



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
NÚCLEO DE PESQUISA E INOVAÇÃO EM TECNOLOGIA DA INFORMAÇÃO (NPITI)
LABORATÓRIO DE TECNOLOGIAS EDUCACIONAIS, ASSISTIVAS E MULTIMÍDIA (TEAM)

Relatório do Projeto Runners (2024)

Gustavo Henrique Lima de Araújo

Natal, 15 de fevereiro de 2025

Lista de Imagens

Figura 1 – Conexão ESP32-sensor	4
Figura 2 – Página de Download do <i>EclipseMosquitto</i>	12
Figura 3 – Instalação <i>Mosquitto</i> passo 1	12
Figura 4 – Instalação <i>Mosquitto</i> passo 2	13
Figura 5 – Instalação <i>Mosquitto</i> passo 3	13
Figura 6 – Instalação <i>Mosquitto</i> passo 4	14
Figura 7 – Propriedades do Sistema	14
Figura 8 – Variáveis de Ambiente	15
Figura 9 – Editar a variável de ambiente	15
Figura 10 – Comando <i>mosquitto -h</i>	16
Figura 11 – Arquivo <i>mosquitto.conf</i>	16
Figura 12 – Comandos no arquivo <i>mosquitto.conf</i>	17
Figura 13 – Iniciando <i>Mosquitto</i>	17
Figura 14 – Bibliotecas de Assets Utilizadas no Projeto	18
Figura 15 – <i>Main Camera</i> no jogo	19
Figura 16 – <i>"bg - back"</i> no jogo	19
Figura 17 – <i>BackgroundAssets</i> no jogo	20
Figura 18 – Elementos de <i>BackgroundAssets</i> no jogo	21
Figura 19 – <i>BackgroundController</i> no jogo	21
Figura 20 – <i>vehicle - 2</i> no jogo	26
Figura 21 – <i>Canva</i> no jogo	28
Figura 22 – <i>Border e Fill</i>	29
Figura 23 – <i>SpeedBar</i> no jogo	30
Figura 24 – <i>Timer</i> no jogo	31
Figura 25 – Configurações do <i>Timer</i>	31
Figura 26 – <i>MqttClient</i> no Inspetor	34
Figura 27 – Inspetor Parte 3	34

Sumário

1	INTRODUÇÃO	3
2	PROJETO	4
2.1	ESP32	4
2.2	Servidor	12
2.3	Unity	18
	Bibliografia	35

1 INTRODUÇÃO

Esse relatório tem o principal objetivo de apresentar tudo o que foi feito dentro do projeto REMO, mostrar como o projeto e seus vários elementos estão postos e como ele interagem entre si.

Pelo projeto apresentar uma certa complexidade, o arranjo das informações presentes nesse relatório será dividido entre os diferentes elementos do projeto, esses sendo: o ESP32, o servidor MQTT e a aplicação dentro da Unity. Isso foi feito para facilitar a leitura e compreensão desse relatório, também para que, em caso de perda dos arquivos do projeto, as mesmas possam ser facilmente recriadas.

O projeto feito foi baseado no GDD presente no seguinte link [aqui](#).^[1].

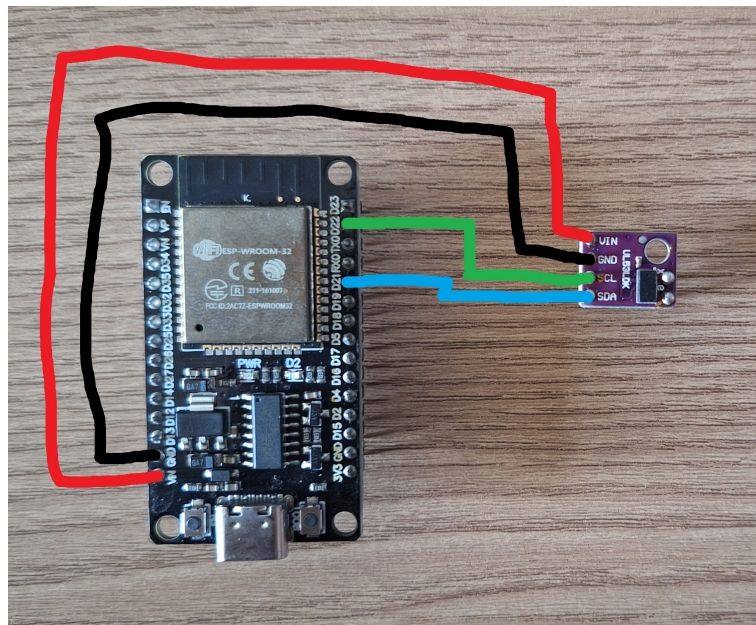
2 PROJETO

2.1 ESP32

O *ESP32* tem a principal função de capturar o movimento da máquina, utilizando sensores, traduzir essa informação para o jogo e então enviar através de algum método de comunicação, nesse caso *MQTT*.

O código foi escrito utilizando a IDE do Arduino, e na parte física o sensor é conectado seguindo o seguinte diagrama

Figura 1 – Conexão ESP32-sensor



Esse agrupamento foi escolhido pois a biblioteca escolhida para realizar a interface com o sensor de forma serial, faz uso de *I2C* então é utilizado as portas de comunicação *I2C*, essas sendo a *D22* e *D21*. Fora isso, o sensor é alimentado pela porta *Vin* e se conecta ao *GND* do *ESP32* pela porta de mesmo nome.

Para a parte de código serão usadas 3 bibliotecas, a primeira delas é chamada de *Adafruit_VL53L0X.h* essa biblioteca é a responsável por se comunicar diretamente com o sensor usando as portas de *I2C*, dessa biblioteca são usados os comandos para inicializar o sensor, definir seu método de captura de dados, e ler os valores que o mesmo está lendo. Além dela, também são usadas as bibliotecas *WiFi.h* e *PubSubClient.h*, essas responsáveis pela parte de comunicação do *ESP32* com o servidor, onde *WiFi.h* é utilizada para estabelecer uma conexão com uma rede sem fio e *PubSubClient.h* é utilizado para se conectar ao servidor *MQTT* bem como para publicar e receber informações.

```
//---Bibliotecas---//
#include "Adafruit_VL53L0X.h" //Biblioteca de comunicação com o sensor VL53L0X
#include <WiFi.h>              //Biblioteca para conexão Wifi
#include <PubSubClient.h>      //Biblioteca para conexão com servidor MQTT
```

Após importar as bibliotecas, o próximo passo no código é definir alguns *MACROS* que serão usados no decorrer do código principalmente relacionado as funcionalidades de cada biblioteca, isso é feito para que, em casos onde é necessário editar alguma informação nessas variáveis, não seja necessários procurar todas aparições dessas variáveis dentro do código e editá-las de forma individual, também porque algumas bibliotecas exigem que seus componentes sejam declarados antes de iniciar o código.

```
//---Macros---//
```

```
//--Informações da rede--//
```

```
const char* ssid = "Teste";    //Domínio de Wifi
const char* pass = "12345678"; //Senha do Wifi
```

```
//--Informações para o servidor MQTT--//
```

```
const char* server = "192.168.41.226"; //Ip da máquina com servidor MQTT
const int mqttPort = 1883;             //Porta de rede MQTT
const char* username = "HjYXDTCQNhY20zQcKx0YGB0"; //Usuário do dispositivo MQTT
const char* password = "hChdo9lTnAi4w1hTv2iz0zDg"; //Senha do dispositivo MQTT
const char* channelIdSub = "Remo_Esp"; //Canal de comunicação Unity->Esp
const char* channelIdPub = "Esp_Remo"; //Canal de comunicação Esp->Unity
```

```
//--Objetos das Bibliotecas--//
```

```
Adafruit_VL53L0X distSensor = Adafruit_VL53L0X(); //Objeto do sensor
WiFiClient net;                                     //Cliente do Wifi
PubSubClient client(net);                           //Cliente MQTT
```

Existem outras variáveis que foram configuradas como *Macros*, mas elas são atreladas as funções implementadas no *ESP32* e serão explicadas conforme forem aparecendo.

Com os macros declarados, foram implementados algumas funções necessárias antes das funções de *setup* e *loop* que serão usadas para realizar a conexão e reconexão com o *Wifi* e o servidor *MQTT*. O algoritmo consistem em um loop que tenta se conectar e persiste enquanto não houver uma conexão, essa lógica foi adaptada dos exemplos presentes nas bibliotecas com algumas alterações.

```
//---Funções---//
```

```
//--Conecta ao Wifi--//
void conectaWifi(){
    Serial.println("Tentando Conectar ao WiFi");
    while(WiFi.status() != WL_CONNECTED){
        WiFi.begin(ssid, pass);
        Serial.print('.');
        delay(5000);
    }
    Serial.println("");
    Serial.println("Conectado ao WiFi");
}

//--Conecta o cliente ao MQTT--//
void conectaMQTT(){
    if(WiFi.status() != WL_CONNECTED){
        conectaWifi();
    }
    Serial.println("Tentando Conectar ao MQTT");
    while (!client.connected()) {
        String client_id = "esp32-client-";
        client_id += String(WiFi.macAddress());
        Serial.printf("The client %s connects to the public MQTT broker\n",
            client_id.c_str());
        if (client.connect(client_id.c_str(), username, password)) {
            Serial.println("Public EMQX MQTT broker connected");
        } else {
            Serial.print("failed with state ");
            Serial.print(client.state());
            delay(2000);
        }
    }
    Serial.println("");
    Serial.println("Conectado ao MQTT");
}
```

Além dessas funções de conexão, a biblioteca *PubSubClient.h* requer a implementação de uma função de *callback* que é chamada todas as vezes que o *ESP32* receber uma mensagem pelo canal *MQTT* onde ele está inscrito. O objeto que representa, o cliente

MQTT, irá receber a função de callback como argumento quando ele for inicializado. Como o objetivo dessa função é tratar o recebimento de mensagens ela foi deixada bem básica, uma vez que no momento o jogo em *Unity* não está enviando dados para o *ESP32*.

```
//--Mensagem Recebida--//
void callback(char *topic, byte *payload, unsigned int length) {
    Serial.print("Message arrived in topic: ");
    Serial.println(topic);
    Serial.print("Message:");
    for (int i = 0; i < length; i++) {
        Serial.print((char) payload[i]);
    }
    Serial.println();
    Serial.println("-----");
}
```

