinemática inversa quando está a *True* e o de cinemática direta quando é *False*
robot.ik.target_xyz	Vec3 (vetor com três números reais)	Posição cartesiana no formato (x,y,z), para a qual o robô se deve movimentar
robot.ik.target_rpy	Vec3	Orientação desejada para o robô, definida através das rotações (Θ_z, Θ_x, Θ_y) a aplicar aos eixos do referencial global: uma rotação em z graus em torno do eixo Z, uma rotação em x graus em torno do eixo X e uma rotação de y graus em torno do eixo Y, por esta ordem.
robot.tool.tcp_xyz	Vec3	Posição cartesiana do TCP no formato (x,y,z). De notar que esta posição é definida em relação ao ponto de conexão entre o robô e a ferramenta.
robot.tool.tcp_rpy	Vec3	Orientação do TCP, definida pelas rotações (Θ_z, Θ_x, Θ_y) a aplicar aos eixos originais do TCP.
robot.ik.error_linear$_{(*)}$	Número real	Erro linear existente entre a posição do TCP e a posição desejada.

Por outro lado, para controlar o mecanismo de cinemática direta, foi permitido o acesso aos seis controladores dos motores do robô (*drives*), disponibilizando as propriedades presentes na Tabela 3.3. De notar, que estes drives são identificados pela sua posição no robô (entre 1 e 6), sendo apresentadas apenas as propriedades para um driver genérico *i*, com $1 \leq i \leq 6$ (*robot.drive[i]*) na Tabela 3.3.

Tabela 3.3 - Propriedades destinadas a controlar o robô em cinemática direta, (*) - Apenas é permitido ler a propriedade

Propriedades do robô	***Data Type***	**Descrição**
robot.drive_values(*)	Vetor de 7 números reais	Valor atual dos ângulos de cada motor (nas posições 1 a 6, a posição 0 é sempre igual a 0)
robot.drive_targets	Vetor de 7 números reais	Valor desejado para os ângulos de cada motor (nas posições 1 a 6, a posição 0 é sempre igual a 0)
Propriedades dos controladores do robô	*Data Type*	**Descrição**
robot.drives[i].current_value(*)	Número real	Valor atual do ângulo do motor i
robot.drives[i].target_value	Número real	Valor desejado para o ângulo do motor i
robot.joints[i].limit_lower(*)	Número real	Ângulo mínimo que uma junta, e consequentemente o motor responsável pelo seu movimento pode possuir.
robot.joints[i].limit_upper(*)	Número real	Ângulo máximo que uma junta, e consequentemente o motor responsável pelo seu movimento pode possuir.

Por fim, a biblioteca fornecida possui as possibilidades de desenhar as trajetórias do TCP do robô, desenhar linhas, ler variáveis de entrada e escrever variáveis de saída, através dos métodos e propriedades presentes na Tabela 3.4.

Tabela 3.4 - Métodos/propriedades destinados a controlar o ambiente do RVP

Propriedades	***Data Type***	**Descrição**
robot.tool.trail	Boolean	Se *TRUE*, desenha a trajetória por onde o TCP do robô passa
Métodos	Descrição	
robot.tool.clear_trail()	Apaga as trajetórias do robô desenhadas no ambiente	
debug_lines.create_line (vetor de Vec3's, color)	Desenha uma sequência de linhas retas entre uma sequência de pontos no formato Vec3, com uma cor desejada. Esta *color* é uma estrutura de dados que especifica uma cor no sistema RGB, através de numa sequência de quatro números reais: a quantidade de vermelho, a quantidade de verde, a quantidade de azul e a opacidade da linha	
debug_lines.clear()	Apaga as linhas desenhadas no ambiente pelo método *debug_lines.create_line*	
MODBUS_client.get_output (inteiro)	Devolve o estado da variável de saída do ambiente desejada	
MODBUS_client.set_input (inteiro, boolean)	Altera o estado da variável de entrada do ambiente indicada para a estrutura booleana fornecida	

3.4 Desenvolvimentos a realizar

Tendo uma aplicação capaz de emular o robô e uma biblioteca com os métodos básicos necessários ao seu controlo, é agora necessário criar métodos mais complexos a partir dos fornecidos, de forma a completar esta biblioteca com as funcionalidades enunciadas no Capítulo 2. Na Tabela 3.5 podem-se recordar essas funcionalidades, assim como reparar que algumas delas, como a edição do TCP ou suspender temporariamente (pausar) a execução do programa, já se encontram definidas na aplicação criada ou na linguagem *Python*, sendo apenas necessário criar comandos para as executar diretamente.

Tabela 3.5 - Funcionalidades a Implementar

Funcionalidade	Componentes a Implementar	RVP
Especificação de movimentos	-Movimento de juntas síncrono/assíncrono, linear, circular, de apenas uma junta, junta-a-junta e incremental	Falta
	-Controlo da velocidade e suavização das descontinuidades da trajetória	Falta
Incorporação de ferramentas/TCPs	-Apenas 1 ferramenta simples	Possui
	-Definir, mover e alterar vários TCPs	Possui
Interação com o ambiente	-Apenas variáveis binárias	Possui
	-Funções de pausa, comentários, *print*, *multi-thread* e capacidade de chamar outros programas	*Python* possui
Definição da RPL	-Criar uma biblioteca para uma linguagem existente	Possui
	-Tipificar pontos-alvo, orientações e conjuntos de posições de juntas	Possui
	-Tipificar referencias	Falta
Definição da interface gráfica	-Visualização e edição da posição atual do TCP	Possui
	-Visualização e edição do estado das juntas	Possui
	-Manipulação da juntas e TCP graficamente	Possui

Daqui, pode-se concluir que, das funcionalidades apresentadas na Tabela 1.1, as que exigem um esforço real por parte do autor, e por isso são o foco do seu trabalho, são:

- A implementação dos três tipos principais de movimentos (de juntas, linear e circular);
- O controlo da velocidade dos movimentos;
- O controlo das descontinuidades do trajeto;
- A implementação de referenciais.

Para tal decidiu-se usar a versão 1.35 do IDE *Visual Studio Code*(Microsoft 2020) apresentada na Figura 3.11 que permite escrever programas na linguagem desejada(*Python* 2.7.16), e possui várias ferramentas de depuração destinadas a ajudar na escrita desta

linguagem. De notar que esta escolha foi arbitrária e qualquer outro IDE pode ser usado em vez deste.

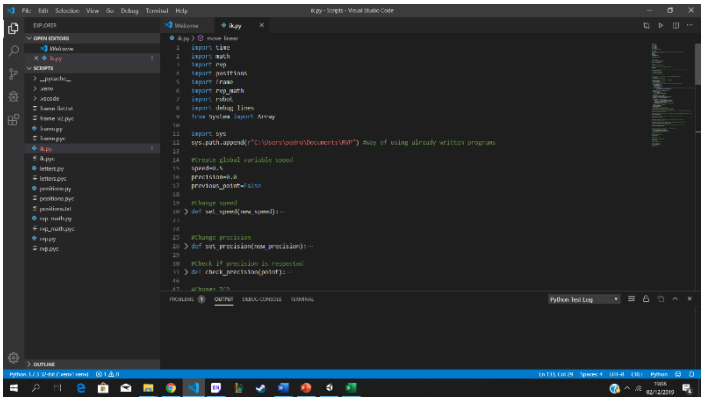

Figura 3.11 - IDE *Visual Studio Code*

3.5 Síntese

Este capítulo expôs a interface desejada para a aplicação a desenvolver, assim como os requisitos e tecnologias informáticas que foram necessários para o seu desenvolvimento, de acordo com os requisitos funcionais especificados no Capítulo 2. Para tal começou-se por especificar a aplicação final desejada: um ambiente onde o utilizador poderá armazenar e executar programas robóticos, após escrevê-los numa aplicação à sua escolha.

Depois, seguiu-se a apresentação das tecnologias usadas para criar os três modelos responsáveis pela emulação do robô virtual: o modelo matemático de cinemática inversa (*coordinate descent algorithm*) responsável por decidir quais os movimentos a realizar pelo robô, o motor físico do *Unity*, que possibilita esse movimento no ambiente virtual, e o modelo gráfico, responsável pela apresentação visual do robô e dos seus movimentos.

De seguida, foi apresentada e descrita a primeira versão do RVP criada pela Real Games, com o dito robô e os vários painéis que se consideraram necessários, descrevendo as funcionalidades disponíveis em cada um e o método escolhido para desenvolver novos programas (um ficheiro de texto segundo a linguagem *Python* 2.7.16).

Por fim, foi ainda apresentada a biblioteca de métodos e propriedades fornecidas pela Real Games, que deve ser completada com as funcionalidades enunciadas no Capítulo 2. Esta expansão será, então, feita no capítulo seguinte, através da criação de métodos novos e versões mais complexas dos métodos existentes.

4 Projeto e desenvolvimento das facilidades de programação do RVP

Na posse de uma biblioteca com as "primitivas" necessárias à programação da movimentação do RVP foi então possível começar a desenvolver e juntar a essa biblioteca métodos mais complexos, responsáveis por proporcionar as funcionalidades enunciadas no Capítulo 2. Para tal e, por uma questão de facilidade de gestão e utilização, começou-se por criar quatro bibliotecas menores, por onde esses métodos serão divididos consoante a funcionalidade que proporcionam, como se mostra na Tabela 4.1.

Tabela 4.1 - Bibliotecas de *Python* desenvolvidas

Bibliotecas a criar	Conteúdo
Frame	Métodos para controlo dos referenciais
Ik	Métodos de cinemática inversa
Rvp	Métodos de cinemática direta
Rvp_math	Métodos matemáticos genéricos

Este capítulo apresenta, então, os métodos incluídos nestas bibliotecas e explica a abordagem usada para as implementar, com especial foco nas funcionalidades que, devido à sua complexidade, exigiram uma maior atenção e trabalho; nomeadamente:

- A implementação dos três tipos de movimentos principais (de juntas, linear e circular);
- O controlo da velocidade dos movimentos;
- O controlo de descontinuidades nas trajetórias;
- A implementação de referenciais.

De notar, que os métodos da biblioteca *rvp_math* tratam funções matemáticas genéricas bem conhecidas e facilmente identificáveis. Daí se ter optado por não as discutir neste texto.

4.1 Tipificação das primitivas necessárias

Antes de descrever como foram elaborados os métodos principais convêm chamar a atenção para a maneira como foram definidos os pontos-alvo, vetores cartesianos e vetores de orientação, que são os dados de entrada de muitas delas e se apresentam na Tabela 4.2.

Tabela 4.2 - Tipificação dos pontos-alvo e vetores a usar

Primitivas	Tipo de dados	Exemplo
Pontos-alvo	3 variáveis reais e uma *string*	p = [0,5; 0,8; 1,0] p = [0,5; 0,8; 1,0; "a"]
Vetores cartesianos	3 variáveis reais e uma *string*	v1 = [0,1; 0,1; 0,2] v1 = [0,1; 0,1; 0,2; "a"]
Vetores de orientação	3 variáveis reais	Orientation = [0; 0; 90]

Começando pelos pontos-alvo e vetores cartesianos, estes foram definidos como sendo uma *array* de três ou quatro componentes, em que as três primeiras são as suas coordenadas cartesianas (*x*, *y* e *z*) e a quarta será o referencial em que está definido. De notar que esta quarta componente é opcional, sendo apenas necessário incluí-la quando o referencial a usar é diferente do global.

Por outro lado, os vetores de orientação, são especificados como *arrays* de três componentes: uma rotação em z graus em torno do eixo Z, uma rotação em x graus em torno do eixo X e uma rotação de y graus em torno do eixo Y, por esta ordem. Como dito anteriormente, estes três tipos de dados são as variáveis de entrada da maioria dos métodos a desenvolver, por isso, e por uma questão de facilidade de compreensão, estes são representados como se fossem estruturas de dados no decorrer do trabalho e na especificação dos métodos desejados.

