Mestrado em Engenharia Eletrotécnica e de Computadores

Roller Coaster

Robótica Avançada

Grupo I

Ricardo Rodrigues nº11611
José Rodrigues nº10227
Rui Carvalho nº12634
Filipe Rodrigues nº12560

4 de Março de 2020

Resumo

Este trabalho prático foi realizado no âmbito da disciplina de Automação avançada, lecionada no curso de Mestrado em Eletrónica e Computadores, no Instituto Politécnico do Cávado e do Ave. O objetivo principal foi o desenvolvimento de um ambiente de jogo assemelhando-se a uma montanha russa e com simulação por parte do robot Kuka. Foram desenvolvidas funções capazes de passar para esse cenário todos os movimentos gerados pelo robot dentro das limitações cinemáticas do mesmo. A comunicação com o robot também é um dos aspetos preponderantes descritos neste relatório. Por fim é apresentada uma conclusão que evidência as dificuldades encontradas na realização do trabalho proposto.

Palavras Chave (Tema): Montanha russa, simulação, realidade virtual

Palavras Chave (Tecnologias): *Software Unity*, *Visual Studio*, *Orange Edit*, *Oculus DK2*

Índice

Índice de Figuras

1 Introdução

Neste capítulo é apresentado um breve enquadramento do trabalho realizado, assim como os objetivos propostos para a realização do mesmo. Em simultâneo, são também descritas as tecnologias que foram necessárias para a sua realização, bem como os contributos académicos resultantes da execução deste trabalho.

1.1 Enquadramento

Este trabalho foi desenvolvido no âmbito da disciplina de robótica avançada e tem como objetivo, fazer a simulação de uma montanha russa através de um robot KUKA. Pretende-se futuramente um cenário, representado pela Figura 1, que permita ao utilizador escolher um percurso de montanha russa ao seu agrado, sendo enviado a partir do computador toda essa informação para o robot.

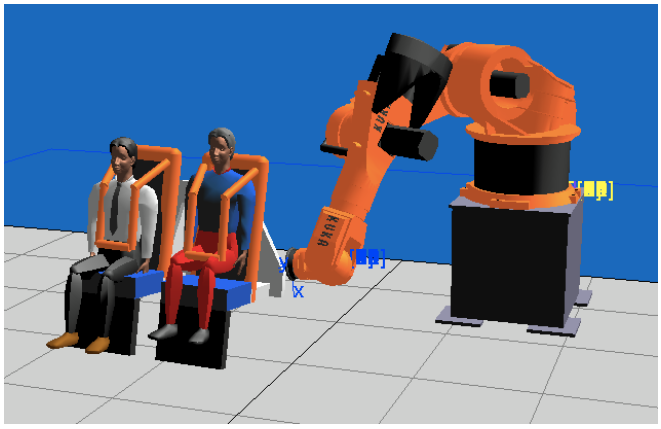

Figura 1-Cenário final pretendido.

O utilizador estando sentado na cadeira, poderá dar a ordem para o robot iniciar a simulação e, simultaneamente, visualizar o percurso através de uns de realidade virtual.

O ambiente gráfico foi desenvolvido através do software Unity e apresentado ao utilizador através desses óculos da marca Oculus. O objetivo principal é simular os movimentos da pista da montanha russa através do robot, sendo esses movimentos originados pela pista criada no software *Unity*.

Numa primeira fase, são adquiridos pontos correspondentes a posições físicas do robot dentro dos limites dos próprios eixos, evitando deste modo as singularidades que eventualmente possam ocorrer. Posto isto, no *software Unity* são lidos esses pontos que seguidamente são utilizados na formulação da pista da montanha russa. A geração do trajeto, tem como base métodos do *software Unity* para a construção de jogos e cenários.

1.2 Tecnologias utilizadas

Para a realização deste trabalho foram indispensáveis alguns softwares nomeadamente o Unity IDE, OrangeEdit IDE, Qt Creator IDE, Visual Studio Community.

1.3 Contributos deste trabalho

Este trabalho prático permitiu adquirir conhecimento de programação do robô Kuka, assim como o estabelecimento da comunicação do motor de jogo Unity com o próprio robô.

1.4 Organização do relatório

Este relatório encontra-se divido em 3 capítulos. O capítulo 1 pertence à introdução onde está descrito o enquadramento do trabalho, assim como as tecnologias utilizadas e os contributos que o mesmo proporcionou. No capítulo 2 apresenta-se toda a descrição do trabalho realizado maioritariamente no *software Unity,* no que aponta para o desenvolvimento das funções do programa. Por último, no capítulo 3 é apresentada a conclusão do trabalho realizado.