Existe também outra função implementada para que, conectado ao *broker MQTT*, o *ESP32* possa se inscrever em um canal para receber mensagens, porém esse código também não é utilizado, o motivo sendo, uma vez que o jogo não está enviando dados para o *ESP32*, não existe motivo para utilizar essa função, sua implementação existe majoritariamente para verificar se o *broker MQTT* está funcionando de forma ideal ou não.

```
//--Se inscreve no canal--//
void subscribeToChannel(){
    if(!client.connected()){
        conectaMQTT();
    }
    Serial.print("Tentando se inscrever no canal");
    Serial.println(channelIdSub);
    while(!client.subscribe(channelIdSub)){
        Serial.print(".");
        delay(1000);
    }
    Serial.println("");
    Serial.print("Inscrito ao canal: ");
    Serial.println(channelIdSub);
}
```

Por último, mas não menos essencial, foi criado uma função para se conectar ao

sensor, nessa função é necessário também definir o modo de funcionamento que o sensor opera, nesse caso será o modo de medição contínua.

```
//--Conectar sensor--//
void conectaSensor(){
    Serial.println("Tentando Conectar ao Sensor");
    while (!distSensor.begin()) {
        Serial.print(F("."));
        delay(1000);
    }
    Serial.println("");
    Serial.println("Conectado ao sensor");
    distSensor.startRangeContinuous();
}
```

Agora, para as funções essenciais do Arduino, *setup* e *loop*, explicando primeiro o *setup*, primeiro é definido o canal de comunicação serial com o computador para *debug* e verificação das funcionalidades do *ESP32* na fase de testes. Em seguida, é realizado a conexão com o *Wifi*, pela função *begin()* nativa da biblioteca e a função *conectaWifi()*. Em seguida é informado o servidor *MQTT* pela função *setServer()* e definido a função de *callback* utilizando *setCallback()*. Após isso, o último passo se torna realizar a conexão com o sensor utilizando *conectaSensor()*.

```
//---Setup---//
void setup() {
    Serial.begin(115200);
    WiFi.begin(ssid, pass);
    conectaWifi();
    client.setServer(server,mqttPort);
    client.setCallback(callback);
    conectaSensor();
}
```

A função de *loop()* é dividida em 3 partes, uma que se preocupa em manter a conexão do *ESP32* na rede e no sensor, uma responsável por receber e armazenar os valores do sensor e por último uma parte responsável por tratar essa informação e enviar ela para o jogo.

```
//--Loop--//
void loop() {
```

```

    client.loop(); //Mantem a conexão MQTT
    delay(10);

//-----//

    if(!client.connected()){
        Serial.println("Perda de conexão com servidor");
        conectaMQTT();
    }
    if(!distSensor.begin()){
        Serial.println("Perda de conexão com sensor");
        conectaSensor();
    }

//-----//

    if (millis() - lastMillis > 5000){
        lastMillis = millis();
        client.publish(channelIdPub, "ping");
    }

```

Dentro da parte de conexão, é colocado a função da biblioteca *PubSubClient.h* responsável por manter a conexão continua com o *broker MQTT* seguido de uma função *delay()* que, de acordo com o exemplo da biblioteca, consegue diminuir a instabilidade da conexão. Em seguida, são colocado dois *if()* que estão sempre verificando se o *MQTT* e o sensor *VL53L0X* ainda estão conectados, no caso onde não estejam então é chamada a função de conexão de cada parte individualmente para que o elemento possa ser reconfigurado. Por último, é programado um algoritmo que envia um *ping* para o servidor evitando que o *ESP32* seja desconectado por inatividade, ele utiliza um macro *lastMillis* que é um *unsigned long* de referência para saber quanto tempo se passou desde a última vez que foi enviado uma mensagem, isso é feito comparando o valor de *lastMillis* com a função *millis()* que retorna por quanto tempo o programa está sendo executado.

```

//-----//

    if(distSensor.isRangeComplete() && contagem < 3) {
        valor += distSensor.readRange();
        contagem += 1;
    }

```

Em seguida é realizada a medição do sensor, isso é feito através de uma função da biblioteca do sensor capaz de retornar *true* ou *false* se o valor no sensor estiver pronto para ser lido ou não, além disso também é introduzido a variável *contagem*, um *int* cujo valor inicial é sempre 0, seu papel é contar quantas vezes o valor será lido antes de ser enviado, nesse caso 3 vezes, isso é feito pois não é bom usar o valor de forma direta só pela primeira medida, usar uma média aritmética com os últimos 3 valores lidos gera mais confiança no número usado para gerar o sinal que será enviado ao jogo.

Por isso, em seguida é usado uma variável *valor*, também um *int* inicializado com 0, que irá somar os valores lidos do sensor 3 vezes, por último será incrementado o valor de *contagem* em 1.

```
//-----//  
if(contagem == 3){  
    valor = valor/contagem;  
    contagem = 0;  
    if(valor > tresholdMax && isClose){  
        isClose = false;  
    }  
    if(valor < tresholdMin && !isClose){  
        isClose = true;  
        client.publish(channelIdPub, "1");  
        lastMillis = millis();  
    }  
  
    valor = 0;  
}
```

A última parte do loop se responsabiliza de enviar a informação para o jogo, para isso primeiro é verificado se a variável *contagem* chegou ao valor de 3, então será feita a média aritmética dos valores recuperados na variável *valor*, isso é feito dividindo *valor* por *contagem* que nesse momento é igual a quantidade de vezes que *valor* foi incrementado com o valor do sensor. Após isso, *contagem* tem seu valor zerado para que seja possível repetir a lógica do algoritmo.

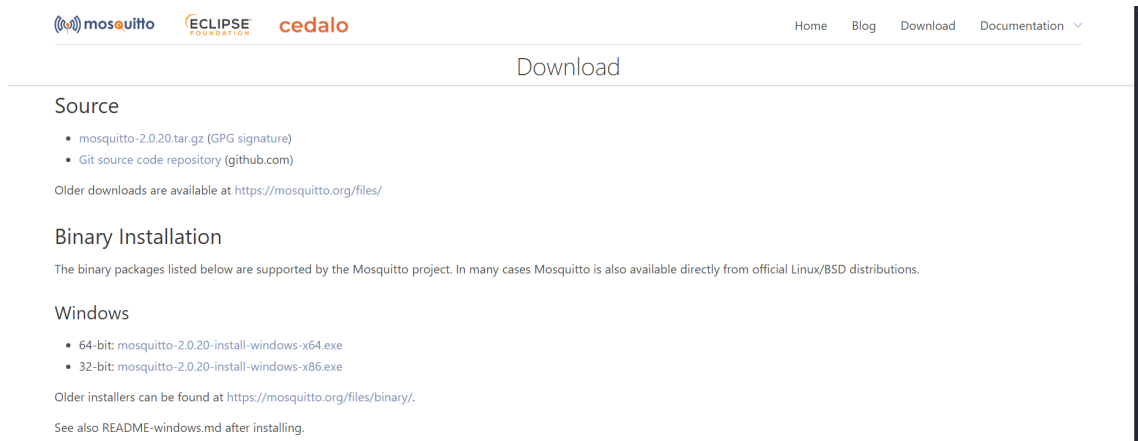
Posteriormente, é feito uma comparação para saber se a variável *valor* é maior do que *tresholdMax*, um macro que define um valor representativo da distância física na máquina de remo onde será considerado que o banco esteja longe do sensor, além disso é usado uma variável booleana *isClose* responsável por rastrear quando o banco está longe ou perto, por isso quando *valor* passa de *tresholdMax* e *isClose* é igual a *true*, o valor de *isClose* é então passado para *false*.

Da mesma forma, também terá um *thresholdMin* que representa a distancia física onde o banco do remo fica perto do sensor, se *valor* ficar menor do que *thresholdMin* e *isClose* é igual a *false*, então *isClose* é passado para *true* e é enviado uma mensagem com o valor de 1 para o jogo, adicionalmente *lastMillis* é resetado. Após esse processo, a variável *valor* é então zerada para que o processo possa ser repetido sem erro.

2.2 Servidor

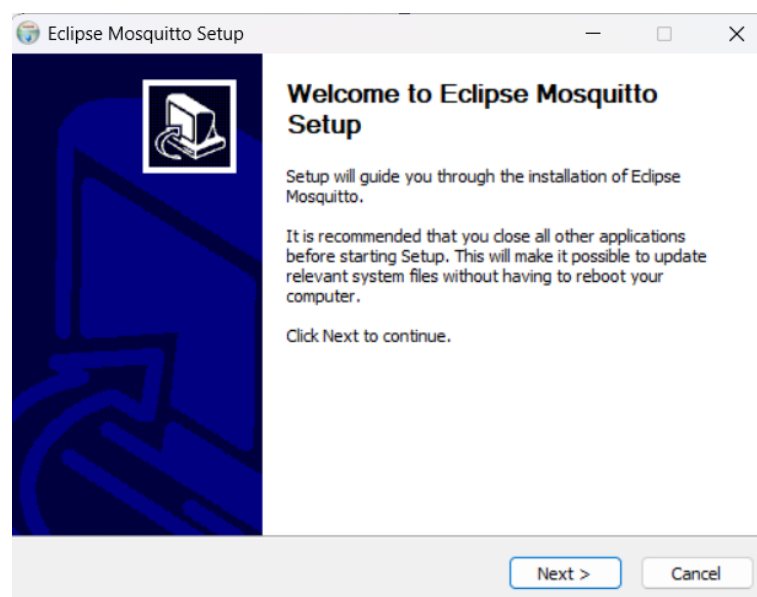
Para o funcionamento do projeto foi necessário utilizar um Broker *MQTT*, capaz de receber mensagens e enviar mensagens entre diferentes dispositivos usando protocolo *MQTT*, para isso foi decidido pela instalação da versão mais recente do *EclipseMosquitto* até o presente momento quando esse relatório está sendo escrito (versão 2.0.20).

Figura 2 – Página de Download do *EclipseMosquitto*



Uma vez que todo o projeto está sendo feito em *Windows*, foi baixado o instalador para *Windows* 64-bit, mostrado na imagem 2. Ao executar o instalador foram seguidos os seguintes passos. A primeira tela mostrada na figura 3, ele recomenda que todos os outros aplicativos sejam fechados antes de começar a instalação, nesse passo é pressionado o botão "Next >".

