

Documento de criação e uso da Simulação Robô - Godot

1.0. Geral:

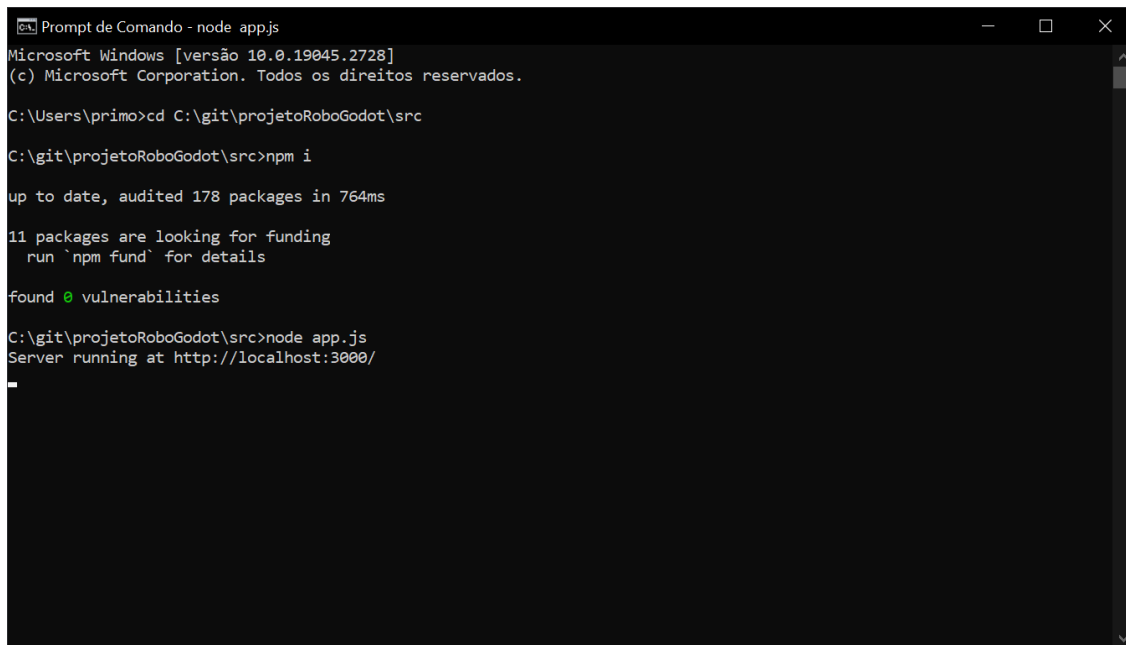
O projeto se trata de uma simulação básica do controle remoto de um braço robótico. O foco principal é o “index.html”, que é a página web para realizar o controle do robô. Por outro lado, existe também a simulação criada em Godot, que responde aos comandos do Front End.

1.1. Setup:

Ao iniciar o servidor Node, e acessar o “localhost:3000”, a página de simulação já abre automaticamente em uma nova guia. Caso não aconteça, é possível executar o arquivo “executavel.exe”, na pasta “./src/Godot/executavel”.

Em ordem para abrir o servidor node:

- 1 - Pressione Win + R, digite “cmd” e de enter.
- 2 - No cmd, digite “cd” e caminhe até o diretório “src” do projeto.
- 3 - Digite “npm i” espere o download das dependências.
- 4 - Por fim, digite “node app.js”, o servidor deve iniciar e basta acessar a url fornecida.



```
Prompt de Comando - node app.js
Microsoft Windows [versão 10.0.19045.2728]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\primo>cd C:\git\projetoRoboGodot\src

C:\git\projetoRoboGodot\src>npm i

up to date, audited 178 packages in 764ms

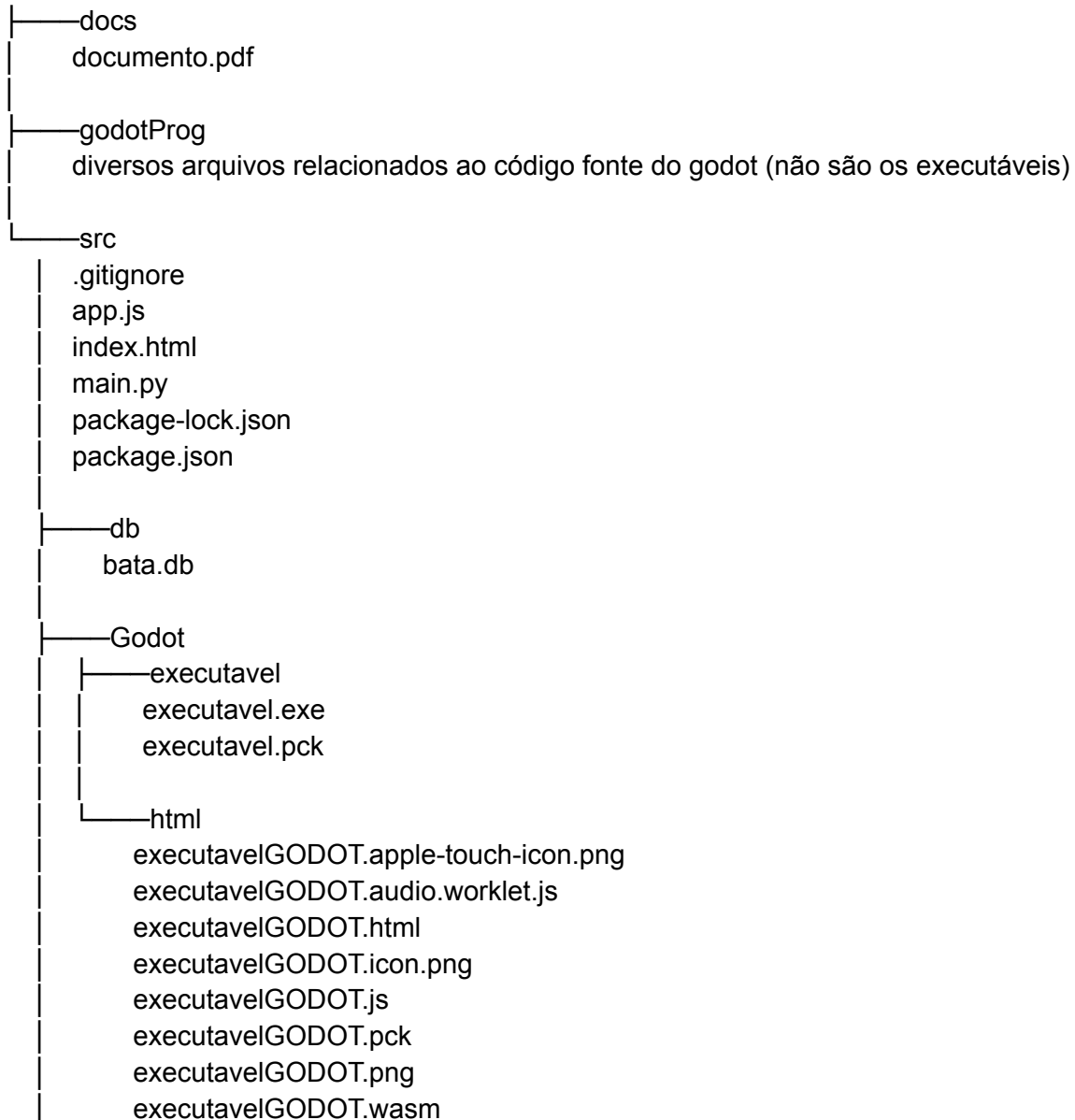
11 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\git\projetoRoboGodot\src>node app.js
Server running at http://localhost:3000/
```

1.2. Estrutura de pastas

C:.



2.0. Front End:

O front end foi criado utilizando basicamente HTML, CSS, JavaScript, e os frameworks Bootstrap e JQuery.