4.2 Biblioteca *RVP*

A biblioteca RVP engloba os métodos responsáveis pelos movimentos de cinemática direta que, como esperado, são extremamente fáceis de implementar, correspondendo apenas ao uso direto de um ou dois dos métodos já existentes na aplicação fornecida, como se pode ver na Tabela 4.3. De notar que alguns deles também implementam movimentos incrementais, para aplicações de teste e porque estes podem ser interessantes num contexto de aprendizagem.

Tabela 4.3 - Métodos de cinemática direta implementados

Métodos	Descrição
move_j(array)	Recebe um vetor com os ângulos desejados para cada motor e movimenta-os para esses valores através do método *robot.drive_targets*
move_by_increment(array)	Recebe um vetor de ângulos a incrementar aos atuais, lê a posição atual de cada motor, verifica se o movimento está dentro dos limites do motor e, se estiver, executa-o
move_by_increment_separated(array)	Igual à anterior, mas incrementa uma junta de cada vez
moveto(motor,real)	Movimenta a junta desejada para a posição desejada através do método *drive.target_value*
move_joint_by_increment(inteiro,real)	Incrementa a junta desejada (identificada através da sua posição entre 1 e 6) pelo valor real fornecido
home()	Movimenta o robô para a sua posição inicial
wait_robot_position()	Espera que todos os motores do robô tenham um erro inferior a 0,1º em relação à posição final desejada, antes de prosseguir o programa
pause(real)	Espera o tempo fornecido em segundos antes de proceder a simulação

4.3 Biblioteca *Frame*

Para se incorporar referenciais no robô decidiu-se que estes seriam um dado especificado aquando da criação de pontos, ou seja, se um ponto não estiver especificado no referencial global, deve possuir uma quarta componente, correspondente a uma *string* com o nome do referencial a usar.

Posteriormente, todos os métodos de cinemática inversa irão verificar se o ponto possuí ou não essa quarta componente e se este a possuir irão alterá-lo do referencial especificado para o global através do método *change_point_frame*, antes de continuar o programa. Na prática isto foi implementado e pode ser usado através dos métodos apresentados na Tabela 4.4:

Tabela 4.4 - Métodos para manipulação de referenciais

Métodos da classe frame	**Descrição**	
Frame(ponto,vetor,vetor,vetor)	Cria um objeto *frame* (referencial) através do seu ponto de origem e dos três vetores ortonormais que o definem, expressos no referencial global.	
get_universal_point(self,ponto)	Transforma as coordenadas do ponto para o referencial global	
get_universal_vector(self,vetor)	Transforma as coordenadas do vetor fornecido para o referencial global	
Propriedades de um frame	***Data Type***	**Descrição**
origin	Ponto	Origem do *frame* expressa no referencial global
vector1, vector2 e vector3	Vetor	Vetores bases do *frame* expressos no referencial global
transformation_matrix	Matriz[3][3]	Matriz de transformação do *frame* para o referencial global
Outros Métodos	**Descrição**	
new_frame(string,ponto, vetor,vetor,vetor)	Cria e guarda um *frame* num ficheiro de texto, identificando-o através do nome (string) fornecido	
get_frame(string)	Devolve o *frame* guardado no ficheiro de texto	
change_point_frame(ponto)	Verifica em que referencial o ponto está e devolve o ponto equivalente no referencial global	
change_vector_frame(vetor)	Verifica em que referencial o vetor está e devolve o vetor equivalente no referencial global	

De notar que os métodos *change_point_frame* e *change_vector_frame* foram criados na biblioteca *ik* onde são utilizados.

4.4 Biblioteca *Ik*

A biblioteca *Ik*, possui os métodos responsáveis pelos vários movimentos desejados, assim como vários métodos responsáveis pelo controlo das propriedades desses movimentos. Desses os métodos principais, encontram-se apresentados na Tabela 4.5. No entanto, convêm denotar que a criação destes métodos, envolveu processos muito menos óbvios que os das bibliotecas RVP e Frame, e por isso estes métodos serão explicados sucintamente ao longo desta secção.

Tabela 4.5 - Métodos principais da biblioteca Ik

Métodos	Descrição
set_speed(real)	Altera o valor da velocidade do TCP robô para a fornecida. De notar que a velocidade inicial é 0,5 ms^{-1}
set_precision(real)	Altera o valor do erro de precisão aceitável nas descontinuidades da trajetória para o valor fornecido. De notar que o valor inicial é 0 m
move_j(ponto)	Move o robô para a posição fornecida da maneira mais rápida possível
move_j_all(ponto,ponto)	Move o robô do primeiro ponto fornecido para o segundo da maneira mais rápida possível
move_linear(ponto)	Move o robô para o ponto fornecido através de uma linha reta
move_linear_all(ponto,ponto)	Move o robô do primeiro ponto fornecido para o segundo através de uma linha reta
move_c(ponto,ponto)	Move o robô para o segundo ponto fornecido através de um círculo que passa pelo primeiro ponto fornecido
move_c_center (ponto,ponto,real)	Move o robô para o ponto desejado através de um círculo especificado pelo ponto atual, o ponto desejado (primeira entrada), o centro (segunda entrada) e a direção a tomar
move_c_center_all (ponto,ponto,ponto,real)	Move o robô através de um círculo especificado pelo ponto inicial (primeira entrada), o ponto final (segunda entrada), o centro (terceira entrada) e a direção a tomar
wait_robot_position()	Assegura que o robô atingiu a posição desejada antes de prosseguir o programa
check_precision(ponto)	Assegura que o robô respeitou a precisão desejada no ponto fornecido antes de prosseguir o programa

Nesta altura, convêm também referir que existem dois tipos de controlo dos movimentos em robótica:

- Controlo continuo de trajetória – Consiste na decomposição de cada movimento em pequenos intervalos, de forma a se impor e corrigir a trajetória a cada momento
- Controlo sincronizado de trajetória – Consiste no cálculo do ponto final e dos movimentos que cada um dos eixos do robô tem de realizar. Permite sincronizar os eixos de forma a começarem e acabarem o movimento ao mesmo tempo, assim, como aplicar gradientes ao movimento desses eixos, realizando movimentos lineares e circulares.

Obviamente que o mais interessante e capaz deste dois é o controlo sincronizado de trajetória; no entanto, como dito anteriormente o método de cinemática inversa escolhido não permite saber os valores das juntas do robô do ponto final à priori. Daí o tipo de controlo a usar neste projeto foi o "Controlo continuo de trajetória" e a representação dos movimentos lineares, circulares e controlo de velocidade tiveram de ser emulados por métodos alternativos(Cardial et al. 1988).

De notar que ao contrário destes movimentos, o movimento de juntas síncrono apenas pode ser realizado através do controlo sincronizado de trajetória e por isso esta biblioteca apenas possui a versão assíncrona do movimento de juntas.

4.4.1 Garantia de alcance do ponto alvo – Método *wait_robot_position*

Antes de especificar os movimentos desejados convêm possuir um método capaz de detetar que o robô efetivamente realizou o movimento desejado, ou seja, atingiu o ponto final deste, o que é, obviamente, obrigatório no fim da especificação de cada movimento, senão o controlador do robô irá ordenar todos os movimentos de seguida e o robô só realizará o último deles. Para tal criou-se então o método *wait_robot_position* também na ninlioteca Ik.

Em cinemática direta este método calcula constantemente a diferença entre o ângulo atual de cada um dos motores e o desejado até que este seja menor que 0,1 graus (o erro permitido) e, apenas nesse momento, prossegue o programa.

Quanto à cinemática inversa, infelizmente a arquitetura implementada não permite usar o mesmo método pois a verificação da posição é feita segundo as coordenadas cartesianas do TCP e o erro linear daí resultante varia muito com a distância à base do robô. Consequentemente, qualquer limite escolhido ou vai dar um falso positivo nas proximidades do robô ou será impossível de respeitar nas extremidades. Daí a solução escolhida para este problema foi ir medindo constantemente o erro e assumir que o robô chegou ao seu destino quando este estabiliza (na prática é calculada a diferença entre o erro atual e a média dos dez anteriores). Isto funciona bem, mas faz com que exista um *delay* (o tempo da leitura das dez medições) entre o momento que o robô chega ao seu destino e o momento que o programa prossegue.

4.4.2 Curva de suavização das descontinuidades da trajetória – Método *set_precision*

Como dito anteriormente, em robótica os movimentos são especificados entre cada sequência de dois pontos, sem saber o aconteceu antes, ou acontecerá a seguir; por isso é normal aparecerem mudanças súbitas de trajetória que obrigam o robô a desacelerar e literalmente parar no ponto final do movimento antes de começar o próximo, para evitar *overshoot*.

Obviamente estas flutuações de velocidade são prejudiciais para os motores e em várias aplicações industriais como, por exemplo, soldadura, por isso a maioria dos robôs oferecem a hipótese de suavizar a transferência entre os movimentos através de uma curva nas redondezas do ponto de mudança (aresta da trajetória), como se pode ver na Figura 4.1.

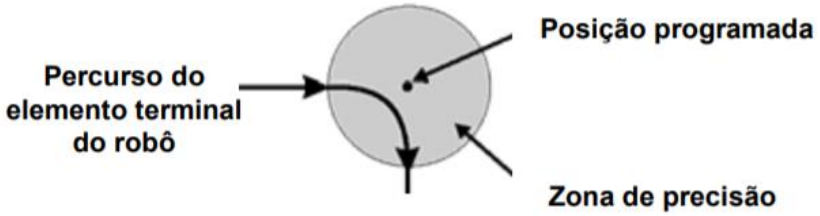

Figura 4.1 - Método de suavização das descontinuidades de velocidade(Abreu 2018)

Esta curva é definida previamente, pelo valor do erro de posição que ela provoca: quanto maior o erro aceitável, maior a curvatura e menor a diminuição de velocidade, existindo, por isso, um compromisso entre este erro e a velocidade desejada nas redondezas das arestas da trajetória(RoboDK 2019b).

No projeto em causa, isto foi então especificado através de uma variável global que representa o erro de precisão aceitável chamada de *precision* e dois métodos, um destinado a permitir ao utilizador alterar o valor a usar (*set_precision*) e outro chamado no fim de cada movimento de forma a apenas deixar prosseguir o programar assim que o requisito de precisão é cumprido (*check_precision*), como se pode ver na Tabela 4.6. De notar que se o valor escolhido for zero vai ser chamada a função *wait_robot_position*, em vez de entrar no ciclo, pois é impossível atingir um erro igual a zero e, como já foi apresentado previamente, a função *wait_robot_position* prossegue o programa quando o menor erro possível é atingido.

Tabela 4.6 - Métodos para controlo da curva de suavização das descontinuidades da trajetória

Métodos	Descrição
set_precision(real)	Altera o valor da variável global precisão para o desejado, de notar que o valor inicialmente predefinido é 0 metros
check_precision(ponto)	Entra num ciclo em que calcula constantemente a distância entre o ponto atual e o desejado, apenas saindo quando a diferença entre os dois for menor que a precisão desejada.

É ainda importante referir que para a precisão operar corretamente é preciso guardar o ponto do destino anterior, pois o movimento seguinte deve usar esse ponto anterior como ponto de início em vez da posição atual do robô, se não o erro de precisão vai-se propagar pelo novo movimento. Isto foi implementado através de uma variável global que guarda o ponto desejado: *previous_point*.