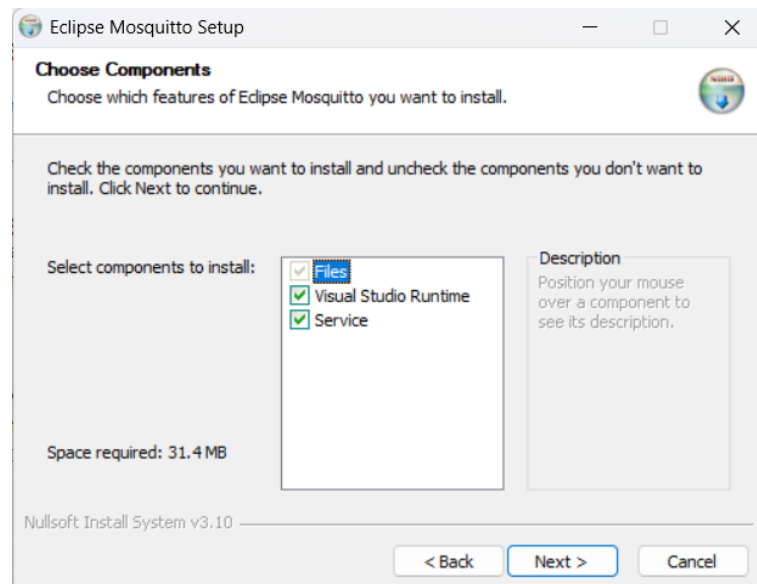
Figura 3 – Instalação *Mosquitto* passo 1



A janela então mudará para a janela apresentada na figura 4, que pergunta quais outros componentes do *Mosquitto* o usuário deseja instalar, não há necessidade de mexer

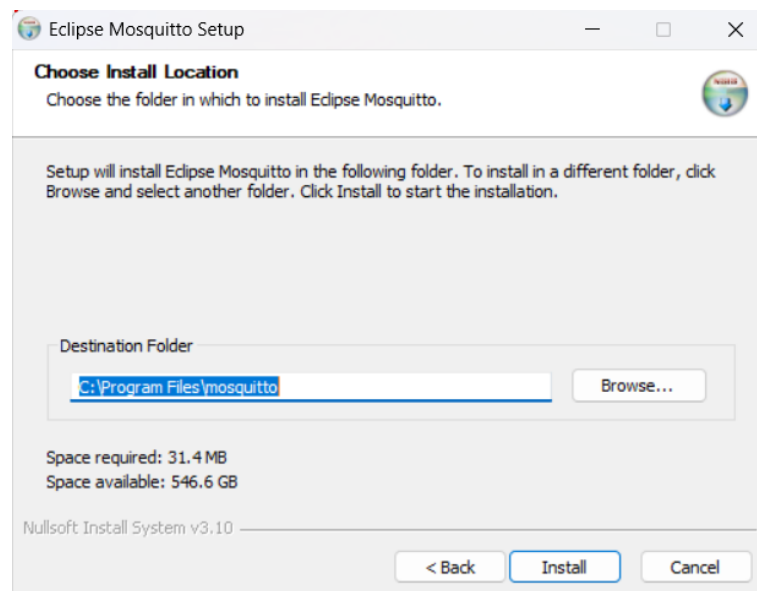
nessa janela então só será pressionado novamente o botão "Next >".

Figura 4 – Instalação *Mosquitto* passo 2



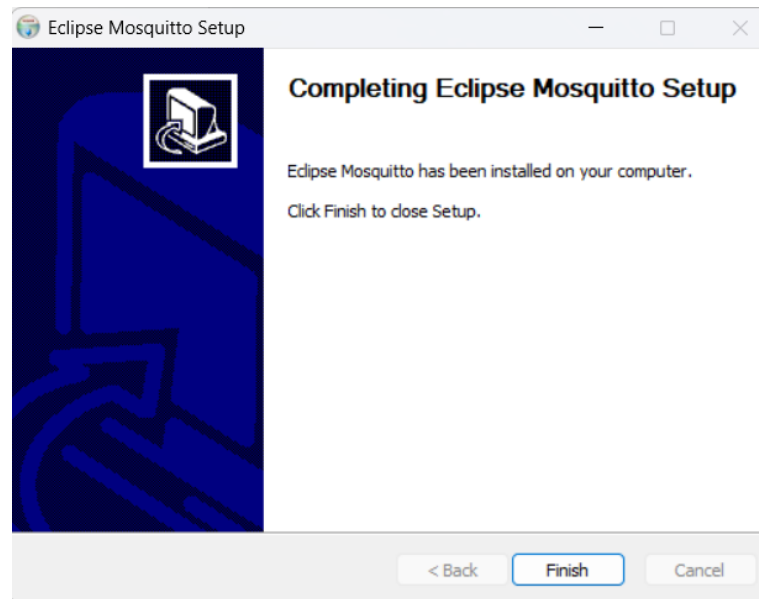
Em seguida, será apresentado a tela que pergunta ao usuário em qual diretório do computador onde está sendo realizada a instalação que o *Mosquitto* será instalado, essa tela é mostrada na figura 5. Para o projeto atual ele não será alterado, porém o local de instalação não deve fazer diferença. Será pressionado o botão "Install".

Figura 5 – Instalação *Mosquitto* passo 3



O programa irá então mostrar uma tela de carregamento e então irá perguntar se deve concluir a instalação, figura 6. Para concluir a instalação basta clicar no botão "Finish".

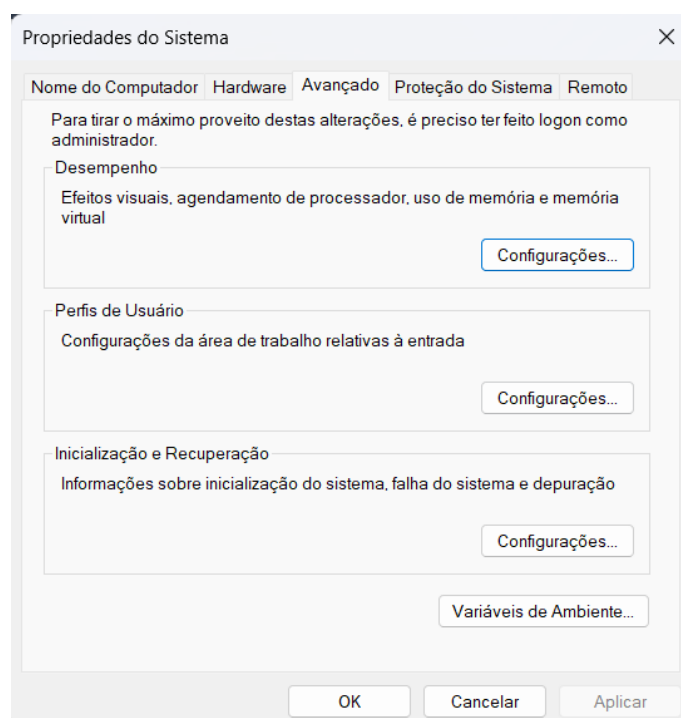
Agora com o aplicativo está instalado, para facilitar a utilização do broker, será necessário mexer em algumas configurações de sistema do *Windows*, em principal para

Figura 6 – Instalação *Mosquitto* passo 4

poder usar a palavra *mosquitto* como um comando no terminal a partir de qualquer diretório.

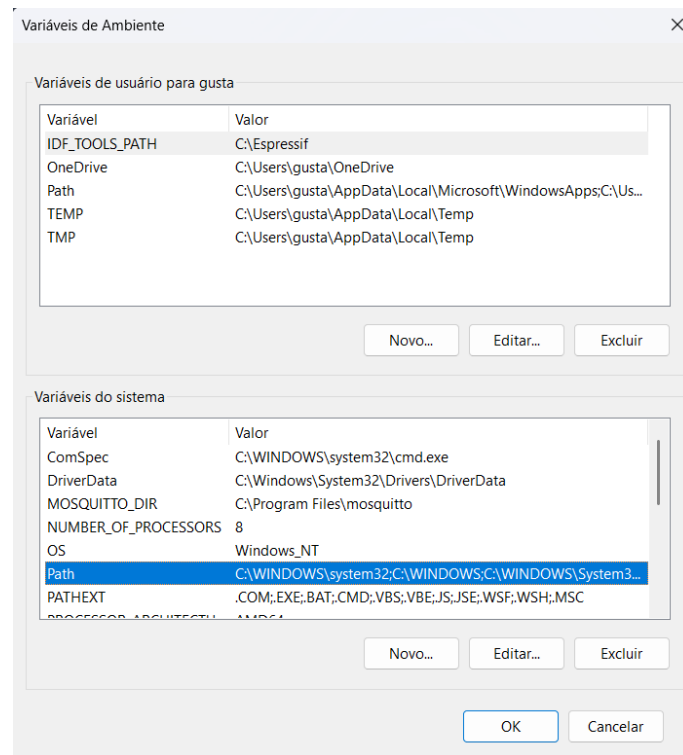
O passo a passo inicial para fazer isso é o seguinte: Abrir o explorador de arquivos, procurar e em seguida clicar com o botão direito em "*Este Computador*" e selecionar a opção "*Propriedades*", dentro de propriedades se encontra um atalho chamado "*Configurações Avançadas do Sistema*", ao clicar nela a janela mostrada na figura 7 com o nome "*Propriedades do Sistema*" irá aparecer.

Figura 7 – Propriedades do Sistema



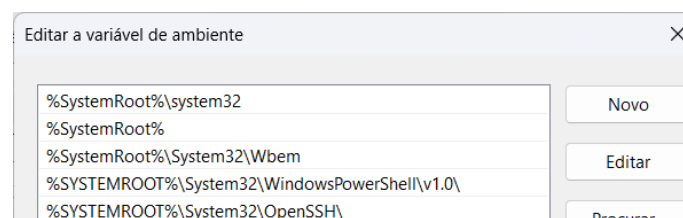
Nessa janela, em seguida é selecionado o botão "*Variáveis de Ambiente...*", isso irá abrir uma janela com o nome "*Variáveis de Ambiente*" mostado na figura 8, na parte inferior será selecionado a variável "*Path*", com ela selecionada então será pressionado o botão "*Editar...*".

