



**ISEL**  
INSTITUTO SUPERIOR DE  
ENGENHARIA DE LISBOA

# Ship Conquest

48361, Francisco Barreiras, A48361@alunos.isel.pt  
48362, Tomás Carvalho, A48362@alunos.isel.pt

Orientador: Engenheiro Paulo Pereira, paulo.pereira@isel.pt

Relatório do projeto realizado no âmbito de Projecto e Seminário  
Licenciatura em Engenharia Informática e de Computadores

Junho de 2023



# INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

## Ship Conquest

48361 Francisco Cortes Castel'Branco Barreiras

---

48362 Tomás Rua de Carvalho

---

Orientador Engenheiro Paulo Pereira

---

Relatório do projeto realizado no âmbito de Projecto e Seminário  
Licenciatura em Engenharia Informática e de Computadores

Junho de 2023



---

# Resumo

A indústria dos jogos é uma das mais relevantes e em maior crescimento no campo do entretenimento nos dias de hoje. Sendo um dos objetivos deste trabalho publicar o resultado final obtido para demonstrar as nossas capacidades, o projeto foi centrado em torno de um jogo composto por uma componente de aplicação cliente e uma componente de servidor. O projeto consiste num jogo de exploração multijogador de estratégia em tempo real (RTS), onde cada jogador controla uma frota de navios e com estes pode explorar o mundo, lutando contra adversários e conquistando novas ilhas. O sistema do projeto integra uma aplicação cliente designada para dispositivos móveis *Android* e *iOS*, e uma aplicação back-end hospedado pelo serviço Google.

Neste projeto foram enfrentados diversos desafios, tanto a nível de desenvolvimento do produto, de gestão de tempo como também a nível de possíveis custos pela aplicação servidor quando estiver hospedada.

Foram trabalhados diversos conceitos técnicos, nomeadamente *Procedural Generation* e Algoritmos pseudo aleatórios de ruído como o *Perlin Noise* e o *Simplex Noise* para a geração dos mundos, funções de interpolação como as *Bézier Curves*, para a representação dos caminhos percorridos e algoritmos de pesquisa como o  $A^*$  para encontrar caminhos entre pontos.

Com a conclusão deste trabalho é possível demonstrar as competências técnicas, conhecimento e pensamento crítico dos autores deste trabalho, desenho e planeamento, assim como justificar e documentar as soluções escolhidas em comparação com as demais para os problemas encontrados. Devido à pequena janela temporal de desenvolvimento de projeto, alguns aspetos fora do foco atual do projeto mas ainda relevantes não tiveram a atenção desejada inicialmente como o equilíbrio do jogo e algumas funcionalidades opcionais.



---

# Abstract

The game industry is one of the most relevant and presents one of the biggest growth rates among the entertainment business these days.

Considering one of the objectives of this project was publishing and deploying the final result to evidence the creators' capabilities as Software developers, this project is centered around a game built upon a client app and a server component. \*- The project consists of a multiplayer exploration RTS game (Real Time Strategy Game), where each player controls a fleet to explore the world, fight enemies and conquer islands.

The project's system integrates a client application designed for mobile devices, running *Android* or *iOS* and a back-end app hosted by Google Cloud Platform.

It was opted to choose the Flutter framework to build the client application due to its cross-platform functionalities, it also provides a good development experience, competence for complex graphical interfaces and relevance in the current job market. The Spring framework was chosen for the back-end due to its quick configuration process, development experience and a rich ecosystem of tools to implement and deploy a Web application.

In this project, several tough challenges were faced, both in terms of product development, time management and also in terms of possible hosting and deploying costs for the server application.

Several technical concepts were worked on, namely *Procedural Generation* and pseudo-random noise algorithms such as *Perlin Noise* and *Simplex Noise* for the generation of the game's worlds, interpolation functions such as *Bézier Curves*, to represent the paths taken by the ships and search algorithms such as  $A^*$  to find paths between points.

With the conclusion of this work it is possible to demonstrate the technical skills, the acquired knowledge and critical thinking by the authors of this project, design and planning, as well as ability to justify and document the chosen solutions in comparison with some already existing solutions for the faced problems. Due to the small time window given to develop the project, some aspects outside of the current focus of the project, although still relevant, did not receive the attention initially desired, such as balancing the game and some optional features.





---

# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Motivação . . . . .	1
1.2	Objetivos . . . . .	1
1.3	Ideia . . . . .	2
1.4	Requisitos . . . . .	4
1.4.1	Requisitos essenciais . . . . .	4
1.4.2	Requisitos opcionais . . . . .	4
1.5	Arquitetura . . . . .	5
1.5.1	Tecnologias . . . . .	5
<b>2</b>	<b>Abordagem</b>	<b>7</b>
2.1	Geração do mundo . . . . .	7
2.1.1	Como? . . . . .	8
2.1.2	Desafios . . . . .	8
2.1.3	A Nossa Abordagem . . . . .	8
2.2	Renderização . . . . .	11
2.2.1	A Nossa Visão . . . . .	11
2.2.2	A Nossa Abordagem . . . . .	11
2.2.3	Desafios . . . . .	14
2.3	O Movimento Dos Navios . . . . .	15
2.3.1	Movimento Em Jogos Multijogador . . . . .	15
2.3.2	A Nossa Visão . . . . .	16
2.3.3	A Nossa Solução: Curvas De Bézier . . . . .	16
2.3.4	Obstáculos . . . . .	17
2.4	A Criação dos Caminhos . . . . .	20
2.4.1	Como? . . . . .	20
2.4.2	Requisitos e Abordagem . . . . .	20
2.4.3	Desafios . . . . .	22
2.5	Eventos . . . . .	23