2 Funções principais

Neste capítulo está representado todo o sistema desenvolvido para estabelecer um sistema de comunicação entre o nosso sistema de simulação e o robô KUKA.

2.1 Comunicação Unity - Robô

Para estabelecer a comunicação com o robô KUKA, utilizou-se a biblioteca open-source JOpenShowVar. Também foi necessário desenvolver no nosso IDE um método de comunicação com esta plataforma, devido à dll (Dynamic link library) disponibilizada pelo professor não ser compatível com o IDE utilizado. A dll foi desenvolvida com recursos a componentes da framework Qt, sendo esta incompatível com a plataforma Unity que foi utilizada para a realização do projeto.

Deste modo, foi necessário desenvolver uma classe que foi denominada de "Client", responsável por criar um cliente TCP e gerir toda a comunicação com o servidor.

Além das funcionalidades clássicas de classes para comunicação como o "connect" e "disconect", esta tem de codificar e descodificar transmissões segundo a API (Application programming interface) que o servidor utiliza.

Em seguida são apresentadas as principais funções da classe Client:

Na imagem seguinte é apresentada a função *Connect()* que permite estabelecer uma comunicação para o servidor KUKA. A função começa por instanciar um objeto csv para armazenar os dados referentes as posições do robot. Em seguida é inicializada a comunicação.

```
public void Connect()
{
    writer = new CSVWriter();
    Debug.Log("TryConnect");
    this._socket = new TcpClient();
    receiveBuffer = new byte[dataBufferSize];
    this._socket.BeginConnect(System.Net.IPAddress.Parse(ip), port, ConnectCallback, this._socket);

}
```

Figura 2-Função connect() da classe Client.

A próxima função representada na imagem seguinte, começa por verificar se existe comunicação com o robot e caso exista, os dados são recebidos e armazenados num buffer.

```
private void ConnectCallback(IAsyncResult ia)
{
    this._socket.EndConnect(ia);    //
    if (!this._socket.Connected)
    {
        Debug.Log("Unnable to Connect!");
        return;
    }
    this._socket.NoDelay = true;    //parameter to accelerate transmission
    readStream = this._socket.GetStream();
    Debug.Log("Connected!");
    readStream.BeginRead(receiveBuffer, 0, dataBufferSize, ReceiveCallback, null);
}
```

Figura 3-Função ConnectCallback da classe Client.

A função ReceiveCallback() permite armazenar os dados vindos do robot num buffer. A função começa por analisar o tamanho da variável readStream para verificar se existe dados. Caso existam dados no buffer, estes são armazenados.

```
private void ReceiveCallback(IAsyncResult ia)
{
    try
    {
        int _byteLenght = readStream.EndRead(ia);
        if (_byteLenght <= 0)
        {
            //TODO: diconnect
            Disconnect();
        }

        byte[] _data = new byte[_byteLenght];
        Array.Copy(receiveBuffer, _data, _byteLenght);
        //string newMessage = System.Text.Encoding.UTF8.GetString(_data, 0, _byteLenght);
        Debug.Log("Received data: ");
        byte[] mesRes = new byte[_data.Length];
        System.Buffer.BlockCopy(_data, 8, mesRes, 0, _data.Length-8);

        //string newMessage = System.Text.Encoding.Default.GetString(mesRes);
        Debug.Log("Message: " + Encoding.ASCII.GetString(mesRes) + " END message!");

        if (AStatus.sceneNumber == 1)
        {
            string saved = Encoding.ASCII.GetString(mesRes);
            writer.addRecord(saved);
        }
        readStream.BeginRead(receiveBuffer, 0, dataBufferSize, ReceiveCallback, null);
    }
    catch
    {
        //TODO: disconnect
        Disconnect();
    }
}
```

Figura 4-Função ReceiveCallback() da classe Client.