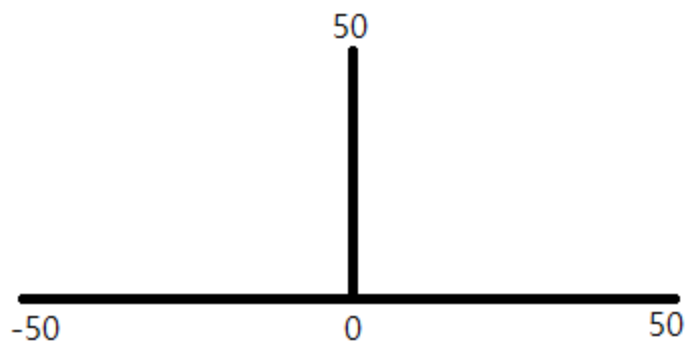
Os quatro botões superiores tratam do movimento “bidimensional” do braço robótico, ou seja, no eixo X e Y. Os dois botões inferiores tratam do eixo Z, para cima e para baixo.

Logo abaixo, as “Posições atuais” apresentam através de coordenadas, a posição atual da ponta do braço robótico, em X, Y e Z, respectivamente. Vale ressaltar que há um limite na movimentação, sendo no eixo X de -45 a 45, no eixo Y de 0 a 60, e no eixo Z de 0 a 10.

Interface:



Movimentação:



Código do front end em HTML, apresentando os botões:

```
<body>
  <div class="contaier">
    <div class="d-flex justify-content-center title">
      <h1 class="text-center">Controle de Braço Robótico</h1>
    </div>

    <div class="d-flex justify-content-center btnclass">
      <button id="left"></button>
      <button id="front">^</button>
      <button id="back">v</button>
      <button id="right">></button>
    </div>

    <div class="d-flex justify-content-center btnclass">
      <button id="up">^^</button>
      <button id="down">vv</button>
    </div>
  </div>
```

Função que envia as posições para o servidor:

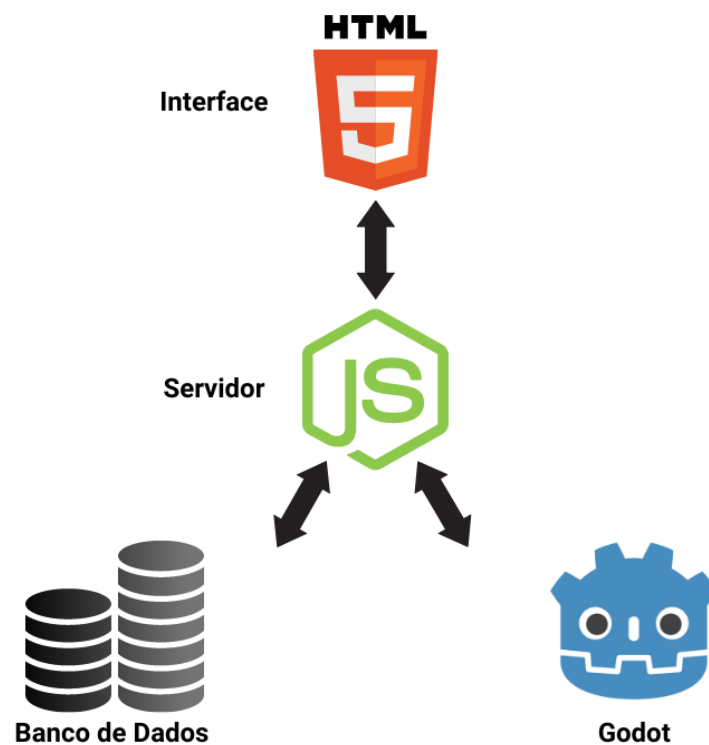
```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    console.log(this.responseText);
  }
};
xhttp.open("POST", "http://localhost:3000/senddata", true);
xhttp.setRequestHeader("Content-type", "application/json");
var data = JSON.stringify({
  x: positions[0],
  y: positions[1],
  z: positions[2]
});
xhttp.send(data);
// bootstrap toast notification
$('.toast').toast('show');
```