Figura 8 – Variáveis de Ambiente



Será aberta uma janela com o nome "*Editar a variável de Ambiente*", figura ,onde estarão dispostas as variáveis do sistema que podem ser selecionadas e alteradas. Como o objetivo é criar uma variável de sistema capaz de iniciar a execução do *Mosquitto*, então será pressionado o botão *Novo*.

Figura 9 – Editar a variável de ambiente



A janela então irá pedir para digitar o endereço de uma pasta, o endereço que deve ser inserido deve ser igual ao endereço do diretório onde o *Mosquitto* foi instalado, nesse caso será igual ao diretório mostrado na figura 5. Para verificar de que a operação funcionou, é necessário abrir o terminal do *Windows* e digitar o comando *mosquitto -h*, a resposta deve ser semelhante a mostrada na figura 10.

Figura 10 – Comando *mosquitto -h*

```

Microsoft Windows [versão 10.0.26100.3037]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\gusta>mosquitto -h
mosquitto version 2.0.20

mosquitto is an MQTT v5.0/v3.1.1/v3.1 broker.

Usage: mosquitto [-c config_file] [-d] [-h] [-p port]

-c : specify the broker config file.
-d : put the broker into the background after starting.
-h : display this help.
-p : start the broker listening on the specified port.
    Not recommended in conjunction with the -c option.
-v : verbose mode - enable all logging types. This overrides
    any logging options given in the config file.

See https://mosquitto.org/ for more information.

C:\Users\gusta>

```

O próximo passo é realizar a configuração do brooker, para isso é preciso abrir O explorador de arquivos do *Windows* a pasta onde o *Mosquitto* está instalado e abrir um arquivo chamado *mosquitto.conf*, esse arquivo é um arquivo de configuração que o *Mosquitto* usa para funcionar, dentro desse arquivo por padrão existem várias instruções de como realizar essa configurações e as várias opções disponíveis.

Figura 11 – Arquivo *mosquitto.conf*

Nome	Data de modificação	Tipo	Tamanho
devel	28/11/2024 10:14	Pasta de arquivos	
acfile.example	16/10/2024 16:26	Arquivo EXAMPLE	1 KB
ChangeLog	16/10/2024 16:26	Documento de Te...	141 KB
cjson.dll	03/10/2024 18:56	Extensão de aplica...	35 KB
edl-v10	16/10/2024 16:26	Arquivo	2 KB
epl-v20	16/10/2024 16:26	Arquivo	15 KB
libcrypto-1_1-x64.dll	24/08/2021 06:57	Extensão de aplica...	3.338 KB
libcrypto-3-x64.dll	03/10/2024 19:00	Extensão de aplica...	4.596 KB
libssl-1_1-x64.dll	24/08/2021 06:57	Extensão de aplica...	668 KB
libssl-3-x64.dll	03/10/2024 19:00	Extensão de aplica...	803 KB
mosquitto.conf	16/10/2024 16:26	Arquivo CONF	41 KB
mosquitto.dll	16/10/2024 16:28	Extensão de aplica...	91 KB
mosquitto	16/10/2024 16:28	Aplicativo	255 KB
mosquitto.ico	16/10/2024 16:26	Arquivo ICO	34 KB
mosquitto_ctrl	16/10/2024 16:28	Aplicativo	53 KB
mosquitto_dynamic_security.dll	16/10/2024 16:28	Extensão de aplica...	99 KB
mosquitto_passwd	16/10/2024 16:28	Aplicativo	24 KB
mosquitto_pub	16/10/2024 16:28	Aplicativo	51 KB
mosquitto_rr	16/10/2024 16:28	Aplicativo	56 KB
mosquitto_sub	16/10/2024 16:28	Aplicativo	57 KB

Dessa forma, as configurações colocadas serão *"allow_anonymous true"*, que permite que usuários sem *username* possam realizar uma conexão com o *broker*. Também, é preciso adicionar *"listener 1883"* e *"listener 8083"*, o primeiro é para o uso da porta

1883 que é a padrão para o uso de *MQTT* e a segunda foi colocada para testes usando *Websocket* porém a implementação não foi finalizada. O último comando então adicionado ao arquivo foi `"socket_domain ipv4"` que também foi colocado para testes com *Websocket*. Os comandos ficarão escritos como mostrados na figura 12.

Figura 12 – Comandos no arquivo *mosquitto.conf*

```
907 allow_anonymous true
908 listener 1883
909 listener 8083
910 socket_domain ipv4
```

Com tudo configurado, então é possível finalmente iniciar o *Mosquitto*, para fazer isso é necessário digitar no terminal `"mosquitto -v -c <o caminho para mosquitto.conf>"`, na figura é mostrado um exemplo de como o programa é iniciado, no exemplo o terminal já está no diretório que contém *mosquitto.conf*.

Figura 13 – Iniciando *Mosquitto*

```
C:\Program Files\mosquitto>mosquitto -v -c mosquitto.conf
1738963899: mosquitto version 2.0.20 starting
1738963899: Config loaded from mosquitto.conf.
1738963899: Opening ipv6 listen socket on port 1883.
1738963899: Opening ipv4 listen socket on port 1883.
1738963899: Opening ipv4 listen socket on port 8083.
1738963899: mosquitto version 2.0.20 running
```

Agora com essa janela aberta é possível acompanhar quais dispositivos estão conectados ao *broker* e quais mensagens estão sendo enviadas entre os dispositivos, pois essas informações serão impressas no terminal conforme esses eventos ocorrem.

Para acessar o valor de *IP* da máquina na rede que será colocado tanto no código do *ESP32* quanto no jogo na *Unity* é só abrir o terminal e digitar `"ipconfig"` e pegar o valor referente ao `"Endereço IPv4"`.




2.3 Unity

A parte da *Unity* do projeto revolve ao redor do desenvolvimento do jogo que irá receber as mensagens do sensor pelo *broker MQTT*, nessa parte serão detalhadas todos os objetos que estão presentes no jogo no momento em que esse documento está sendo escrito, suas propriedades, configurações e *scripts*.

Para realizar o projeto, foram necessários três bibliotecas de *assets*, eles podem ser encontrados na lista de "*MyAssets*" da conta da *Unity* do *TEAM*, seus nomes são "*Warped Vehicles*", "*Warped Space Shooter*" e "*MQTT for Unity*", eles podem ser visualizados na figura 14.

Enquanto "*Warped Vehicles*" e "*Warped Space Shooter*" são bibliotecas que vão ser usadas apenas para deixar o jogo bonito, "*MQTT for Unity*" é uma biblioteca que facilitou bastante a integração da *Unity* com o *MQTT*.

Figura 14 – Bibliotecas de Assets Utilizadas no Projeto

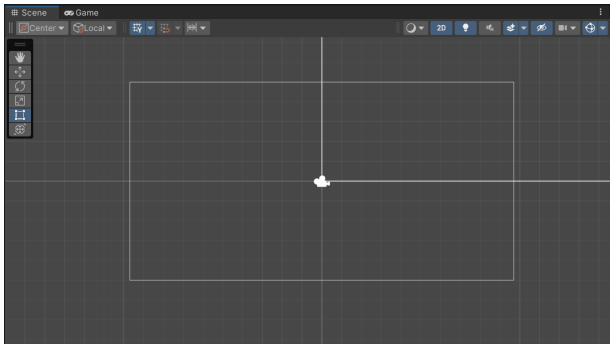
	<div>ANSIMUZ</div> <div>Warped Vehicles</div> <div>16.4 MB</div> <div>Purchase date: Feb 7, 2025</div> <div>Organization: team_lab(Personal)</div>	<div>Last updated: May 11, 2021 • Version: 1.0</div> <div>First release</div> <div>Add label Hide asset</div>	Open in Unity
	<div>ANSIMUZ</div> <div>Warped Space Shooter</div> <div>10.5 MB</div> <div>Purchase date: Feb 7, 2025</div> <div>Organization: team_lab(Personal)</div>	<div>Last updated: Oct 27, 2020 • Version: 1.0</div> <div>First release</div> <div>Add label Hide asset</div>	Open in Unity
	<div>ROCWORKS</div> <div>MQTT for Unity</div> <div>759.8 KB</div> <div>Purchase date: Sep 24, 2024</div> <div>Organization: team_lab(Personal)</div>	<div>Last updated: Jan 31, 2024 • Version: 1.1.3</div> <div>1.1.3 Added CleanSession option and unsubscribe function</div> <div>more</div> <div>Add label Hide asset</div>	Open in Unity

Agora sobre o jogo em si, o primeiro objeto dentro da interface do jogo é uma "*Main Camera*", esse objeto é padrão da *Unity* que é criado junto com a cena, para esse caso ela só foi ajustada para um ângulo que deixasse o jogo com uma visão em 2D, na figura 15b, mostra as suas componentes visíveis no inspetor e suas coordenadas na cena.

Em seguida, o jogo possui um *GameObject*, chamado "*bg – back*" para representar o plano de fundo do jogo, esse objeto fica centralizado no meio da cena alinhado com o espaço da câmera e usa o sprite de mesmo nome dentro da biblioteca de *assets* de *Warped Space Shooter*.