A próxima função, *Send()*, tem como finalidade enviar dados para o robot. A principal utilidade desta função será comandar o robot através da aplicação.

```
public void Send(string var,  int idMsg)
{
    Debug.Log("startSending");
    int lunghezza, varNameLen;
    byte var_hByte, var_lByte, msg_hByte, msg_lByte;
    byte hByteMsg, lByteMsg;
    // Message ID ( MAX: 0xFFFF )
    hByteMsg = Byte.Parse(((idMsg & 0xff00) >> 8).ToString());
    lByteMsg = Byte.Parse((idMsg & 0x00ff).ToString());
    //var length
    varNameLen = var.Length;
    var_hByte =  Byte.Parse(((varNameLen & 0xff00) >> 8).ToString());
    var_lByte = Byte.Parse((varNameLen & 0x00ff).ToString());

    lunghezza = 2 + 1 + varNameLen;
    msg_hByte = Byte.Parse(((lunghezza & 0xff00) >> 8).ToString());
    msg_lByte = Byte.Parse((lunghezza & 0x00ff).ToString());
    byte[] buffer =  {
        hByteMsg, //message ID
        lByteMsg, //message ID
        msg_hByte, //MSG length
        msg_lByte, //MSG length
        0x00, //0 - read, 1 - write
        var_hByte, //next var length
        var_lByte, //next var length
        };
    byte[] b= System.Text.Encoding.UTF8.GetBytes(var);
    byte[] finalMess = new byte[buffer.Length + b.Length];
    System.Buffer.BlockCopy(buffer, 0, finalMess, 0, buffer.Length);
    System.Buffer.BlockCopy(b, 0, finalMess, buffer.Length, b.Length);
    if (!this._socket.Connected)
    {
        Debug.Log("SocketnotConnected!");
        return;
    }
  readStream.BeginWrite(finalMess, 0, finalMess.Length, WriteCallback, null);
}
```

Figura 5-Função Send da classe Client.

A configuração inicial quando o cliente inicia a transmissão é apresentada pela Figura 5.

```
byte[] buffer =  {
    hByteMsg, //message ID
    lByteMsg, //message ID
    msg_hByte, //MSG length
    msg_lByte, //MSG length
    0x00, //0 - read, 1 - write
    var_hByte, //next var length
    var_lByte, //next var length
    };
```

Figura 6-Variável tipo byte para estabelecer a comunicação.

Após esta estrutura estar definida é acrescentado os bytes referentes à variável a ler/escrever e também o novo valor da variável caso seja uma operação de escrita.

Foi decidido utilizar um sistema assíncrono de comunicação realizado por callback de forma a que o sistema não fique comprometido com as tarefas de comunicação.

2.2 Aquisição de pontos do robô KuKa

Ainda na fase embrionária do projeto, tendo em consideração as limitações que poderiam existir no que se refere às restrições dos eixos do robô Kuka, optou-se pela aquisição de pontos em tempo real correspondentes a movimentos dentro de limites de controlo do Kuka.

Numa primeira abordagem os pontos adquiridos foram conseguidos a partir de uma *thread* executada em QT (esta funcionalidade foi mais tarde implementada dentro da aplicação do Unity) que faz a aquisição de 50 pontos com uma frequência de 1 segundo, sendo posteriormente armazenados num ficheiro com extensão CSV (Comma-separated values). Seguidamente, no software Unity é carregado o ficheiro pela aplicação e consequentemente percorrido linha a linha, retirando os valores pretendidos para uma lista, onde é feito um tratamento de dados devido a particularidades da linguagem C#. No seguimento deste procedimento, é instanciada uma lista de vetores com 3 posições, que tem como finalidade armazenar as posições X, Y, Z para a criação do ponto no espaço, como os eixos Y e Z do robô estão invertidos em relação aos eixos do ambiente do unity é necessário guardar os dados de Y como Z e vice-versa.

Na seguinte imagem é apresentada a função para adicionar um novo ponto do robot ao ficheiro.

```
public void addRecord(string _data)
{
    try
    {
        StreamWriter writer = new StreamWriter(@filePath, true);
        writer.WriteLine(_data);
        writer.Close();

    }
    catch
    {
        Debug.Log("Unnable to WriteFile!");
    }
}
```

Figura 7-Função addRecord() da classe CSVWritter.

Para a criação de um novo ficheiro para armazenar os pontos, existe a função createNewFile(), apresentada na seguinte Figura.

```
public void createNewFile()
{

    try
    {
        StreamWriter writer = new StreamWriter(@filePath, false);
        //writer.WriteLine("");
        writer.Close();

    }
    catch
    {
        Debug.Log("Unnable to WriteFile!");
    }
}
```

Figura 8-Função createNewFile() da classe CSVWritter.

2.3 Criação da montanha russa

Após o ponto estar criado no vetor "pointPosition" é invocado o método "InsertNewPointAt2" da classe "spline" para inserir cada ponto na *spline* sequencialmente. Posteriormente é chamado o método "AutoConstructSpline()" que otimiza os valores da spline para que se possível esta não adquire vértices.

```
            spline.AutoConstructSpline();


            vr.BuildRollercoasterTrack(GameObject.Find("Track"),
            GameObject.Find("BezierSpline"), leftRailPrefab, rightRailPrefab,
            crossBeamPrefab, resolution);
            cart = GameObject.Find("Rollercoaster");
            cart.transform.Translate(0, 3, 0);
            return true;
        }
        catch
        {
            return false;
        }
    }
```

Finalmente é requisitado o método "BuildRollercoasterTrack" responsável pela construção da *spline*.