4.4.3 Controlo de velocidade – Método *set_speed*

Normalmente, o método usado para controlar a velocidade em robótica consiste em calcular o contributo de cada um dos seis motores para o movimento, através do seu posicionamento no início e no fim deste, e manipular a velocidade máxima de cada um de forma a respeitar a velocidade desejada no TCP. No entanto, como já foi referido, o método de cinemática inversa usado não nos permite saber o estado das juntas no ponto final, o que torna impossível

o controlo da velocidade, sendo necessário aplicar algo capaz de emular indiretamente este controlo.

Para isso, decidiu-se dividir o movimento em vários pequenos movimentos e controlar a distância e tempo decorridos entre cada ordem de movimento, pois a velocidade é igual à distância a dividir pelo tempo, logo se tanto a distância a percorrer como o tempo estão limitados a velocidade também está. Daqui resulta que, embora na prática, o robô esteja a movimentar-se à maior velocidade possível entre cada pequeno movimento, como se vê na Figura 4.2, a velocidade do movimento geral pode ser controlada.

Por exemplo, na Figura 4.2, querendo uma velocidade de 1,0 ms^{-1} e usando um intervalo entre comandos de 0,1 s, o robô calcula que a distância entre dois pontos seguidos deverá ser 1,0 * 0,1 = 0,1 m e, como tal, divide o movimento em vários incrementos distanciados desse valor, ordenando ao robô para se movimentar entre esses pontos no intervalo definido de 0,1 s, donde resulta a velocidade de 1,0 ms^{-1} desejada.

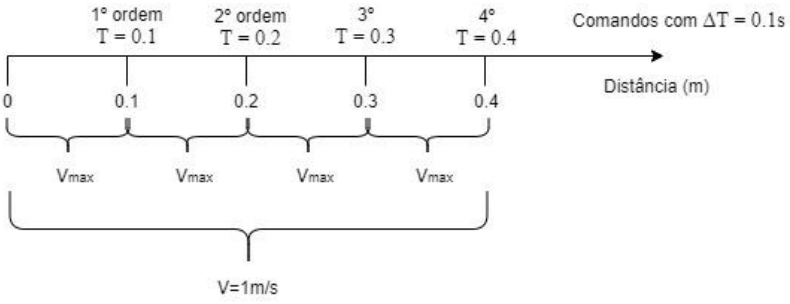

Figura 4.2 - Exemplo do controlo de velocidade para uma velocidade desejada de 1,0 m/s e uma cadencia de comandos de 0,1 s

Para especificar isto no projeto a desenvolver foi então criada a variável global *speed* e o método *set_speed* capaz de a alterar. Como os movimentos realizados são à base de incrementos, esta variável global é depois usada para calcular o incremento a usar em cada caso.

4.4.4 Movimento linear – Método *move_linear*

Tal como para a velocidade, o movimento linear propriamente dito necessita do conhecimento do estado final das juntas do robô, que não é obtenível na arquitetura usada, por isso foi necessário usar novamente algo que não é o movimento desejado, mas o falseia a um nível aceitável.

Para tal, o movimento linear foi implementado através do método *move_linear(ponto)* que, usa apenas o ponto final do movimento como entrada. Este lê a posição atual do robô e cria um vetor correspondente ao movimento desejado ao subtrair as coordenadas do ponto inicial às do ponto final. Este vetor será depois dividido em vários incrementos consoante a velocidade especificada, que serão realizados sequencialmente através de movimentos de juntas como se vê na Figura 4.3. Daqui resulta, que, com um número suficientemente grande de incrementos, o movimento parece linear.

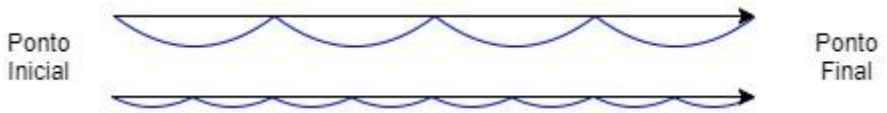

Figura 4.3 - Emulação de um movimento linear (a preto) através de vários movimentos de juntas (a azul)

De notar, que se decidiu guardar também um método intermédio usado durante a criação do método *move_linear*, nomeadamente o método *move_linear_all*, que usa como entrada o ponto inicial e final desejados, pois pode ser interessante em algumas situações.

4.4.5 Movimento circular – Método *move_c*

O movimento circular foi implementado através da equação paramétrica de um círculo em 3D, que usa dois vetores ortogonais e unitários v1 e v2 e o ponto central c do círculo desejado, para desenhar um círculo no plano definido por esse ponto e vetores, através da alteração do ângulo teta entre 0 e 2π:

$$(x, y, z) = c + r * \cos(\theta) * v1 + r * \sin(\Theta) * v2 \quad (4.1)$$

Esta equação permite, ao variar o ângulo Θ, obter a sequência de pontos por onde o robô deve passar para realizar a circunferência desejada, assim como, através do cálculo inverso, calcular o ângulo da posição inicial e final desejadas.

Daqui tendo o ângulo inicial e final, pode-se calcular a sequência de pontos do movimento desejado e aplicar o mesmo método usado para o movimento linear: executar vários incrementos através de movimentos de juntas.

Assim, obteve-se uma primeira versão funcional do movimento circular, à qual foi posteriormente adicionada uma variável *booleana* capaz de controlar a direção com que se percorre a equação paramétrica, ou seja, qual dos dois semicírculos a percorrer entre o ponto inicial e final. De notar que mesmo assim, só por tentativa e erro é que se consegue descobrir qual dos dois semicírculos é realizado a *True* e a *False*. O método daqui resultante foi o *move_c_center_all*.

Após assegurar o correto funcionamento desta etapa, criou-se, então, o movimento a partir de 3 pontos do círculo. Esta versão é mais interessante pois a definição de dois pontos e do centro obriga a que estes pontos sejam muito bem especificados: a distância entre os dois pontos e o centro (raio) tem de ser exatamente a mesma ou o programa dará erro, o que pode ser terrivelmente difícil de obter num ambiente tridimensional. Para além disso, ao utilizar o método dos 3 pontos, a direção também fica automaticamente definida pelo ponto intermédio sendo preciso menos um dado de entrada.

Na prática, isto foi implementado, a partir do método da biblioteca *rvp_math get_center(ponto, ponto, ponto)*, que usa fórmulas matemáticas para obter o ponto central a partir dos três pontos fornecidos e depois executa o método descrito previamente.

Tendo esta parte funcional, o método foi ainda simplificado para usar o ponto final do movimento anterior como ponto inicial, passando a ser apenas necessário especificar o ponto intermédio e o final ao chamá-lo. Assim obteve-se o método *move_c* que realiza o movimento circular da forma mais simples; no entanto, visto que alguns dos métodos intermédios também possuem algum interesse, decidiu-se guardar as versões apresentadas na Tabela 4.7 que usam diferentes dados de entrada.

Tabela 4.7 - Métodos criados para realizar o movimento circular

Métodos	Entradas
move_c_center_all(ponto,ponto,ponto,integer)	Ponto inicial, ponto final, centro e direção
move_c_center(ponto,ponto,integer)	Ponto final, centro e direção
move_c_all(ponto,ponto,ponto)	Ponto inicial, intermédio e final
move_c(ponto,ponto)	Ponto intermédio e final.

4.4.6 Métodos complementares

Nesta subsecção listam-se outros métodos presentes na biblioteca Ik, cuja implementação não importa apresentar em detalhe pois são extremamente simples. Isto inclui também alguns métodos que resultaram apenas da reformulação semântica dos fornecidos de forma a uniformizar a sintaxe usada, como se pode ver na Tabela 4.2. No entanto convêm realçar os métodos de desenhar linhas no ambiente que permitem comparar o trajeto realizado pelo robô com o desejado e avaliar o seu desempenho.

4.2 - Métodos auxiliares da biblioteca Ik

Métodos	Descrição
change_TCP(ponto,orientação)	Modifica a posição do TCP em relação à última junta do robô, através das funções
change_orientation (real,real,real)	Altera a orientação do robô. Recebe como entrada os ângulos das três rotações a realizar em torno dos eixos na ordem (Θz,Θx,Θy)
pause(real)	Espera o tempo fornecido em segundos antes de proceder a simulação
trail_on()	Desenha a trajetória por onde o TCP passa
trail_off()	Para de desenhar a trajetória do TCP
clear_trail()	Apaga as trajetórias desenhadas no ambiente
home()	Movimenta o robô para a sua posição inicial
create_line_move_l (ponto,ponto)	Desenha um movimento linear no ambiente
create_line_move_c (ponto,ponto,ponto)	Desenha um movimento circular através do ponto inicial, final e intermédio deste
create_line_move_c_center (ponto,ponto,ponto,real)	Desenha um movimento circular através do ponto inicial, final, centro e direção deste
clear_debug_lines()	Apaga as linhas desenhadas através do comando *debug_lines.clear*
Set_color(real,real,real)	Define a cor a utilizar para as linhas desenhadas no ambiente através do código RGB

4.5 Síntese

Este capítulo descreveu a metodologia usada no contexto da programação do RVP, explicando detalhadamente o processo de desenvolvimento das funcionalidades enunciadas no capítulo 2, com atenção especial aos métodos de manipulação de referenciais, movimentos lineares, circulares e controlo da velocidade/curvas de suavização das descontinuidades da trajetória. Para além disso foram apresentadas de forma sucinta as quatro bibliotecas desenvolvidas que serão, agora, juntadas à biblioteca inicialmente fornecida de forma a completar a aplicação RVP e testá-la.

No próximo capítulo serão relatados ensaios de software para testar os vários métodos desenvolvidos e avaliar se a implementação das funcionalidades desejadas foi feita de forma satisfatória.

5 Experimentação e teste da solução

Implementadas as funcionalidades pretendidas, importa testá-las para verificar se estas funcionam e satisfazem os requisitos especificados no Capítulo 2. Para tal, foram criados pequenos programas para testar os vários métodos individualmente, assim como um programa mais complexo capaz de representar uma aplicação de robótica comum em ambientes industriais, nomeadamente, uma operação de paletização de caixas.

No entanto, para poder realizar estes testes, principalmente a operação de paletização, foi necessário simular um ambiente externo dentro da aplicação, sendo necessário o contributo da Real Games. Assim, os engenheiros da empresa envolveram-se novamente no projeto para tratar dos novos problemas de design e controlo, acabando por decidir adicionar ao ambiente os componentes que se podem observar na Figura 5.1:

- Dois tapetes transportadores com um sensor de proximidade cada;
- Emissor de caixas com 0,3 m de comprimento por 0,2 m de largura;
- Emissor de paletes com 1,0 m de comprimento por 0,8 m de largura;
- Uma ferramenta de sucção colocada no elemento terminal do robô com dois métodos: *suction_on*() e *suction_off()* para ligar e desligar, respetivamente a sucção.