Figura 15 – *Main Camera* no jogo

(a) *Main Camera*



(b) *Main Camera* Componentes

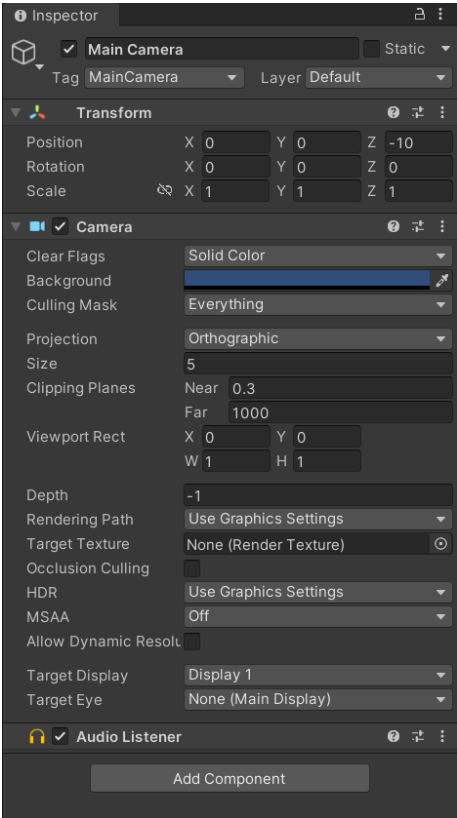
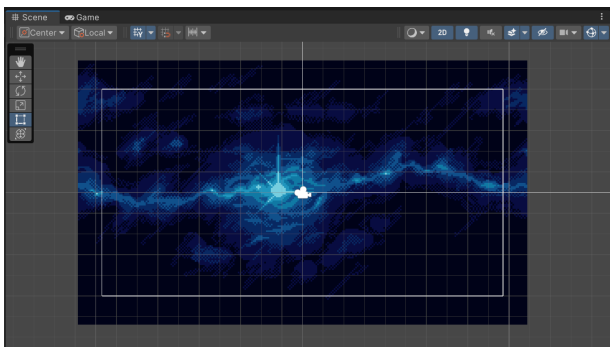
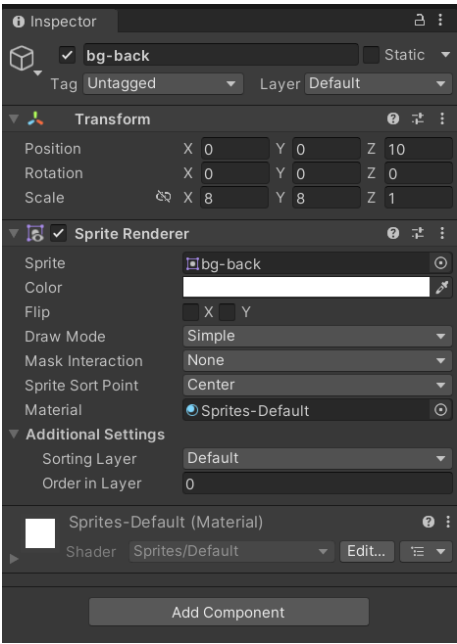


Figura 16 – “bg – back” no jogo

(a) “bg – back”



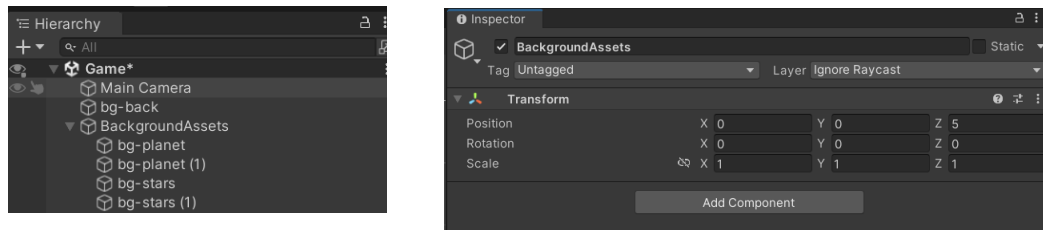
(b) “bg – back” Componentes



Para o jogo poder mostrar ao jogador que a nave está se movendo serão colocados objetos do cenário se movendo na tela da direita para a esquerda, dessa forma criando a ilusão de que a nave em si está se movendo da esquerda para a direita. Para fazer isso, primeiro é criado um *GameObject* vazio chamado *BackgroundAssets*, ele é usado para guardar os elementos de cenário que irão se mover da direita para a esquerda, porém esse objeto em si não possui nenhum componente além de sua localização no cenário, como é mostrado na figura 17b.

Figura 17 – *BackgroundAssets* no jogo

- (a) *BackgroundAssets* na hierarquia (b) *BackgroundAssets* no inspetor



Já os elementos dentro que ele guarda são mais complexos, o intuito foi fazer com que esse objetos pudessem passar pela tela um por vez em uma ordem aleatória, por causa disso foi necessário planejar um sistema onde o *GameObject* é arrastado para a esquerda a uma velocidade adaptativa e quando for detectado que o *GameObject* sair da tela, outro *GameObject* escolhido da lista de forma aleatória para ser levado até além da ponta direita da tela e então arrastado para a esquerda com a mesma velocidade adaptativa.

Dessa forma, além do *sprite* visual, que vem da biblioteca de *Warped Space Shooter* onde eles tem o mesmo nome que o *GameObject*, foi colocado um *BoxCollider* e um *Rigidbody*, que vai ser usado para detectar quando o *GameObject* sair da tela. Na cena do *Unity*, os *GameObjects* da lista estão inicialmente deslocados -44 em Y e cada está deslocado 35.3 em X um do outro, isso pode ser visualizado nas figuras 18a e 18b.

Foi preciso criar então um objeto que pudesse controlar a velocidade do jogo e manipular o *GameObjects* dentro de *BackgroundAssets*, ele foi chamado de *BackgroundController* e é o *GameObject* mais complexo da aplicação, pois é necessário que ele se comunique com várias partes diferentes do jogo.

BackgroundController possui dois *BoxColliders* que para realizar a detecção de quando um dos elementos de *BackgroundAssets* e ele também possui um *script*, que vai receber o script do *GameObject* *MQTTClint* e do *SpeedBar*, que serão explicados mais adiante, além disso ele recebe os *GameObjects* de *BackgroundAssets* em um array.

O *Script* feito foi nomeado de *MovingBackground* e pode ser dividido em 6 partes a serem explicadas, cada uma delas será apresentada a seguir.

Figura 18 – Elementos de *BackgroundAssets* no jogo

- (a) Elementos de *BackgroundAssets* na hierarquia (b) Elementos de *BackgroundAssets* no inspetor

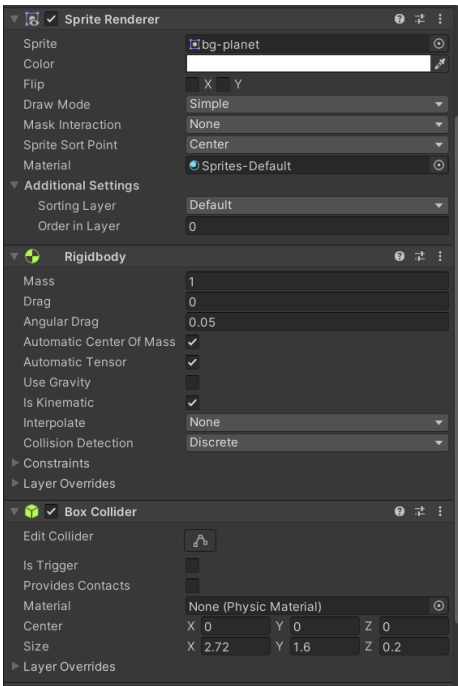
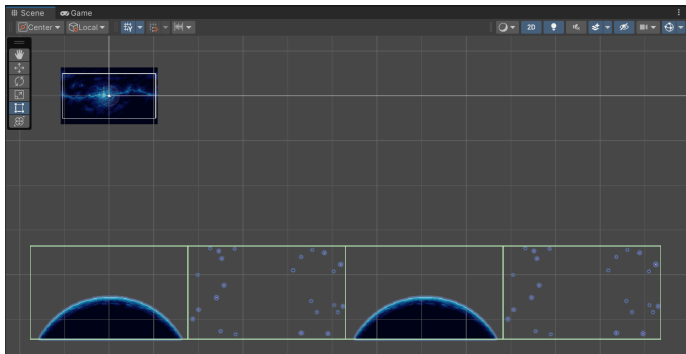
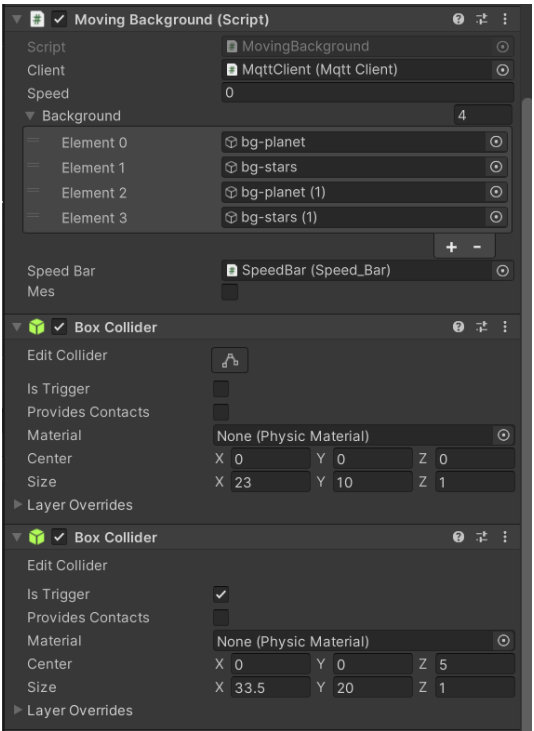
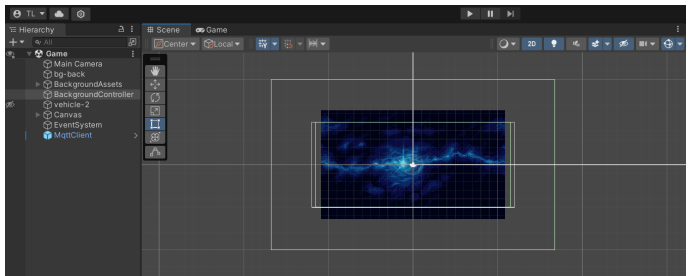


Figura 19 – *BackgroundController* no jogo

- (a) *BackgroundController* (b) *BackgroundController* no inspetor



```
using System;
using System.Collections;
using System.Collections.Generic;
using Rocworks.Mqtt;
using Unity.VisualScripting;
using UnityEngine;

public class MovingBackground : MonoBehaviour
{
    //---MQTT---
    public MqttClient Client;

    //Variáveis
    public float speed = 0;
    private float MaxSpeed = 150f;
    public GameObject[] background = new GameObject[4];
    private int[] cena;
    public Speed_Bar speedBar;
```

Nessa parte, é apresentado as bibliotecas da *Unity* que estão sendo usadas para mexer nesse *script*, essas incluem bibliotecas padrões da *Unity* para mexer com diversos elementos do programa, as bibliotecas padrões do sistema e a biblioteca de *MQTT* que será usada, em seguida o início da classe que segue o padrão de formatação da *Unity*.