3 Aplicação

O controlo da aquisição dos pontos no que diz respeito ao comando enviado ao robot para começar essa tarefa, assim como o comando para iniciar o movimento correspondente, são executados através de uma aplicação criada no *Unity* .

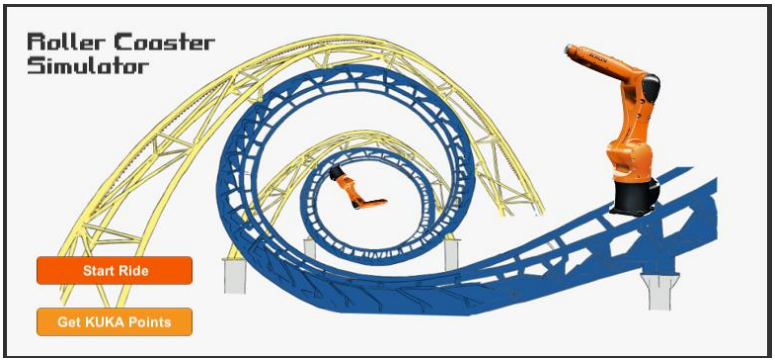

Figura 9-Menu inicial no software Unity.

Na Figura 9 está representado o menu inicial da aplicação onde o utilizador tem disponíveis duas opções o bitão "Get KUKA Points" irá enviar uma ordem para o robô iniciar o seu movimento também começará a aquisição de novos pontos para a criação ou atualização do percurso da montanha russa.

Por sua vez o botão “Start Ride” o utilizador irá ser transportado para o carro da montanha russa e após pressionar a tecla “D” do teclado a viagem irá iniciar. Como mostra na Figura 10

Figura 10- Montanha Russa no software Unity

3.1 Aplicação Robô

A aplicação do robô é relativamente simples e consiste num ciclo que espera um comando externo para iniciar a execução de uma spline pré-programada no controlador do Robô, em que todos os movimentos que irão ser efetuados por este tenham de ser inseridos manualmente de forma evitar o acontecimento de singularidades, como se pode verificar na Figura 11.

Após a execução de todos os movimentos este irá esperar por um novo comando de reinício.

O programa do robô irá ser executado da mesma forma cas se pretenda adquirir pontos para o desenho da montanha russa ou caso esta já esteja desenhada e esteja a simular o percurso.

```
DEF t1( )
INI

PTP rollerHOME CONT Vel=10 % DEFAULT Tool[0] Base[0]
LOOP

WAIT FOR startFun
SPLINE S1 CONT Vel=0.03 m/s CPDAT1 Tool[0] Base[0]

SPL RC_P1
SPL RC_P2
SPL RC_P3
SPL RC_P4
SPL RC_P5
SPL RC_P6
SPL RC_P7


PTP rollerHOME Vel=10 % DEFAULT Tool[0] Base[0]
startFun=false
ENDLOOP
END
```

Figura 11- Programa do robô Kuka

4 Conclusões

A realização deste trabalho prático que integrou o robot KUKA com o software Unity, permitiu a junção de duas tecnologias diferentes. Contudo, existiram várias dificuldades na compreensão da programação no software *Unity*. A estruturação do código de programação é bastante diferente da que habitualmente estamos familiarizados, apesar do desenvolvimento das *scrips* serem feitas em C #. Foi devido a isto, que grande parte do tempo inicialmente despendido neste tralhado, foi gasto no entendimento de como tudo se processa no software Unity. Relativamente aos objetivos declarados inicialmente para o trabalho, ficaram ainda alguns tópicos por concretizar, nomeadamente o envio sincronizado dos pontos para o robot, entre outros aspetos. De qualquer forma, fica patente, que é possível e viável esta interação do robot Kuka com o *software Unity* e, com mais tempo aplicado a desenvolvimento, podem ser obtidos resultados mais satisfatórios.

Bibliografia

[1] Slides Moodle, Programming Industrial Robots, https://elearning.ipca.pt/1920/pluginfile.php/431843/mod_resource/content/0/Programming%20of%20industrial%20robots.pdf , consultado a 05-02-2020

[2] Slides Moodle, Systems variables for Kuka System Software, https://elearning.ipca.pt/1920/pluginfile.php/431846/mod_resource/content/0/KSS_81_82_83_System_Variables_en.pdf , consultado a 14-02-2020