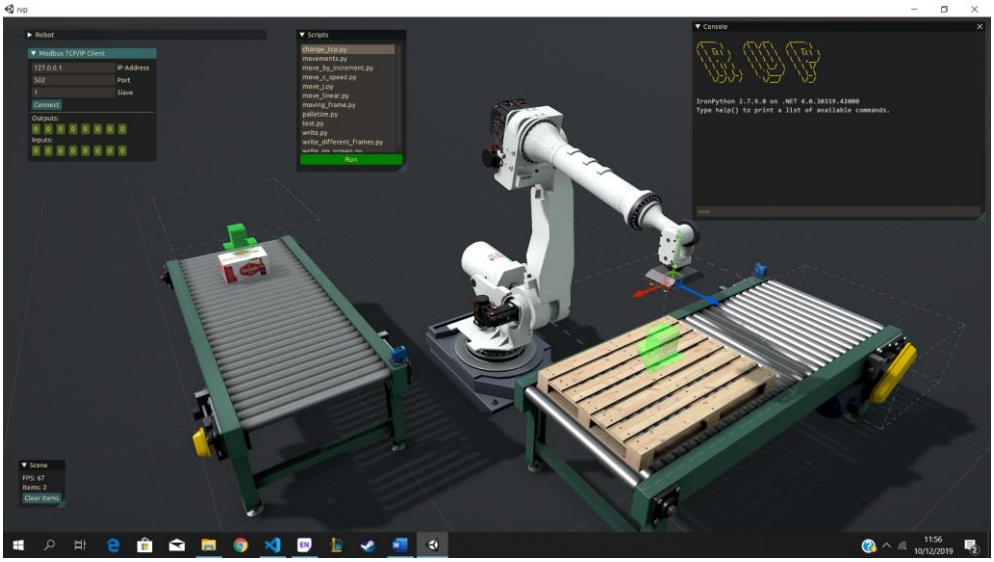

Figura 5.1 - Novo ambiente desenvolvido com tapetes transportadores, sensores, caixas e paletes

É ainda importante referir o *software/hardware* em que estes programas foram executados, apresentado na Tabela 5.1, e que nestas condições os programas correram a uma frequência de 67 FPS (*frames per second*). No entanto convêm denotar que o RVP também pode correr em computadores de menor desempenho que este.

Tabela 5.1 - Especificações do computador usado

Sistema Operativo	Windows 10 Home 64bits
Processador	Intel® Core™ i7-7700HQ CPU @ 2.80GHz
RAM	16 GB
Placa Gráfica	NVIDIA GeForce GTX 1050

5.1 Ensaio Nº1 - Teste/demonstração dos métodos de cinemática direta

Este ensaio destina-se a avaliar e demonstrar os métodos de cinemática direta implementados, assim como, exemplificar o procedimento de programação a usar pelo utilizador para escrever programas. Para tal foi criado um programa que invoca os métodos de cinemática direta (*move_j*, *move_by_increment*, *move_by_increment_separated* e *move_joint_by_increment*) e imprime o erro entre a posição angular das juntas do robô desejadas e as obtidas, após a utilização de cada um dos métodos. Esses erros permitiram então avaliar o desempenho destes métodos.

Para além disso, foram também usados várias vezes os métodos de *home*, *pause*, e impressão de mensagens para explicar ao utilizador o que vai passando e verificar o seu correto funcionamento. O programa resultante foi chamado de *move_by_increment.py* e o resultado da sua execução pode ser visto na Tabela 5.2.

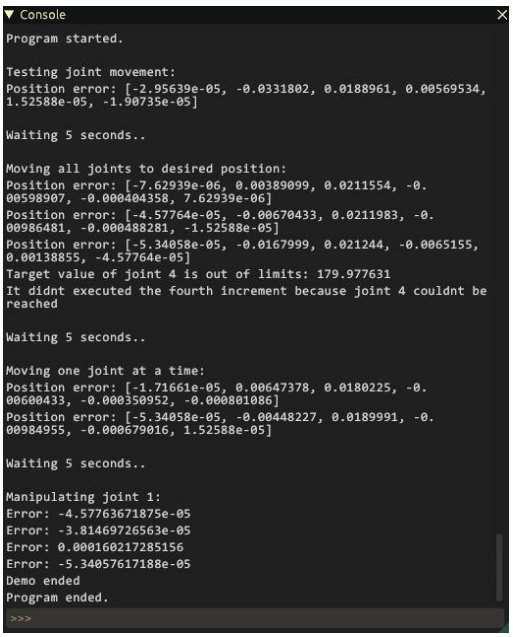

Figura 5.2 - Mensagens impressas pelo programa *move_by_increment.py*

Tendo em conta os dados obtidos pode-se concluir que os movimentos de cinemática direta são feitos com grande precisão, possuindo erros raramente superiores a 0,05º em cada motor; no entanto, convêm denotar que o valor máximo do erro obtido depende de motor para motor

como se vê na Tabela 5.2, possivelmente devido ao tamanho/peso da junta que é movida ou ao facto de possuírem velocidades diferentes.

Tabela 5.2 - Ordem do erro máximo observado para cada motor

Motor	Erro Máximo
1	$5{,}0*10^{-4}$ º
2	$5{,}0*10^{-2}$ º
3	$5{,}0*10^{-2}$ º
4	$1{,}0*10^{-2}$ º
5	$5{,}0*10^{-3}$ º
6	$1{,}0*10^{-4}$ º

Por outro lado também convêm realçar que, se executado consecutivamente, este programa imprime exatamente os mesmos valores de erro, mostrando o caracter determinístico do algoritmo utilizado; no entanto, após vários movimentos, é comum os valores de um ou dois dos motores alterarem-se para outros na mesma ordem de grandeza, provavelmente devido ao motor físico utilizado, que após algumas iterações provoca alterações nos transdutores de posição.

Por fim, quanto às funções de pausa, *home* e impressão de mensagens conclui-se que funcionam como desejado.

5.2 Ensaio Nº2 - Demonstração do movimento linear e circular

Este ensaio destina-se a demonstrar os movimentos linear e circular, apresentando como executá-los e as capacidades que fornecem ao robô, antes de avaliar as suas qualidades (precisão e repetibilidade) no ensaio seguinte. Para tal decidiu-se criar um programa para fazer o robô escrever “FEUP REAL GAMES” no ambiente 3D. No entanto, para tornar esta função mais interessante e usar as funcionalidades da linguagem *Python* ao máximo, decidiu-se não usar os movimentos para criar as letras diretamente no programa, mas definir uma nova biblioteca com um método destinado a escrever cada letra, permitindo ao utilizador escrever qualquer frase desejada no ambiente.

Assim foi primeiramente criada uma biblioteca extra chamada *letters* com os 26 métodos necessários (um para cada letra), nomeadamente *alphabet_a* para a letra *A*, *alphabet_b* para a letra *B* e assim por diante. Estes possuem como dados de entrada o ponto onde se começará a escrever e os dois vetores ortogonais que definem o plano de escrita, como se pode ver para o caso do método *alphabet_f* apresentado na Figura 5.3**.**

```
def alphabet_f(point,v1,v2):
    point1 = rvp_math.sum_vector([point,rvp_math.mv(3,v2)])
    point2 = rvp_math.sum_vector([point1,rvp_math.mv(1.5,v1)])
    point3 = rvp_math.sum_vector([point,rvp_math.mv(2,v2)])
    point4 = rvp_math.sum_vector([point3,rvp_math.mv(1.5,v1)])
    ik.move_linear_all(point,point1)
    ik.move_linear_all(point1,point2)
    ik.move_linear_all(point3,point4)
```

Figura 5.3 - Método *alphabet_f*

De notar que, de forma a facilitar a definição destes métodos, decidiu-se que os vetores do plano fornecidos são usados diretamente para definir a letra, estando esta desenhada dentro de um retângulo com 2 vezes o tamanho do primeiro vetor por três vezes o tamanho do segundo.

De seguida, foi ainda criado o método *write*, que recebendo uma *string*, o ponto inicial e os 2 vetores, escreve o texto encontrado na *string* em maiúsculas, o que significa que o utilizador apenas terá de escrever o programa apresentado na Figura 5.4, para escrever qualquer frase desejada no ambiente. Por exemplo, neste caso "FEUP REAL GAMES", como se observa na Figura 5.5.

```
1   import ik
2   import letters
3   from ik import *
4
5   set_speed(0.2)
6   clear_trail()
7   vector1=[-0.1,0,0]
8   vector2=[0,0.1,0]
9   point= [1.7,0.8,1]
10  trail_on()
11  letters.write("feup real games",point,vector1,vector2)
12  trail_off()
```

Figura 5.4 - Programa necessário à escrita de frases no ambiente, chamado de *write_on_screen.py*

Figura 5.5 - Resultado da execução do programa *write_on_screen.py*

Daqui, pode-se observar claramente a existência dos movimentos lineares e circulares (círculos de três pontos) a um nível aceitável, mas também se nota a existência de um erro de repetibilidade através dos espaços visíveis entre algumas retas que deviam estar conectadas, como se pode ver nas pernas das letras "E" e "F" ao realizar uma aproximação à palavra "FEUP" na Figura 5.6.

Segundo o painel "Robot" do RVP este erro depende da distância à base do robô, como esperado, e é, no limite do espaço de trabalho do robô, aproximadamente 4 mm; no entanto de forma a avaliá-lo melhor realizou-se o ensaio apresentado na secção seguinte.

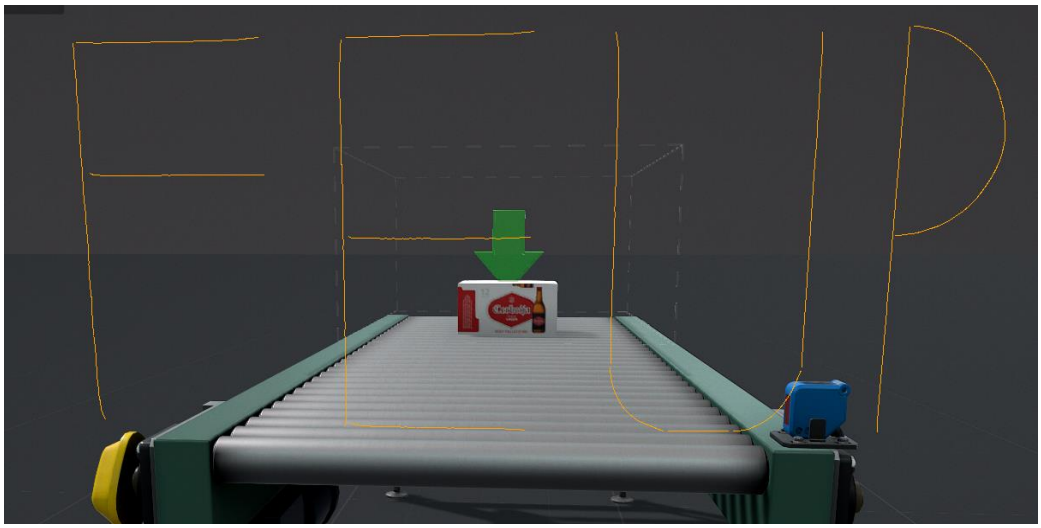

Figura 5.6 - Erros do robô

5.3 Ensaio Nº3 - Avaliação da precisão e repetibilidade do robô

Este ensaio destina-se a quantificar o erro de precisão e de repetibilidade do robô. Para tal foram usados os métodos do movimento linear para desenhar uma linha repetidas vezes e os métodos de criação de *debug_lines* para emular uma régua no ambiente. De notar ainda que também se variou o ponto onde o robô estava antes de receber o comando, de forma a também avaliar o efeito desta posição inicial.

Assim, começou-se por escrever o programa resultante destas especificações que se apresenta na Figura 5.7 e denomina-se por *repeatability.py*.

repeatability.py > ...

```
import ik
from ik import *

set_speed(0.2)
trail_off()
clear_trail()
initial_error=[0,0,0,0,0,0,0,0,0,0]
final_error=[0,0,0,0,0,0,0,0,0,0]

random_pos=[[ 1.10117747, 0.97193114, 2.07961397],···


for i in range(0,10):
    move_j(random_pos[i])
    move_j([-0.5,1.5,-0.9])
    pause(1)
    initial_error[i]=robot.ik.error_linear
    trail_on()
    move_linear([-1.5,1.5,-0.9])
    pause(1)
    trail_off()
    final_error[i]=robot.ik.error_linear

set_color([0,0,250])
clear_debug_lines()
for i in range(-5,5):
    increment=-0.9+i*(+0.001)
    begin=[-0.5,1.5,increment]
    end=[-1.5,1.5,increment]
    create_line_move_l(begin,end)
set_color([0,250,0])
create_line_move_l([-0.5,1.5,-0.9],[-1.5,1.5,-0.9])
```

Figura 5.7 - Programa *repeatability.py*

Este programa foi depois executado, resultando nas retas apresentadas na Figura 5.8, onde se encontram os percursos realizados pelo robô entre os pontos [-0,5; 1,5; -0,9] e [-1,5; 1,5; -0,9] a amarelo, a reta desejada a verde e as *debug_lines*, distanciadas de um milímetro entre si, a azul de forma a emular uma régua. Também convêm denotar que este ensaio foi feito a uma velocidade de 0,2 ms^{-1}.