Para as variáveis, primeiro é criado um objeto *MqttClient* que na interface da *Unity* irá receber o objeto *MQTTClient* da cena, essa variável é responsável por fazer com que esse código consiga interagir com as mensagens *MQTT* recebidas do *ESP32*. A seguir, é mostrado duas variáveis do tipo *float*, com os nomes de *speed* e *MaxSpeed*. *speed* representa a velocidade atual na qual os objetos na tela estão se movendo enquanto *MaxSpeed* delimita uma velocidade máxima a qual *speed* não pode ultrapassar.

Em seguida são declarados dois vetores, *background* é um vetor de *GameObjects* e é ele que irá receber os elementos de *BackgroundAssets*, aqui já foi delimitado que o seu tamanho é 4, porém isso pode ser alterado no futuro. Já o vetor *cena* é um vetor *int* que armazena quais desses elementos está presente na cena em cada momento, ele faz isso guardando o valor da posição dos elementos dentro de *background* que estão sendo usados naquele momento, quase sempre haverá 2 desses elementos sendo mostrados ao jogador.

Por último, *speedBar* é um objeto do tipo *Speed_Bar* que se refere a um *GameObjeto* que representa uma barra lateral no *HUD* do jogo que representa a velocidade do jogador, esse *GameObject* será explicado de forma mais profunda posteriormente.

```
void Start()
{
    speedBar.SetMaxSpeed(MaxSpeed);
    cena = new int[2] {0,1};
    for(int i = 0;i<2;i++){
        background[cena[i]].transform.position
            = new Vector3(3.6f+35.3f*i,0,5);
    }
}
```

Na função *Start*, a primeira coisa feita é definir qual será o valor máximo de velocidade na barra de velocidade, em seguida é inicializado o vetor *cena* e são definidos os *GameObjects* que aparecerão primeiro no jogo, esses sendo os que estão nas posições 0 e 1. Em seguida, dentro do *for*, serão movidos os elementos definidos na variável *cena* para suas posições iniciais imediatamente do lado direito da câmera porém fora de vista, uma do lado da outra em *X*.

```
void FixedUpdate()
{
    if(speed >= MaxSpeed){
        speed = MaxSpeed;
    }

    for(int i = 0;i<2;i++){
        background[cena[i]].transform.Translate
            (Vector3.left * Time.deltaTime * speed);
    }
    speedBar.SetSpeed(speed);
    speed -= 0.2f*speed/100;
    if(speed <= 0){
        speed = 0;
    }
}
```

A função *FixedUpdate*, que funciona como a função *loop* do presente no código do *ESP32*, irá atualizar a tela a em intervalos fixos de tempo que são independentes da taxa de quadros do jogo. Para esse caso, ele realiza 5 principais ações, a primeira é sempre verificar se a velocidade do jogo ultrapassou a velocidade máxima e se isso acontecer, a

velocidade será então igual ao valor máximo, em seguida é executado um *for* que faz o trabalho de mover os elementos ativos em *cena* dentro de *background* baseado no valor da velocidade atual.

Depois, ele executa uma função que vem do objeto *speedBar* para atualizar a barra para exibir o valor de velocidade mais recente e então faz um calculo para reduzir o valor da velocidade a cada atualização de forma que ela diminua na mesma proporção do quão alto é o seu valor atual para que dessa forma ele possa ser representativo da velocidade do movimento do jogador. Adicionalmente o código também verifica se o calculo de *speed* é um valor negativo, em casos onde isso aconteça o valor é resetado de volta para 0.

```
void OnTriggerExit(Collider col){
    col.gameObject.transform.position = new Vector3(0,-44f, 5);
    cena[0] = cena[1];
    switch(cena[1]){
        case 1:
            cena[1] = UnityEngine.Random.Range(2,4);
            break;
        case 2:
            cena[1] = 1 + 2*UnityEngine.Random.Range(0,2);
            break;
        case 3:
            cena[1] = UnityEngine.Random.Range(1,3);
            break;
        default:
            break;
    }
    background[cena[1]].transform.position
    = new Vector3(background[cena[0]].transform.position.x+35.3F,0,5);
}
```

OnTriggerExit é uma função da *Unity* que é executada quando o *collider* vinculado ao objeto com o *script* detecta que um objeto não está mais em contato com o *collider*, ele então retorna como objeto o respectivo *collider* saiu dessa área. Dessa forma, *OnTriggerExit* inicialmente irá resgatar o objeto que saiu da área do *collider* através do *collider* daquele objeto e irá deslocar ele para uma coordenada fora da visão da câmera onde ele irá aguardar para ser chamado novamente.

Em seguida, o primeiro valor de *cena* será substituído pelo segundo, representando que, uma vez que o objeto sai da visão da câmera o objeto que já está sendo visto se torna o novo primeiro objeto, então é necessário escolher um novo segundo objeto, para isso foi

implementado o *Switch*, ele irá verificar qual *elemento* de *background* está sendo usado e irá escolher outro aleatoriamente para substituí-lo, isso foi feito dessa forma para evitar a sensação de que o cenário está se repetindo em loop. Por último, o objeto é então movido para sua nova posição localizado 35.3 unidades de distância da *Unity* a esquerda do objeto que já está passando na tela.

```
void OnMouseDown(){
    if((speed + 10+7*(MaxSpeed-speed)/MaxSpeed)>MaxSpeed){
        speed = MaxSpeed;
    }else{
        speed += 10+7*(MaxSpeed-speed)/MaxSpeed;
    }
}

public void OnMessageArrived(MqttMessage m)
{
    String mStr = m.GetString();
    if(mStr == "1"){
        speed += 20+7*(MaxSpeed-speed)/MaxSpeed;
    }
}
```

As próximas funções são as principais formas de incrementar a velocidade do jogo, porém *OnMouseDown* foi criado principalmente por questões de *debug* enquanto a implementação *MQTT* ainda não estava completamente pronta. Sua funcionalidade é bastante simples, é realizado um calculo para que quanto mais próximo da velocidade máxima estiver a velocidade do jogo, representado por *speed*, mais difícil é para que ela continue subindo, dessa forma é necessário que o jogador seja mais rápido para se manter em uma velocidade de jogo elevada.

OnMouseDown é uma função executada quando o jogo detecta que o botão do mouse foi pressionado, para este caso o jogo primeiro verifica se o valor da nova velocidade é maior do que a velocidade máxima e caso seja então a velocidade recebe o valor da velocidade máxima para que não seja possível ultrapassar esse valor, caso não então o incremento acontece de forma normal.

Já em *OnMessageArrived*, que é uma função da biblioteca de *MQTT*, e ela é chamada quando é recebido uma nova mensagem no canal configurado dentro do *GameObject MqttClient* que será explicado depois, nesse método, primeiro é transformado

o valor da mensagem em *String* e comparado com o valor enviado pelo *ESP32* que representa o momento em que a cadeira física chega na posição desejada. É possível observar que o balanceamento de velocidade é um pouco diferente entre as funções, isso se deve ao movimento da cadeira ser muito mais lento do que o pressionar de um mouse.

```
public float getSpeed(){
    return speed;
}

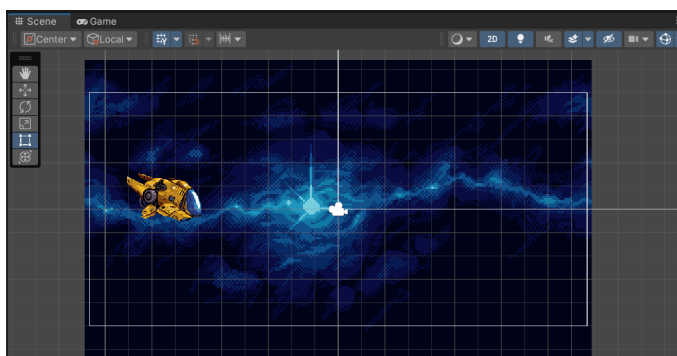
public float getMaxSpeed(){
    return MaxSpeed;
}
```

Por último, são implementado dois *getters* que serão usados exclusivamente pelo código que controla o movimento da nave do jogo, eles servem apenas para retornar os valores de *speed* e *MaxSpeed*.

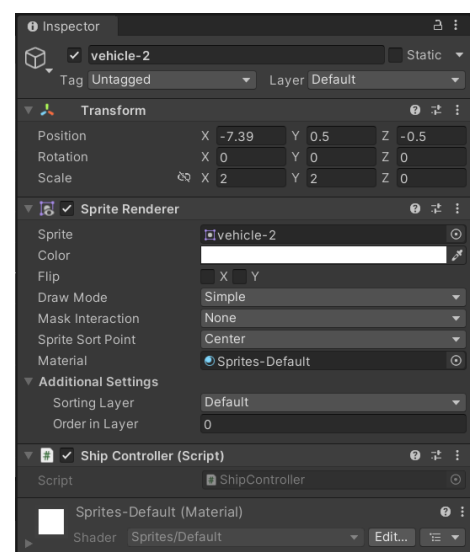
Retornando para a interface da *Unity*, o próximo *GameObject* em *Hierarchy* é o *vehicle - 2*, esse objeto representa uma nave espacial que representa o jogador do jogo, essa nave utiliza do *sprite* de nave de mesmo nome na biblioteca de *assets* "*Warped Vehicles*", ele foi posicionado próximo de sua posição inicial quando o jogo inicia e nele foi adicionado um *script* chamado *ShipController* que controla a posição vertical da nave baseado em sua velocidade. Isso pode ser visualizado na figura.

Figura 20 – *vehicle - 2* no jogo

(a) *vehicle - 2*



(b) *vehicle - 2* no inspetor



O *script* de *vehicle* – 2 é composto por suas variáveis globais e o seu comportamento é descrito nas funções *Start* e *FixedUpdate*. Entre suas variáveis ele possui: uma variável do tipo *GameObject* nomeada de *background* que irá representar o *backgroundController* para que seja possível resgatar os valores de *speed* usando *getSpeed*, em seguida uma variável *float* nomeada *speed*, como o nome diz ela irá guardar o valor de velocidade toda vez que for solicitado, isso também se aplica para *float maxSpeed* que recebe o valor máximo de velocidade.