3.0. Back End:

O back end foi construído utilizando node.js, ambiente de execução de javascript para servidores. Existem as seguintes rotas:

/senddata - Envia as posições de X, Y e Z para o banco de dados.
/getdata - Recebe todas as posições do banco de dados (não é utilizada)
/getlast - Recebe as últimas posições do banco de dados
/delete - Apaga todas as posições do banco de dados, menos a primeira

Este servidor estabelece a comunicação entre a página web, o banco de dados, e a simulação em Godot, com no esquema abaixo:



Basicamente, o servidor recebe as requisições HTTP tipo POST da página web, e envia os dados para o banco. Ao mesmo tempo, a simulação requisita esses dados pelo tipo GET, portanto, o servidor faz o intermédio entre o banco de dados e o godot.

A comunicação é feita utilizando o ip localhost (127.0.0.1), e a porta 3000.

Rota que envia os dados ao servidor:

```
// Rota que envia os dados x, y e z para o banco
app.post('/senddata', urlencodedParser, function(req, res) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  let db = new sqlite3.Database(dbpath, sqlite3.OPEN_READWRITE, (err) => {
    if (err) {
      console.error(err.message);
    }
    console.log('Connected to the data database.');
```

```
  });

  let sql = `INSERT INTO movement (x, y, z) VALUES (${req.body.x}, ${req.body.y}, ${req.body.z})`;

  db.run(sql, function(err) {
    if (err) {
      return console.log(err.message);
    }
    console.log(`A row has been inserted with rowid ${this.lastID}`);
  });

  db.close((err) => {
    if (err) {
      console.error(err.message);
    }
    console.log('Close the database connection.');
```

```
  });

  res.send('Data received');
});
```

Rota que recebe o último dado do banco:

```
// Rota que retorna o último dado do banco
app.get('/getlast', (req, res) => {
  res.statusCode = 200;
  res.setHeader('Access-Control-Allow-Origin', '*'); // Isso é importante para evitar o erro de CORS

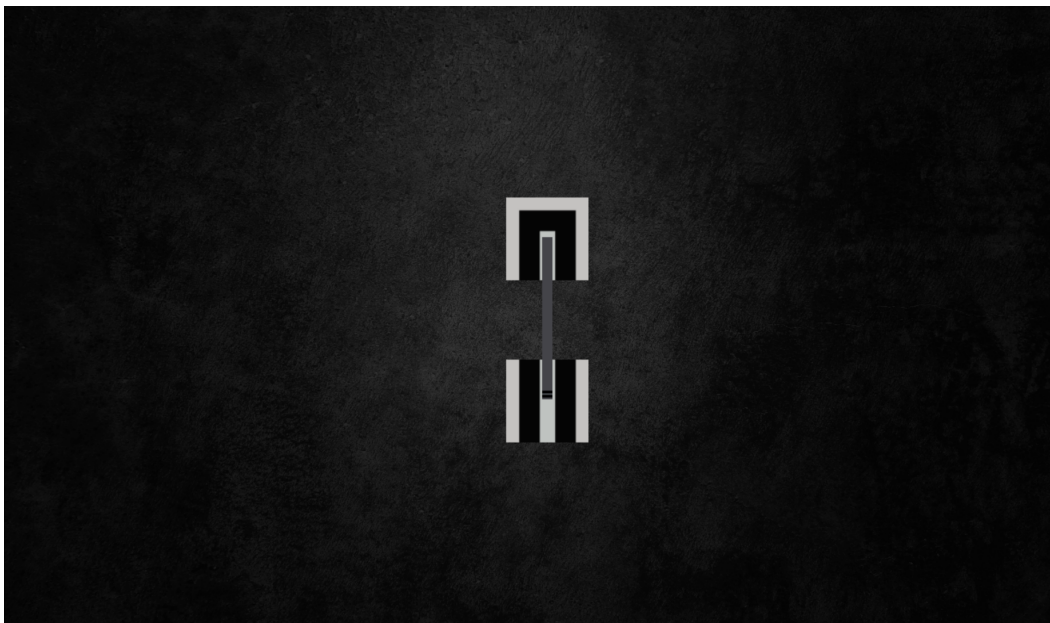
  var db = new sqlite3.Database(dbpath); // Abre o banco
  var sql = 'SELECT x, y, z FROM movement ORDER BY id DESC LIMIT 1;';
  db.all(sql, [], (err, rows) => {
    if (err) {
      throw err;
    }
  })
  res.send(rows)
});
db.close(); // Fecha o banco
});
```