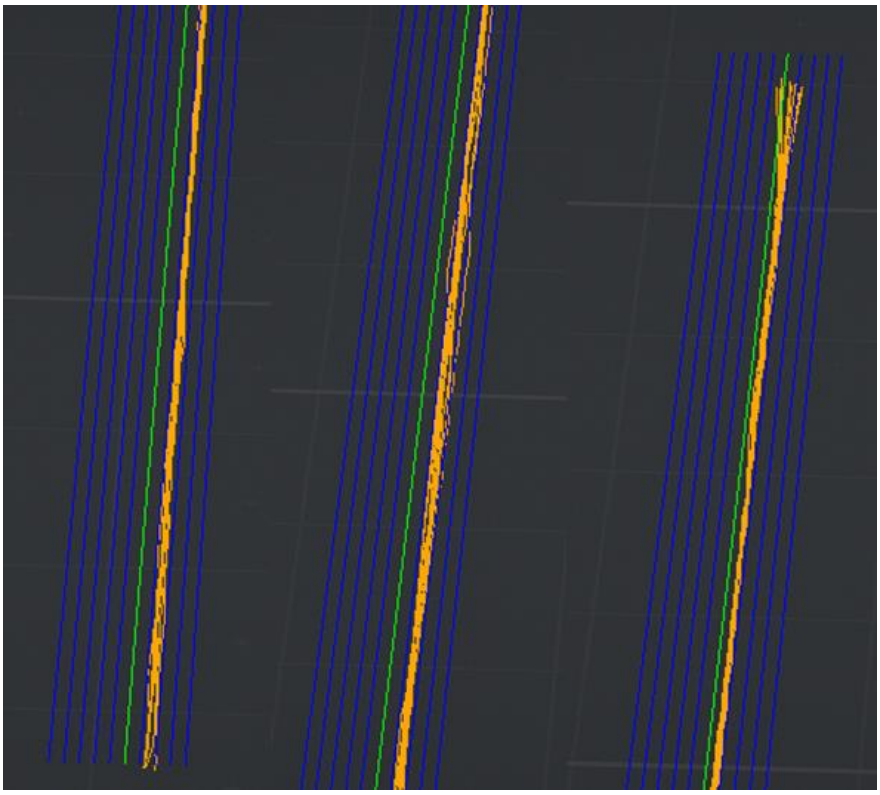

Figura 5.8 - Secções de aproximadamente 0,1 m das retas resultantes do programa *repeatability.py*

Avaliando então estas retas, a primeira coisa que se repara é a existência de um erro de precisão, que varia ao longo do movimento, mas em média parece situar-se entre 1,0 e 1,5 mm. No entanto convém relembrar que este erro está projetado numa régua 2D, mas é na verdade 3D, sendo obviamente maior. Por exemplo, a média dos erros do ponto inicial também obtida neste programa foi de 2,59 mm e a dos pontos finais 3,61 mm. Daqui, convém realçar que, como esperado, o erro é maior quanto mais longe se está da base do robô.

Por outro lado, quanto à repetibilidade repara-se que esta também é cerca de 1 mm, mas que em certos pontos da trajetória chega a atingir os 2 mm, isto apenas na régua 2D, obviamente o erro 3D será maior como para a precisão.

Por fim convém ainda referir que a precisão/repetibilidade dos pontos iniciais é similar em todas as retas, podendo-se concluir que o movimento anterior não afeta o erro deste movimento, ou se o fizer, é numa proporção muito menor ao erro já existente. Da mesma forma, também se pode esperar que estes erros sejam similares nos movimentos circulares pois eles, tal como os lineares, consistem de pequenos movimentos de juntas.

Obviamente se este erro é aceitável ou não dependerá da aplicação a realizar; no entanto convêm referir que ele é uma ordem de grandeza superior ao do robô real no qual o RVP foi baseado, que possui um erro de repetibilidade apenas ±0,2 mm, reduzindo obviamente a gama de operações capaz de efetuar.

5.4 Ensaio Nº4 - Teste/demonstração do controlo de velocidade

Este ensaio destina-se a testar o efeito de diferentes velocidades na trajetória no movimento. Para tal, foram usados os métodos dos movimentos lineares e circulares em conjunto com o método *set_speed* capaz de alterar a velocidade a que o TCP do robô se deve movimentar. De notar que alguns dos testes foram realizados a velocidades que o robô real não consegue cumprir para observar a resposta do robô emulado a esse tipo de situações.

Começando pelo teste da velocidade no movimento linear foi simplesmente chamado o método *move_linear* para realizar o movimento linear entre os pontos [1,0; 1,0; 1,0] e [2,0; 1,0; 1,0] a uma velocidade de 1,0; 2,5 e 4,0 ms^{-1}, como se vê na Figura 5.9.

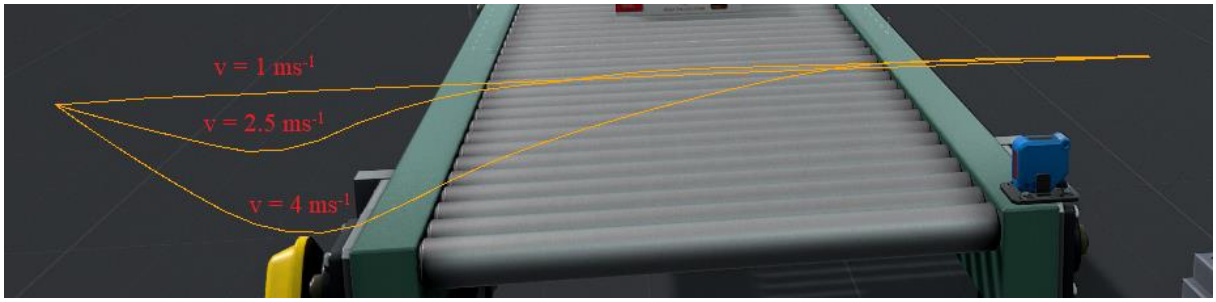

Figura 5.9 - Movimento linear a diferentes velocidades (programa *move_linear_speed.py*)

Daqui, vê-se claramente que quanto maior a velocidade mais o movimento realizado se afasta do movimento linear desejado, o que já era esperado, pois a velocidade foi especificada através do envio de ordens sucessivas ao robô, assumindo que este era capaz de as acompanhar, o que na realidade não acontece, pois, o robô possui uma velocidade máxima, que provavelmente é menor que os 2,5 ms^{-1}. Este facto vai fazer com que o robô receba uma nova ordem de movimento antes de acabar a anterior, alterando imediatamente o trajeto para cumprir a nova ordem e ficando uma parte da anterior por cumprir. Estas pequenas partes por cumprir vão somando ao longo do movimento aumentando cada vez mais o erro.

Na Figura 5.10 pode-se ver um exemplo deste mecanismo. Nesta é possível ver a castanho o movimento linear desejado entre o ponto inicial e final, assim como o processo que o controlador do robô normalmente faria para o desenhar: dividi-lo em vários movimentos de juntas a preto e ordená-los ao robô no intervalo de tempo que produz a velocidade desejada. Assim, se a velocidade do robô fosse maior que a desejada, como é normal, ele atingiria o ponto final (círculos pretos) de cada movimento de juntas, antes de receber a ordem para realizar o próximo movimento. No entanto, como a velocidade máxima do robô é menor que a desejada, ele não consegue terminar o 1º movimento de juntas a tempo e recebe a ordem para realizar o 2º a meio deste (no ponto vermelho), mudando o trajeto a realizar para o movimento de juntas a vermelho, que está muito mais longe da trajetória linear desejada que os movimentos de junta pretos. Isto vai repetir-se nos movimentos seguintes e o erro continuará sempre a aumentar, resultando no movimento final a laranja.

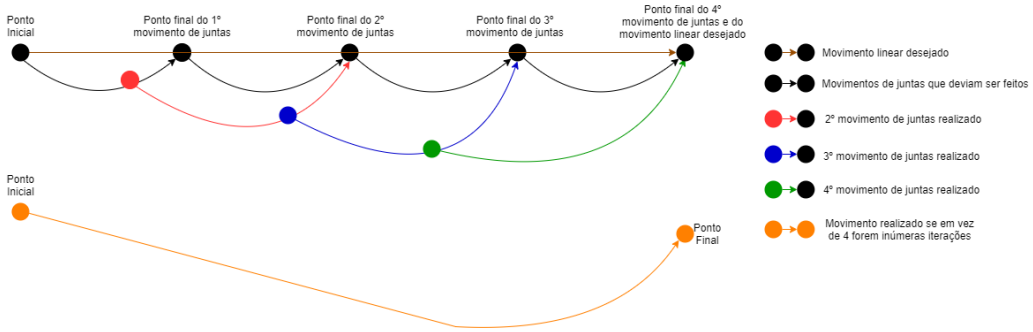

Figura 5.10 - Exemplificação da acumulação do erro de velocidade

De notar que este erro é sempre acumulativo e deve ser corrigido, possivelmente através da limitação da velocidade máxima; no entanto visto que a velocidade máxima que não produz erro varia de movimento para movimento devido à estrutura do robô, esta limitação é difícil de impor.

Também é de notar que que este erro não é encontrado em outras aplicações de robótica semelhantes, pois nestas a velocidade do TCP é controlada através da manipulação direta da velocidade de cada motor, e se algum deles não for capaz de cumprir a velocidade desejada, a velocidade de todos os motores é reduzida proporcionalmente, mantendo a trajetória correta a uma velocidade do TPC menor, mas corresponde à máxima possível para o dito movimento).

Quanto ao movimento circular foram também desenhados vários semicírculos idênticos a velocidades de 0,5; 2,0 e 4,0 ms^{-1} com os métodos *move_c_center_all* e *move_c_all*, apresentados na Figura 5.11, e tal como no movimento linear aparece o mesmo fenómeno de erro ao longo do movimento; no entanto, ao contrário do movimento linear onde o movimento foge para um trajeto aparentemente aleatório, os movimentos circular fogem claramente na direção do centro da circunferência.