A seguir, *float hight* é uma variável que, como o nome já diz, irá armazenar o valor atual da altura da nave na tela, da mesma forma *float maxHight* recebe o valor máximo que a nave pode subir, para que ela não ultrapasse a tela acidentalmente. Por último, *float offset* é um *offset* de altura como o nome indica já que para que a nave comece no canto inferior da tela ela precisa estar um pouco deslocada para baixo, esse deslocamento é representado pelo *offset*.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ShipController : MonoBehaviour
{
    private GameObject background;
    private float speed;
    private float maxSpeed;
    private float hight;
    private float maxHight = 7f;
    private float offset = 3.5f;
```

Dentro da função *Start* são definidas então 3 coisas, primeiro é atribuído a *background* o *GameObject backgroundController* para que ele seja capaz de usar suas funções e em seguida é feito exatamente isso para resgatar o valor atribuído para *maxSpeed* usando *getMaxSpeed*, por último é definido que a *speed* da nave quando o jogo começa é igual a 0.

```
void Start(){
    background = GameObject.Find("BackgroundController");
    maxSpeed
    = background.GetComponent<MovingBackground>().getMaxSpeed();
    speed = 0;
}
```

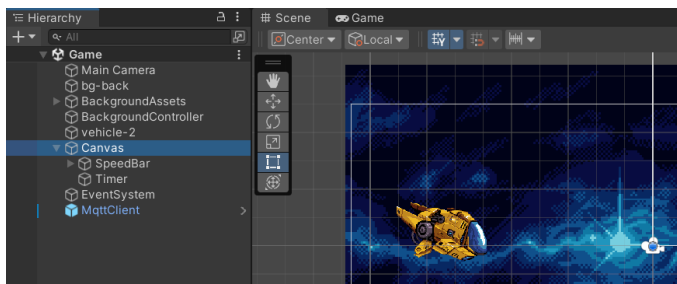
Dentro de *FixedUpdate* o valor de *speed* será sempre atualizado para o valor atual de velocidade da variável pertencente a *MovingBackground*, usando isso é então feito um calculo para definir a posição vertical, armazenado em *hight*, isso é feito calculando o percentual de velocidade comparado com a velocidade máxima possível $speed/maxSpeed$ e multiplicando esse valor pela altura máxima, em seguida é subtraído o *offset* para que a nave fique na posição correta. Por último, o valor de é aplicado no vetor posição da nave para que ela passe a impressão que está se movendo.

```
void FixedUpdate()
{
    speed = background.GetComponent<MovingBackground>().getSpeed();
    hight = -offset+(speed*maxHight/maxSpeed);
    this.transform.position = new Vector3(-7.39f,hight,-0.5f);
}
```

Em seguida, na hierarquia da *Unity* está o *GameObject Canva*, este *GameObject* se especializa em adicionar elementos visuais no *HUD* do jogo. Para o caso deste jogo, serão adicionados dois elementos de *HUD*, uma barra lateral que indica a velocidade do jogador chamada de *SpeedBar* e um temporizador chamado de *Timer*.

Figura 21 – *Canva* no jogo

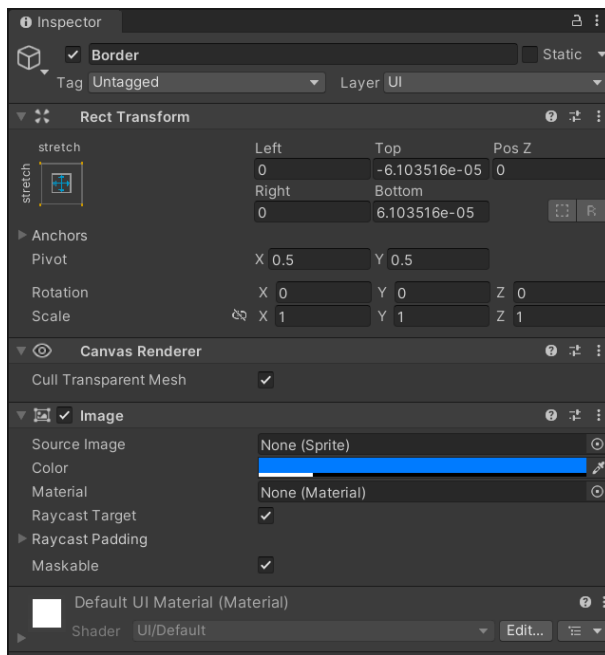
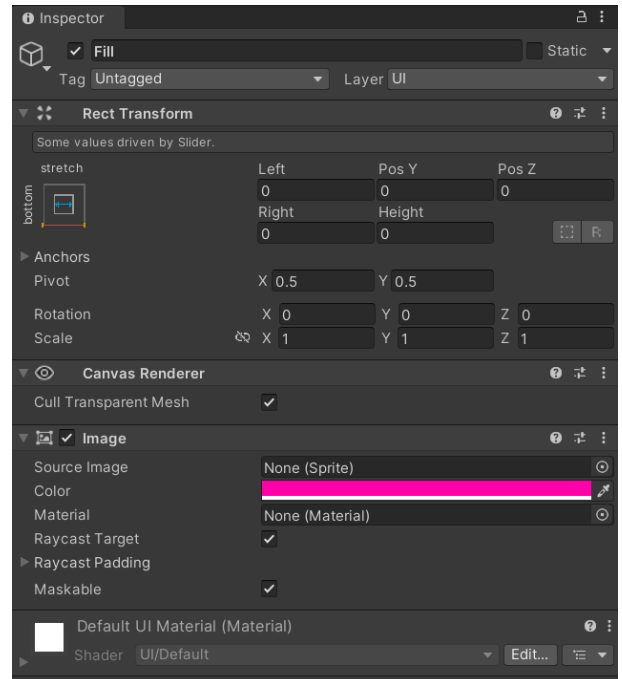
(a) *Canva* na hierarquia



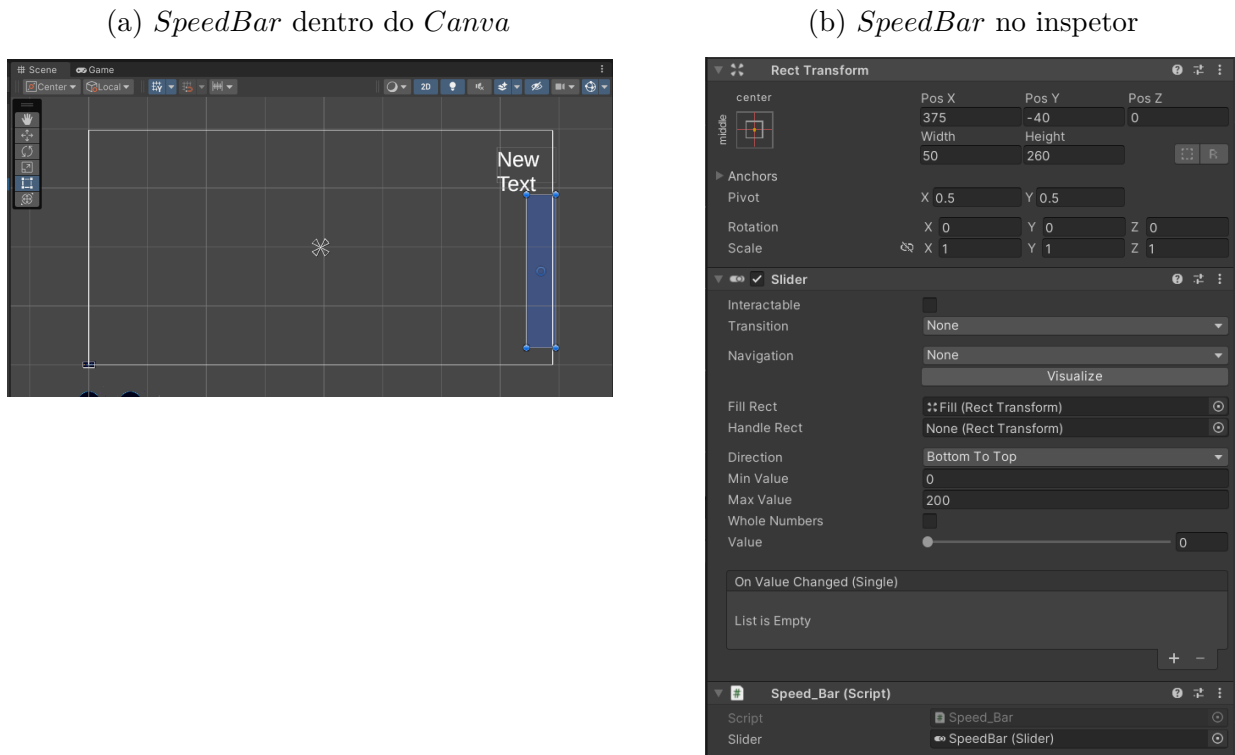
(b) *Canva* no inspetor



Sobre o *GameObject SpeedBar*, ele é composto de outros dois *GameObjects* que representam a moldura da barra e o seu preenchimento, chamados respectivamente de *Border* e *Fill* ambos esses objetos são classificados como imagem dentro da lista de assets de *UI*, porém eles não carregam nenhum *sprite* e são compostos por apenas cores sólidas. A única diferença entre os elementos, são suas cores, suas coordenadas e seus *Anchor Presets*, que determinam como o objeto irá de comportar no *Canva*.

Figura 22 – *Border* e *Fill*(a) *Border* no inspetor(b) *Fill* no inspetor

Já o *SpeedBar* em si é considerado um elemento chamado *slider*, já que *Border* irá representar o fundo da barra, para que *Fill* possa dar a impressão de preenchimento, é necessário atribuir ela como *Fill Rect* dentro dos atributos de *slider*, isso pode ser verificado na imagem 23b.

Figura 23 – *SpeedBar* no jogo

O *slider* também possui um *script* chamado *Speed_Bar* responsável por atualizar a barra baseado nos valores de *speed* de *MovingBackground*. Ele é um código simples, precisando apenas de um objeto que representa ele mesmo e das bibliotecas de *UI*, isso ocorre pois o momento em que o valor é alterado ocorre dentro do próprio *MovingBackground*, por causa disso é necessário apenas que *Speed_Bar* tenha os *setters* que vão definir qual é o valor máximo e o valor atual.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Speed_Bar : MonoBehaviour
{
    public Slider slider;

    public void SetMaxSpeed(float speed)
    {
        slider.maxValue = speed;
    }

    public void SetSpeed(float speed)
```

```

{
    slider.value = speed;
}

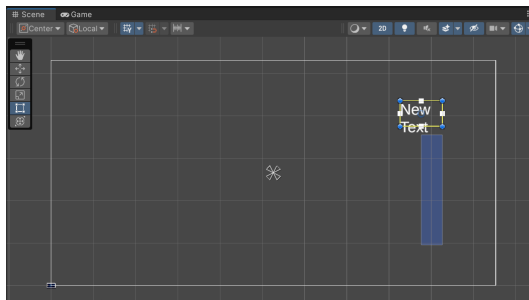
}

```

O segundo objeto dentro do *Canva* é o *Timer*, que é um objeto de texto para *UI*, ele possui várias configurações que envolvem mexer na fonte, tamanho da letra, cor da letra, entre outras coisas, ele também possui componentes que permitem configura-lo em relação ao *Canva*.