4.0. Simulação:

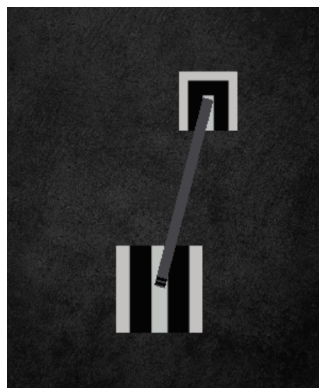
Como já mencionado, a simulação foi feita utilizando o Godot Engine, ambiente e engine para desenvolvimento de jogos. A simulação é básica, se trata de uma visão vertical (de cima para baixo) de um braço robótico.

O script fica o tempo todo buscando por uma nova linha no banco de dados, e quando encontra, atualiza a posição do braço, em tempo real. A parte superior retrata a ponta, a parte inferior a base, e a linha conectando-os representa o braço.

Imagem da simulação:



Representação de um movimento na diagonal e para baixo:



Código que acessa a rota para receber os últimos dados do banco:

```
1  extends HTTPRequest
2
3  # Array para salvar as posições
4  var arrayPositions = [0,0,5]
5
6  func _process(delta):
7      ># Faz sucessivas requisições HTTP
8      >self.connect("request_completed", self, "_on_request_completed")
9      >var err = self.request("http://127.0.0.1:3000/getlast")
10     ># if err != OK:
11     >> push_error("Um erro ocorreu!")
12
13     # Função que recebe os dados da requisição e os inputa no array
14     func _on_request_completed(result, response_code, headers, body):
15         >var positions = parse_json(body.get_string_from_utf8())
16         >arrayPositions[0] = positions[0]["x"]
17         >arrayPositions[1] = positions[0]["y"]
18         >arrayPositions[2] = positions[0]["z"]
19
```

Com isso, é possível construir um código para a movimentação do sprite:

```
1  extends Sprite
2
3  func _process(delta):
4      ># Centraliza e atualiza as posições com nas informações do HTTP request
5      >position.x = (get_viewport_rect().size.x + texture.get_size().x/2) / 2 + getRequests()[0] * 3
6      >position.y = (get_viewport_rect().size.y / 2) - getRequests()[1] * 3
7      ># if getRequests()[2] < 6:
8      >> scale.x = getRequests()[2]/5 + 0.25
9      >> scale.y = getRequests()[2]/5 + 0.25
10     ># else:
11     >> # Tratativa para o sprite não ficar gigante
12     >> scale.x = getRequests()[2]/6 + 0.3
13     >> scale.y = getRequests()[2]/6 + 0.3
14
15     # Função que importa a requisição do outro script
16     func getRequests():
17         >var impHTTP = get_parent().get_node("Sprite/HTTPRequest").arrayPositions
18         >return impHTTP
19
```


5.0. Considerações finais:

Há um arquivo com o nome de “main.py”, ele seria um servidor Flask, mas não está sendo utilizado, e foi mantido apenas por motivos de referência.

Este projeto faz parte da grade curricular do Inteli, módulo 5 de Engenharia de Computação. Os demais códigos e arquivos podem ser acessados livremente nos arquivos do projeto, disponíveis em <https://github.com/IsraelNLC/projetoRoboGodot>.