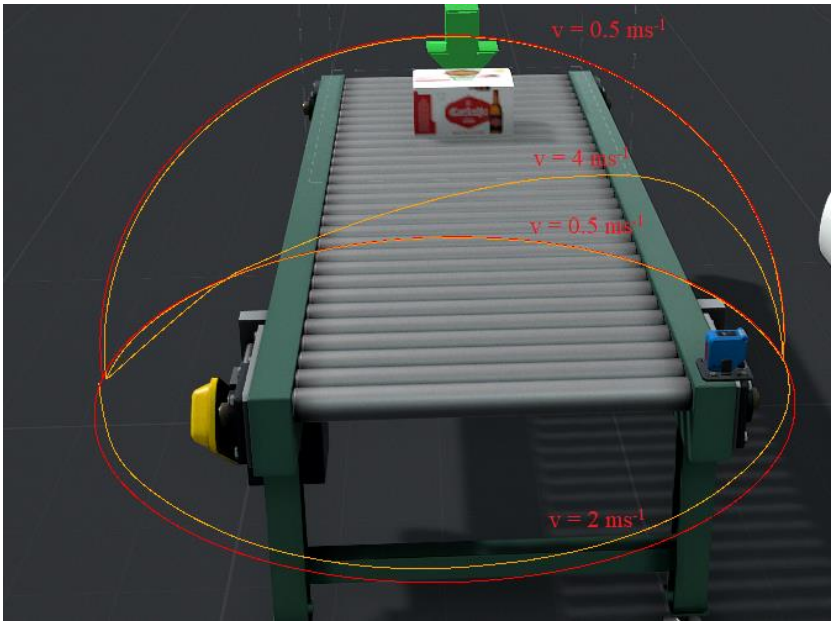

Figura 5.11 - Movimentos *move_c_center_all* (no plano horizontal) e *move_c_all* (na vertical) efetuados a diferentes velocidades e denotados a amarelo, as curvas desejadas encontram-se a vermelho

5.5 Ensaio Nº5 - Teste/desmonstração do mecanismo de suavização das descontinuidades da trajetória

Este ensaio destina-se a testar o efeito de diferentes erros de compromisso na suavização das trajetórias. Para tal, foram usados os métodos dos movimentos lineares e circulares em conjunto com o método *set_precision* capaz de alterar o valor do erro aceitável (precisão do movimento) nas descontinuidades.

Começando pelo movimento linear foi criado um programa que executa dois movimentos seguidos entre os pontos [1,0; 1,0; 1,0], [1,5; 1,0; 1,0] e [1,5; 1,0; 1,5], de forma a realizar um ângulo de 90º (uma descontinuidade) entre eles. A este programa adicionou-se depois o método *set_precision* para mudar o erro de compromisso para 0,1 m e 1,0 m. Os resultados obtidos encontram-se na Figura 5.12.

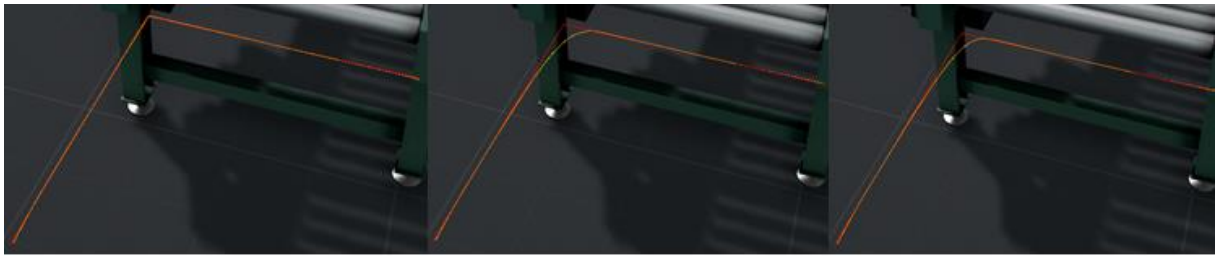

Figura 5.12 - Movimentos lineares com precisões 0,0; 0,1 e 1,0 m da esquerda para a direita (programa *move_linear.py*)

Na Figura 5.12, consegue-se reparar que, quando o erro de suavização desejado é diferente de 0, é efetivamente realizada uma curva entre os movimentos. Na Figura 5.13 é mostrada uma aproximação dessa curva onde são assinalados o ponto em que a precisão (erro) é respeitada e, consequentemente, a curva começa, assim como, a ordem de movimento que é executada para originar essa curva e relacionar os dois movimentos.

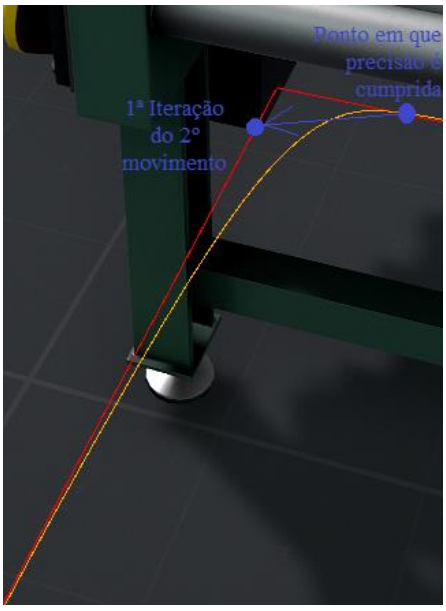

Figura 5.13 - Primeira ordem de movimento realizada após a precisão ser cumprida (entre o ponto em que a precisão é cumprida e o ponto final da 1ª iteração do segundo movimento)

Obviamente, tal como se observou na secção anterior isto vai inserir um erro na trajetória, mas, assumindo que a velocidade máxima do TCP é maior que a especificada no movimento, esse erro irá tornar-se cada vez menor a cada incremento e acabar por desaparecer rapidamente, como se observa na Figura 5.13 onde o robô se move a 0,5 ms^{-1}, velocidade que

já é considerável. No entanto, se o movimento estiver a ser realizado à velocidade máxima ou outra maior, este erro vai obviamente somar-se ao da velocidade.

Por outro lado, também se nota que com um erro de precisão de 0,1 e 1,0 m os resultados são idênticos. Isto acontece num robô real se a curva em que é possível manter a velocidade desejada (0,5 ms^{-1} neste caso) estiver a menos de 0,1 m do ponto final do primeiro movimento, sendo contraprodutivo realizá-la mais longe deste. No entanto, embora o RVP não realize esta verificação e ative o mecanismo de suavização assim que o erro é respeitado, este também está a ser limitado indiretamente pelo mecanismo de controlo de velocidade.

Mais precisamente, como o mecanismo de suavização das descontinuidades no RVP apenas é ativado após a iteração final do movimento anterior ser ordenada, e relembrando o mecanismo de velocidade apresentado na sub-secção 4.4.4, onde supostamente uma ordem tem tempo de ser feita antes da próxima ser ordenada, todas as iterações do 1º movimento menos a ultima já estarão realizadas aquando da ativação do método de suavização, e como tal, o erro máximo provocado por este é igual à distância ao ponto final nesse momento, ou seja, o cumprimento da ultima iteração. Por exemplo, na situação em causa, com a velocidade 0,5 ms^{-1} o cumprimento da última iteração e, logo, o erro de precisão máximo é 4,17 mm.

Assim, embora a transposição dos movimentos possa não ser feita na posição ótima, o facto de produzir um erro na ordem dos milímetros em vez do erro máximo aceitável (1 m) é uma mais valia.

De notar ainda, que se a velocidade especificada for maior que a velocidade máxima do robô, este não consegue cumprir as iterações anteriores e como tal o erro pode ser maior que o cumprimento da última iteração.

Quanto ao movimento circular, todas estas noções são exatamente iguais pois este também é feito a partir de pequenas iterações. Daí, apenas a título de exemplo decidiu-se correr novamente o programa *move_c_speed.py* usado no movimento anterior, mas a erros de compromisso diferentes, como se vê na Figura 5.14.

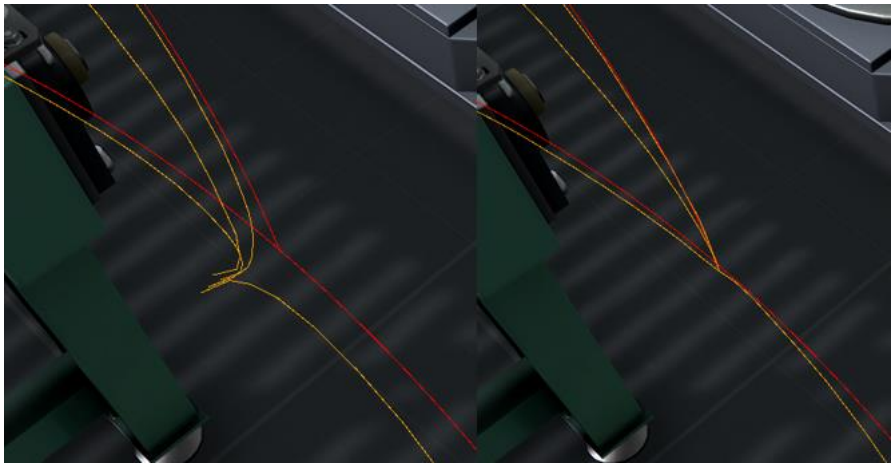

Figura 5.14 - Execução dos movimentos com diferentes precisões – 0,05 na esquerda e 0,00 na direita

De notar um interesse extra neste exemplo, pois um dos movimentos (o movimento de juntas responsável por colocar o robô na posição inicial dos movimentos circulares) não deve ser desenhado. No entanto, como a ordem que relaciona os movimentos é a primeira iteração executada no segundo movimento, ela pertence a este, e como tal é desenhada como parte do movimento circular, permitindo observar o alcance do erro de compromisso escolhido. De

notar ainda, que da mesma forma a iteração final dos movimentos circulares não será desenhada, como se vê na Figura 5.15.

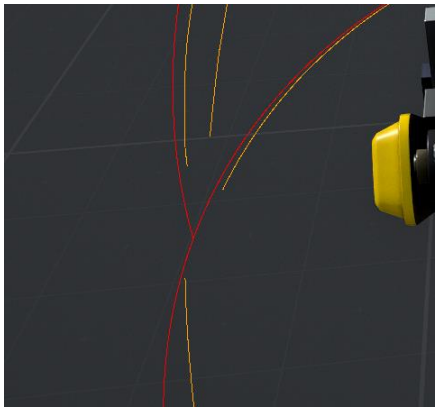

Figura 5.15 - Erro de representação do ponto final devido ao mecanismo de precisão

5.6 Ensaio Nº6 - Teste de mudanças de referenciais

Este ensaio destina-se a testar as possibilidades de criação e manipulação de referenciais instaladas. Para tal foi criado o programa *write_different_frames.py* que pretende escrever a letra "E" através do método *alphabet_e* em posições diferentes, sem alterar os seus dados de entrada (dois vetores e um ponto). Ou seja, esses vetores/ponto devem ser definidos como pertencentes a um referencial, e sempre que esse referencial se mover eles devem se mover junto, alterando a posição em que a letra é desenhada. O programa responsável por este processo apresenta-se na Figura 5.16.

```
import ik
import frame
import letters
from ik import *

trail_on()
set_speed(1)
clear_trail()
clear_debug_lines()
#Create frame
u=[1,0,0]
v=[0,1,0]
w=[0,0,1]

origin=[0,0,0]
frame.new_frame("a",origin,u,v,w)

print("Write letter in universal frame (without a given frame)")
v1=[0.1,0.1,0.2]
v2=[0.1,0.1,-0.2]
p= [0.5,0.8,1]
letters.alphabet_e(p,v1,v2)

print("Write letter in a given frame (universal in this case)")
v1=[0.1,0.1,0.2,"a"]
v2=[0.1,0.1,-0.2,"a"]
p= [0.5,0.8,1,"a"]
letters.alphabet_e(p,v1,v2)

print("Frame with diferent origin: the point changes position, the vectors are the same")
origin=[0,0,1]
frame.new_frame("a",origin,u,v,w)
letters.alphabet_e(p,v1,v2)

print("Frame with different vectors")
v=[0,0,1]
w=[0,1,0]
origin=[0,0,0]
frame.new_frame("a",origin,u,v,w)
letters.alphabet_e(p,v1,v2)

trail_off()
```

Figura 5.16 - Programa *write_different_frames.py*

Da execução deste programa resulta o ambiente da Figura 5.17, onde se observa que a letra "E" foi efetivamente desenhada em diferentes posições/orientações conforme desejado. De notar que, na Figura 5.17, a letra "E" apenas aprece três vezes, pois as primeiras duas vezes

em que é desenhada (sem especificar o referencial e com o referencial global) são iguais, como esperado.