Figura 24 – *Timer* no jogo

(a) *Timer*: dentro do *Canva*



(b) Configurações de *Canva* do *Timer*

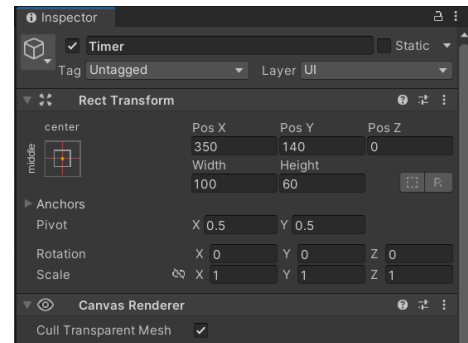
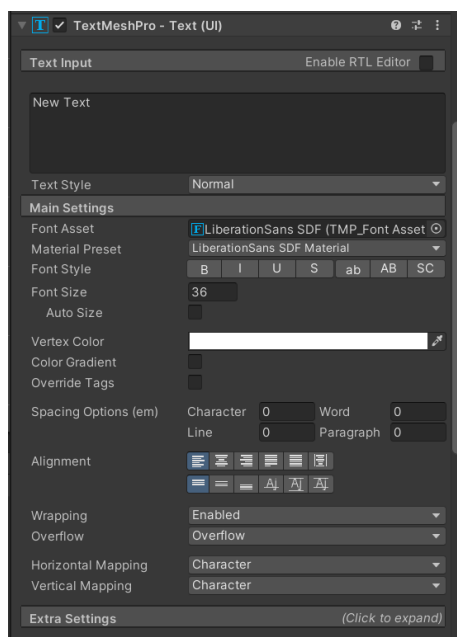
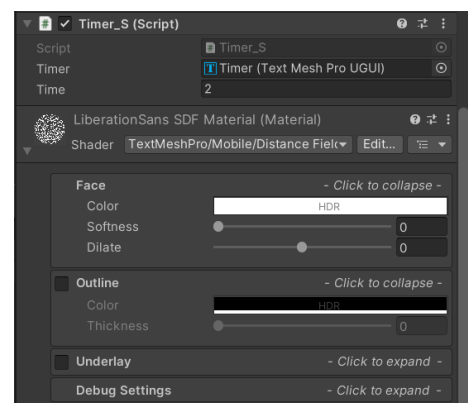


Figura 25 – Configurações do *Timer*

(a) Configuração de Texto



(b) *Script* e outras configurações



O *script* para o *Timer*, chamado de *Timer_S*, irá realizar uma contagem regressiva de 30 minutos desde o momento em que o jogo iniciou e então irá pausar o jogo na *Unity*, isso foi feito para simular o momento em que o jogo irá parar de ser executado e mostrar o desempenho do usuário no exercício, porém essa segunda parte não foi implementada a tempo então ele só simula um temporizador.

Timer_S usa as bibliotecas padrões da *Unity* para *UI* com adição de *TMPPro* que ajuda a mexer com texto, as principais variáveis são *timer* do tipo *TMP_Text* que representa o próprio *Timer*, *int Time* que representa o valor máximo de duração em minutos e por último *int countM* e *float countS* que irão representar o tempo em minutos e em segundos do temporizador respectivamente, *countS* é representado em *float* pois ele interage com valores que podem ter casas decimais.

```
using System.Collections;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using TMPPro;
using UnityEngine;
using UnityEngine.UI;

public class Timer_S : MonoBehaviour
{
    public TMP_Text timer;
    public int time = 30;

    private int countM = 0;
    private float countS = 0;
```

Em *Start* a primeira coisa que o código faz é atribuir de *time* para *countM* e em seguida ele irá alterar o texto da caixa de texto para mostrar os minutos e os segundos no formato 00 : 00. Já em *FixedUpdate* é realizado a subtração de *countS* por *Time.deltaTime* que representa quanto tempo passou desde que a aplicação foi iniciada, caso ele identifique que *countS* chegou abaixo de 0 então o programa irá incrementar 59 para *countS* e diminuir *countM* em 1, isso irá simular a operação de *timer*.

Se o programa identificar que *countM* ficou abaixo de 0 então *Debug.break* é chamado que irá colocar a aplicação dentro da *Unity* em pausa, no caso de quando o programa é exportado o relógio apenas continua contando dentro das casas negativas. Por fim, é atualizado o valor de texto do *GameObject* limitando *countS* a só mostrar valores inteiros.

```
void Start()
{
    countM = time;
    timer.text = countM.ToString()+":"+countS.ToString();
}
void FixedUpdate()
{
    countS-=Time.deltaTime;
    if(countS < 0){
        countS += 59f;
        countM -= 1;
    }
    if(countM < 0){
        Debug.Break();
    }
    timer.text = countM.ToString()+":"+countS.ToString("0");
}
```

Os últimos *GameObjects* na *Hierarchy* funcionam juntos, *MqttClient* é um *Prefab*, isso significa que ele já vem pronto da biblioteca de *MQTT for Unity*, ele é responsável por se conectar ao *broker MQTT*, além disso *MqttClient* tem uma dependência de funcionamento com *EventSystem*, mas não é necessário mexer na configuração de *EventSystem* em específico.

Como mostrado na figura 26a, *MqttClient* irá receber os mesmos valores descritos na seção sobre o *ESP32*, mas os atributos *Username* e *Password* podem ser diferentes, na imagem também é incluído um *Client Id*, mas ele não é obrigatório. Adicionalmente, é necessário adicionar na seção *OnMessageArrived* qual o código que irá receber a mensagem do *broker*, o canal onde deve ser escrito na parte de *Subscribe Topics*.

Dessa forma, o jogo irá conseguir receber as mensagens do *ESP32* e conseguir funcionar da forma desejada.

Figura 26 – *MqttClient* no Inspetor

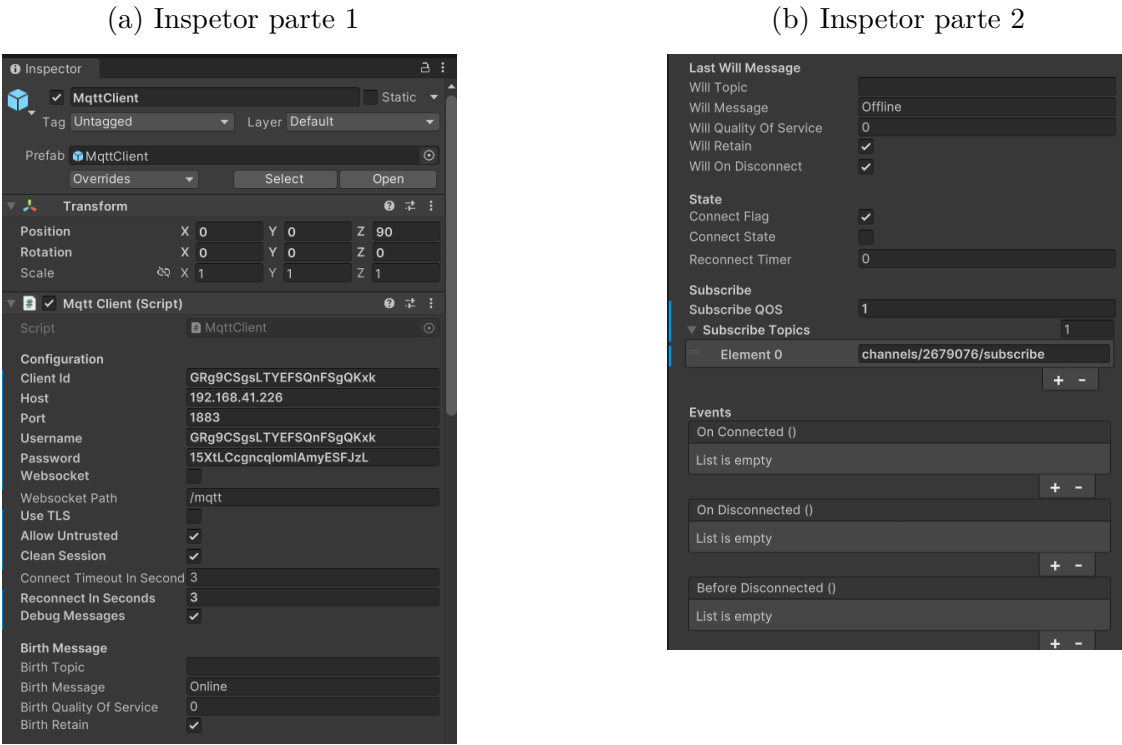
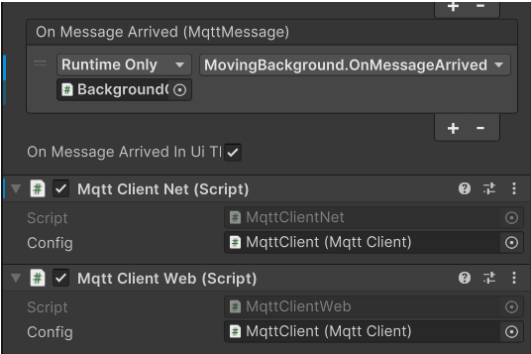


Figura 27 – Inspetor Parte 3



Bibliografia

- [1] Gustavo Henrique Lima de Araujo. “GDD do jogo Runners”. Em: *UFRN* (2024). URL: <https://docs.google.com/document/d/1UjMtCoHBZkXef3hAbdkcWmFnbdSFnic-1VOOgLJwlNU/edit?usp=sharing>.