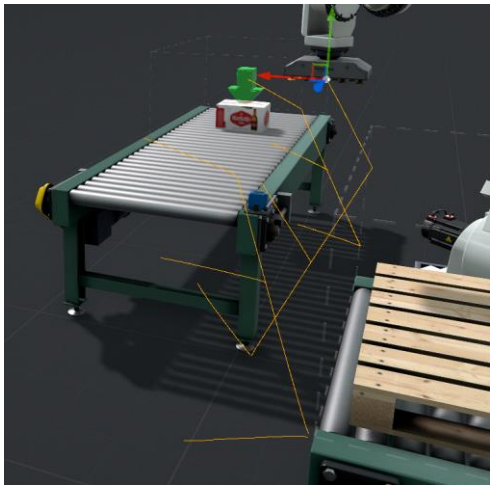

Figura 5.17 - Resultado da execução do programa *write_different_frames.py*

5.7 Ensaio Nº7 - Emulação de uma aplicação prática

Este ensaio destina-se a demonstrar um problema real: emular uma operação de paletização, assim como, testar a comunicação com o exterior, que ainda não havia sido experimentada. Para tal foi necessário estabelecer uma conexão MODBUS TCP/IP com o *softPLC* disponibilizado pelo *software* de desenvolvimento *Codesys 3.5*, de forma a controlar o ambiente externo ao robô (os tapetes transportados e os sensores de proximidade).

Assim, programou-se a lógica necessária ao movimento dos componentes existentes no ambiente do robô no *software Codesys 3.5* e definiu-se que devia haver duas variáveis de comunicação entre o robô e o *softPLC*:

- Uma variável binária alterável pelo *softPLC* que quando a *True* ordena o transporte da caixa entre a posição 1 da Figura 5.18 para a palete;
- Uma variável binária alterável pelo robô que quando a *True* informa o *softPLC* que a palete de deposição está cheia, de modo a que este proceda à respetiva evacuação.

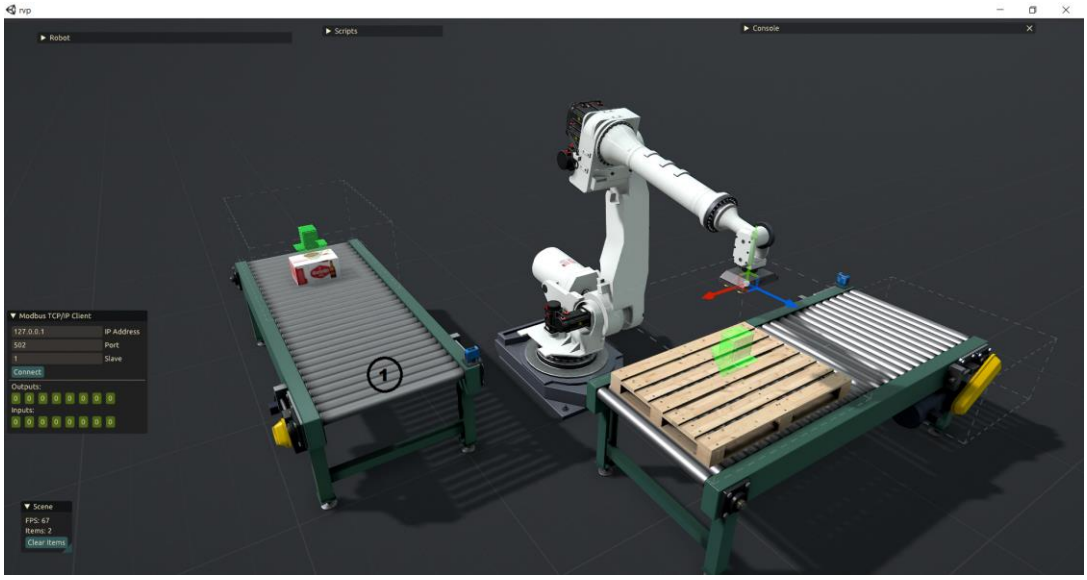

Figura 5.18 - Ambiente onde se vai realizar a operação de paletização

Quanto ao programa a executar pelo robô para esta tarefa, começou-se por definir a sequência de transporte necessária ao movimento de uma única caixa:

- Movimento de juntas para o ponto de aproximação à posição de recolha;
- Movimento linear para a posição de recolha;
- Ligar função de sucção;
- Movimento circular para o ponto de aproximação à posição de deposição;
- Colocar o objeto com a orientação de deposição desejada;
- Movimento linear para a posição de deposição;
- Desligar função de sucção;
- Movimento linear para o ponto de aproximação à posição de deposição.

De seguida, e tendo em conta as dimensões das caixas (0,3x0,2 m) e da palete (1,0x0,8 m) decidiu-se paletizar as caixas em três camadas da forma apresentada na Figura 5.19.

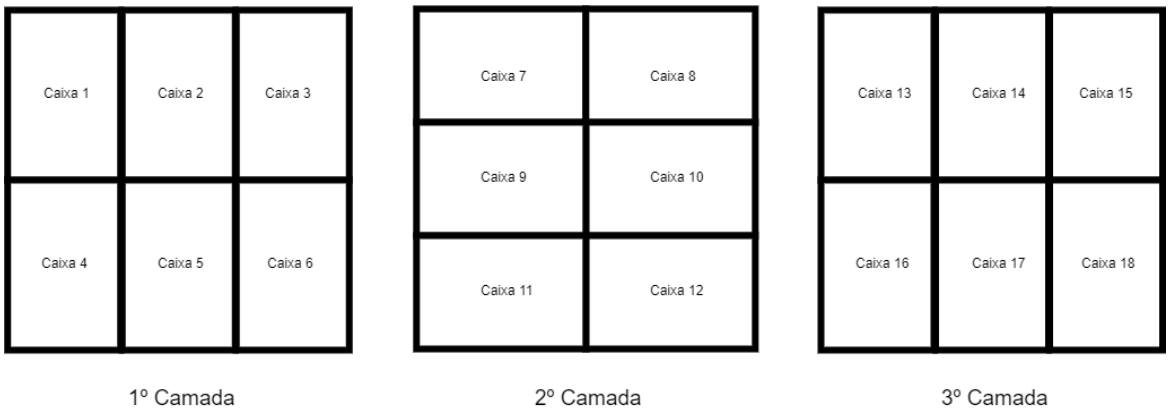

Figura 5.19 - Formato de deposição das caixas

Assim, expandiu-se a sequência de instruções anterior, ao colocá-la dentro de um ciclo que atravessa todas as posições de deposição desejadas ao incrementar alternadamente as três coordenadas cartesianas da primeira posição de deposição. De notar que nesta etapa foi ainda adicionado um intervalo de três milímetros entre as caixas de forma a compensar os erros de posição e repetibilidade observados anteriormente.

Daqui, ficou apenas a faltar incluir as variáveis a comunicar no programa. Para tal colocou-se este algoritmo dentro de um ciclo *while* interminável, que lê constantemente a variável responsável por colocar o robô em movimento (na posição *output* 2) através do método *modbus_client.get_output(2)*, realizando o transporte para a posição de deposição seguinte apenas quando este método responde *True*.

O programa pôde realizar a operação de paletização, faltando apenas informar o *PLC* quando esta acabar, de forma a ele movimentar a palete. Para isso, quando se chega à ultima posição de deposição é usado o método *modbus_client.set_input(2,1)* de forma a alterar o segundo *input* disponível para *True*. O *softPLC* consequentemente nota essa alteração e move a palete.

Assim é então possível colocar o programa, que foi chamado de *palletize.py*, em execução, estabelecer a conexão com o *softPLC* e observar o robô a paletizar as caixas como desejado, como se vê na Figura 5.20.

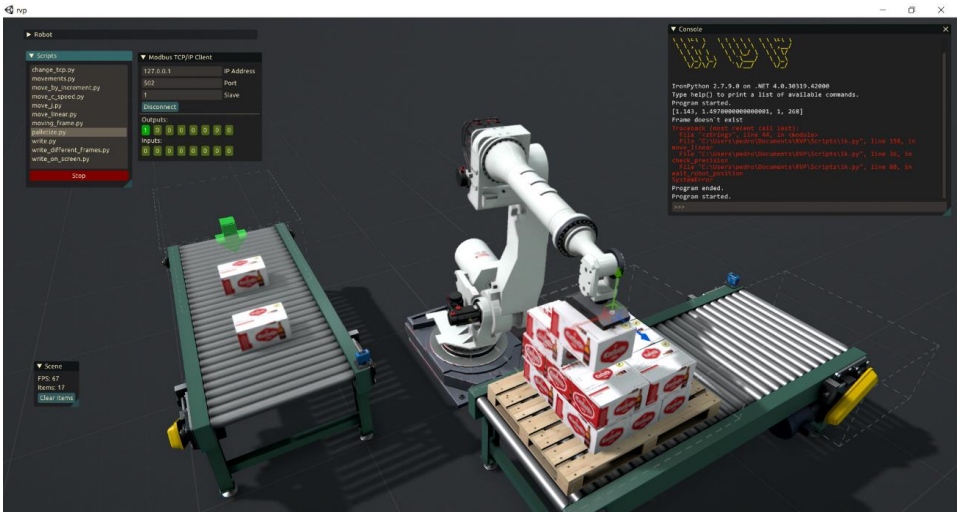

Figura 5.20 - Programa *palletize.py* em execução

Do teste do programa concluiu-se que ele funciona bem, tirando por um pequeno problema: a definição de um erro de apenas 3 mm foi demasiado otimista e, consequente, de vez em quando existe embate entre as caixas. No entanto, visto que este embate acontece muito raramente e apenas pode ser eliminado completamente com folgas à volta de 5 mm, que parecem um pouco mal numa operação de paletização, decidiu-se manter a folga nos 3 mm.

5.8 Apreciação global do desenvolvimento

Tendo em conta o descrito neste capítulo e as conclusões apresentadas na Tabela 5.3, acredita-se que o robô cumpriu as expectativas esperadas: consegue realizar de forma aceitável os três tipos de movimentos principais, assim como controlar as suas variáveis principais: velocidade e raio de precisão das descontinuidades da trajetória. Para além de fornecer uma funcionalidade secundária muito interessante: o controlo de referenciais.

Tabela 5.3 - Principais conclusões obtidas nos ensaios realizados

Ensaios Realizados	Principais Conclusões Obtidas
Teste/demonstração dos métodos de cinemática direta	Os movimentos são executados como desejado, sendo que os motores possuem um erro máximo de $5{,}0*10^{-2}$ º
Demonstração dos movimentos linear e circular	Os movimentos são executados de forma satisfatória permitindo a escrita de letras no ambiente
Avaliação da precisão e repetibilidade do robô	O robô possui um erro de precisão perto de 1,5 mm e um erro de repetibilidade perto de 1,0 mm. De notar que estes erros são uma ordem de grandeza superiores ao do robô real
Teste/demonstração do controlo de velocidade	O controlo de velocidade apenas funciona quando o robô opera a velocidades inferior à máxima. Quando a velocidade desejada é superior à velocidade máxima, aparece um erro na trajetória que vai aumentando ao longo do movimento.
Teste/demonstração do mecanismo de suavização das descontinuidades da trajetória	Este mecanismo funciona aceitavelmente, permitindo que o robô execute uma transição subtil entre os movimentos
Teste de mudanças de referenciais	A manipulação de referenciais funciona satisfatoriamente, permitindo mudar o local de execução de todo um programa facilmente
Emulação de uma aplicação prática	A operação de paletização foi emulada com sucesso, mas de vez em quando acontece um embate entre as caixas. Isto deve-se ao erro de posicionamento no limite do espaço de trabalho poder ultrapassar a folga programada (3 mm)

Assim, estas funcionalidades em conjunto com a linguagem *Python* 2.7, fornecem uma aplicação suficiente boa para executar a maioria das operações básicas de robótica, assim como possibilitam o desenvolvimento de algoritmos mais interessantes pelo utilizador como se viu aquando da criação da biblioteca *letters*.

No entanto ainda é possível melhorar e acrescentar algumas componentes/funções na área da interface gráfica e melhorar os métodos usados para o controlo da velocidade e movimentos, que como se viu possuem defeitos resultantes do modelo de cinemática inversa usado.

Para além disso, sem dúvida alguma, o erro linear da ordem dos milímetros apresenta-se como principal problema/falha desta implementação, sendo necessário testar mais variedades de constantes do controlador PID para tentar minimizá-lo ou desenvolver o algoritmo de cinemática inversa.

Por outro lado, também convêm referir os erros ocorridos na realização de trajetórias a grandes velocidades, que não aparecem noutras aplicações e devem ser corrigidos através de algum algoritmo capaz de limitar a velocidade. Assim como a falta de um método de identificação de colisões com a capacidade de parar o programa, que seria muito útil e interessante; no entanto, também reparar que nesta área das colisões, a presença do motor físico fornece uma grande vantagem ao RVP, pois permite representar essas colisões de forma extremamente fiel à realidade.

Assim, tendo tudo isto em conta, e relembrando que a maioria destes erros não são muito prejudiciais em situações de aprendizagem, achou-se que as expectativas postas nesta aplicação foram agradavelmente obedecidas.

5.9 Síntese

Neste capítulo foi exemplificado como usar as principais funcionalidades implementadas, servindo como um guia de utilização ao utilizador ao demonstrar as aplicações funcionais do ambiente, e concluindo que, qualquer utilizador com os conceitos básicos de linguagens de programação deve ser capaz de usar esta aplicação sem grandes dificuldades.

Para além disso, foram também denotados os principais problemas existentes, nomeadamente a repetibilidade do robô, que possui um erro linear da ordem dos milímetros, não sendo aceitável para muitas aplicações reais. Assim como os erros que podem aparecer da utilização de velocidades demasiado elevadas e as consequências de representação inerentes ao método usado para resolver o problema das descontinuidades da trajetória.

Por fim, foi ainda apresentado um exemplo real de uma aplicação de robótica mostrando as capacidades do RVP e a sua possibilidade de utilização no mercado.

6 Conclusão

No âmbito desta dissertação foi desenvolvido um ambiente computacional 3D centrado num robô virtual programável capaz de executar as funções mais importantes na introdução ao tema da robótica industrial, como os três tipos de movimentos básicos (juntas, linear e circular), o controlo da velocidade e precisão, manipulação de referenciais, entre outros.

6.1 Trabalho realizado e resultados atingidos

Inicialmente, foi detalhado todo o projeto construtivo subjacente, começando pela especificação dos requisitos funcionais e decisão das funcionalidades a incluir no projeto no Capítulo 2, seguindo depois para a aplicação dos aspetos de engenharia necessários ao desenvolvimento desse projeto no Capítulo 3 e culminando na criação dos métodos necessários à implementação das funcionalidades no Capítulo 4, ao qual se deu especial atenção, visto ser a parte de desenvolvimento prática da aplicação onde foram criadas as várias bibliotecas de métodos necessários à manipulação do robô, e que serão posteriormente usadas para criar programas.

Por fim, e como exemplo prático para testar as funcionalidades implementadas, foram ainda criados vários programas de forma a avaliar o desempenho de cada uma delas, assim como um programa que usasse todos essas funcionalidades para executar uma operação real, no caso uma aplicação de paletização que envolve a comunicação do robô com o ambiente através de um protocolo de MODBUS.

Destes testes concluiu-se que o robô desenvolvido possui um erro de posicionamento substancial e que os métodos criados para realizar os movimentos podem provocar erros na trajetória a velocidades elevadas; no entanto, deve-se notar que as velocidades que provocam este erro são inacessíveis pelo robô real. Para além disso, visto que o desejado era uma aplicação simples destinada a um ambiente didático e todas as outras funcionalidades instaladas funcionam aceitavelmente, considera-se que os objetivos estabelecidos foram cumpridos satisfatoriamente e que estes erros são, nesta fase do projeto, aceitáveis.

Por outro lado, quanto ao enriquecimento pessoal do autor, este foi um trabalho interessante, permitindo a obtenção de incontáveis conhecimentos na área da robótica industrial, assim como na programação em *Python*, área que suscita um interesse especial, também havia vontade de experimentar um pouco o software *Unity*; no entanto a falta de conhecimentos de programação mais desenvolvidos, assim como a falta de tempo para a sua aprendizagem, fez com que esta parte fosse executada inteiramente pela empresa.

Por outro lado, a empresa pôde obter uma primeira versão de um robô programável, que, após mais algum desenvolvimento, poderá vir a ser implementado no *Factory I/O* (um software de simulação de ambientes reais e treino de tecnologias relacionadas com a automação industrial). O que será extremamente útil permitindo a inclusão do RVP numa emulação fidedigna de um ambiente fabril e a sua interação com vários outros sistemas, sendo uma ferramenta poderosa que a empresa ganhará em relação à sua concorrência.

6.2 Trabalhos Futuros

Sendo este projeto a primeira versão da aplicação desejada existem obviamente diversos fatores que podem ser melhorados de forma a aproximar a aplicação desenvolvida das existentes no mercado, para além de diversos testes que ainda podem ser realizados.

Começando pelo problema do erro linear, devem ser feitos testes com o controlador PID de forma a ver se é possível obter valores dos controladores que produzam um erro menor, e talvez tentar relacionar os valores dos controladores a usar com a área de atuação do robô (perto ou longe da base por exemplo). Ou, por outro lado, ajustar o mecanismo de cinemática inversa responsável pela obtenção da posição. Neste tópico também seria deveras interessante trocar o algoritmo usado na cinemática inversa por um baseado no modelo matemático da cinemática do robô, como acontece nas outras aplicações. Isto permitiria realizar várias comparações interessantes entre o erro, capacidade de cálculo e velocidade de calculo dos algoritmos envolvidos, de forma a apurar mais claramente as vantagens de cada um na área da robótica. Para além disso, a aplicação de um desses algoritmos “tradicionais” também permitiria o desenvolvimento de novos métodos como o movimento de juntas síncrono e o melhoramento dos métodos usados no controlo da velocidade e na suavização das trajetórias.

De notar que estes métodos também podem ser melhorados com o algoritmo usado (*cyclic coordinate descent*), se este for aplicado de uma forma diferente, e, assim, permitir poder obter a posição final das juntas do robô à priori. Logo, efetuar as alterações necessárias para tal também é algo interessante.

Por outro lado, também será necessário desenvolver mais o projeto, incluindo, primeiramente, métodos capazes de detetar colisões, pois estes são indispensáveis em qualquer aplicação de robótica. Para além disso também se podem adicionar as funcionalidades mais redundantes que foram ignoradas nesta fase, como mostrar os limites do espaço de trabalho e mostrar/esconder objetos, que são sempre um extra bem recebido.

Por fim, deve-se também adicionar mais tipos de ferramentas para emular outros tipos de operações como soldadura e pintura, e avançar na direção de um ambiente de programação gráfico ao transformar os métodos implementados em botões.

Referências

Abreu, Paulo. 2018. *Apontamentos da Disciplina de Robótica Industrial*. Unidade Curricular "Robótica Industrial" do 5º Ano do Curso de Mestrado Integrado em Engenharia Mecânica, FEUP.

Almeida, Diego Varalda de. 2016. "Projeto Controle e Análise de um Manipulador Robótico Modular". Projeto de Graduação em Engenharia Mecânica, Universidade de Mogi das Cruzes, Brazil. https://www.docsity.com/pt/projeto-controle-e-analise-de-um-manipulador-robotico-modular/4891528/.

Anton, Stefan, Thomas Fries, Thomas Horsch, Friedrich Wilhelm Schoer, Cornelius Willnow e Christian Wolf. 2003. "A Framework for Realistic Robot Simulation and Visualisation". Available in: https://www.researchgate.net/publication/228827885_A_Framework_for_Realistic_Robot_Simulation_and_Visualisation.

Cardial, Victor, Henrique O'Neill, Rui Loureiro, A. A. Fernandes e Francisco Freitas. 1988. *Introdução à Robótica Industrial*. FEUP: Encadernação António Sousa Oliveira.

Couto, Carlos. 2000. *Introdução à robótica industrial*. Universidade Aberta: Graforim.

Craig, John J. 2009. *Introduction to robotics: mechanics and control*. 3ª ed.: Pearson Education India.

Engineer On A Disk. 2010. "Industrial Robotics". Acedido a 25 de Janeiro de 2020. http://engineeronadisk.com/V2/book_integration/engineeronadisk-14.html.

Henriques, Renato Ventura Bayan. 2002. "Programação e Simulação de Robôs". Em *Robótica industrial: aplicação na indústria de manufatufatura e de processos*, edição de Vitor Ferreira Romano: Edgard Blucher.

Kenwright, Ben. 2012. "Inverse kinematics–cyclic coordinate descent (CCD)". *Journal of Graphics Tools* 16, no. 4: 177-217.

Lapham, John. 1999. "RobotScript™: the introduction of a universal robot programming language". *Industrial Robot: An International Journal* 26, no. 1: 17-25.

Magalhães, António Pessoa de, Bernard Riera e Bruno Vigário. 2010. "When Control Education is the name of the game". Em *Business, Technological and Social Dimensions of Computer Games: Multidisciplinary Developments*, edição de Maria Manuela Cruz-Cunha, Vítor Hugo Carvalho e Paula Tavares: IGI Global.

Microsoft. 2020. "Visual Studio Code". Acedido a 25 de Janeiro de 2020. https://code.visualstudio.com/.

Mishra, Akhilesh Kumar e Oscar Meruvia-Pastor. 2014. "Robot arm manipulation using depth-sensing cameras and inverse kinematics". Comunicação apresentada em 2014 Oceans-St. John's.

NVIDIA. 2019. "PhysX SDK". Acedido a 25 de Janeiro de 2020. https://developer.nvidia.com/physx-sdk.

OMRON. 2019. "Robôs Colaborativos". Acedido a 25 de Janeiro de 2020. https://industrial.omron.pt/pt/products/collaborative-robots?utm_source=newsletter&utm_medium=mail&utm_campaign=2019-07-newsletter&elqTrackId=e53976b318cc4836be8305edc0382d77&elq=ea4d1e115e5547579c5b1030e4b73eca&elqaid=2967&elqat=1&elqCampaignId=3072#videos.

Pires, J Norberto. 2012. *Automação Industrial*. 5ª ed.: ETEP - Edições Técnicas e Personalizadas.

Python Software Foundation. 2020. "Python". Acedido a 25 de Janeiro de 2020. https://www.python.org/.

Real Games. 2019a. "About Factory I/O". Acedido a 25 de Janeiro de 2020. https://factoryio.com/docs/.

———. 2019b. "Real Games". Acedido a 25 de Janeiro de 2020. https://realgames.co/.

RoboDK. 2019a. "Post Processors". Acedido a 25 de Janeiro de 2020. https://robodk.com/doc/en/Post-Processors.html.

———. 2019b. "Robot Programs". Acedido a 25 de Janeiro de 2020. https://robodk.com/doc/en/Robot-Programs.html#InsSmooth.

Unity Technologies. 2019. "Unity". Acedido a 25 de Janeiro de 2020. https://unity.com/.

Visual Components. 2019. "Visual Components". Acedido a 25 de Janeiro de 2020. https://www.visualcomponents.com/.