2.5.1	Como? . . . . .	23
2.5.2	Desafios . . . . .	23
2.5.3	A Nossa Abordagem . . . . .	24
2.6	Histórico dos pontos visitados . . . . .	28
2.6.1	Como? . . . . .	28
2.6.2	A Nossa Abordagem . . . . .	28
2.6.3	Desafios . . . . .	30
<b>3</b>	<b>Conclusão</b>	<b>31</b>
3.1	Considerações . . . . .	31
3.2	Dificuldades . . . . .	32
3.3	Trabalho Futuro . . . . .	33

---

# Lista de Figuras

1.1	Aquitetura do sistema . . . . .	5
2.1	Ilustração do algoritmo Perlin Noise . . . . .	8
2.2	Exemplo de um conjunto de ilhas em pontos aleatórios . . . . .	9
2.3	Resultado gerado pelo algoritmo Simplex Noise . . . . .	9
2.4	Exemplo de uma textura de Falloff gerada . . . . .	10
2.5	Resultado da transformação aplicada ao terreno . . . . .	10
2.6	Perspetiva ortogonal vs perspetiva isométrica . . . . .	11
2.7	Exemplo aplicado da perspetiva isométrica . . . . .	12
2.8	Node ColorRamp do Blender . . . . .	13
2.9	O nosso gradiente de cores . . . . .	14
2.10	Ilustração de um navio . . . . .	15
2.11	Tipos de movimento . . . . .	16
2.12	Ilustração de uma Curva de Bézier cúbica . . . . .	16
2.13	Exemplo de um caminho complexo . . . . .	18
2.14	Exemplo do resultado do algoritmo de procura . . . . .	21
2.15	Ilustração da simplificação da Curva de Bézier . . . . .	25
2.16	Exemplo de uma amostra de pontos da interseção . . . . .	25
2.17	Ilustração dos planos de colisão construídos para a Curva de Bézier . . . . .	26
2.18	Exemplo entre a interseção de dois caminhos . . . . .	27
2.19	Mini mapa com os caminhos recortados . . . . .	29
2.20	Mini mapa sem tratamento do comprimento dos caminhos . . . . .	29



# Capítulo 1

---

## Introdução

### 1.1 Motivação

Antes de falar sobre a ideia em concreto do trabalho desenvolvido, é importante explicar a motivação e valores que levaram à construção deste trabalho, como também é relevante para compreender muitas das decisões tomadas ao longo do desenvolvimento deste trabalho.

Foi construída a motivação e valores para este projeto com base em vários fatores relevantes como:

- Considerar uma dimensão adequada para desenvolver um projeto sólido, com a documentação apropriada e um relatório sucinto que descreva competentemente o concretizado.
- Tentar condensar a grande maioria dos conhecimentos adquiridos ao longo do percurso da licenciatura neste projeto, visto que é importante que este trabalho represente as competências adquiridas.
- Aprender novos conceitos e "pisar território ainda inexplorado" como temos feito ao longo do percurso académico. É também importante trazer conteúdo novo a este trabalho para o destacar e para aprofundar o conhecimento em novas áreas.

### 1.2 Objetivos

Em relação aos objetivos e pontos principais a alcançar com este projeto são destacados os seguintes:

- Ter uma componente gráfica que ofereça uma experiência visual fluída que tenha uma estética e atmosfera própria assim como responsiva às ações do utilizador.

- Utilizar um conjunto de tecnologias e frameworks que sejam de interesse continuar a usar, visto que é reconhecida a influência que este projeto pode ter no portefólio e por consequência numa futura carreira profissional.
- Hospedar o trabalho para que possa ser utilizado e desfrutado por um público mais amplo. O deploy bem-sucedido deste trabalho permite não apenas demonstrar as competências técnicas, como também a experiência valiosa adquirida durante o ciclo de vida completo de um projeto.
- Desenvolver um projeto que represente um produto final de entretenimento atrativo e distintivo que se revista de interesse para os consumidores.

### 1.3 Ideia

Considerando que se pretende distribuir a aplicação pelas típicas lojas digitais e utilizar este projeto como uma peça relevante no portefólio foi tido também em conta a relevância do mesmo para o mercado de trabalho.

A indústria dos jogos está em crescimento e é uma das maiores vertentes no que toca ao desenvolvimento de *Software* nos dias de hoje. Ao mesmo tempo optou-se por não se utilizar um game engine de modo a não fechar portas a outras oportunidades fora deste mundo.

De forma a aliar estes objetivos, optou-se por desenvolver a aplicação cliente desta ideia em Flutter[1] devido à sua funcionalidade multiplataforma, boa experiência de desenvolvimento com documentação e ferramentas para debug mais avançadas que a concorrência, competência para interfaces gráficas complexas e relevância no mercado de trabalho. A aplicação servidor foi desenvolvida com a *framework* Spring[2] e a linguagem Kotlin, devido à possibilidade de o programador se abstrair de alguns detalhes de implementação do servidor e do código de infraestrutura.

Ao mesmo tempo, considerou-se ser uma ideia que representa bem desenvolvimento dos criadores como engenheiros ao longo do curso, relacionando-se até com os projetos que se têm vindo a desenvolver no âmbito de outras unidades curriculares, que costumam também ser jogos. Deste modo conseguiu adicionar-se camadas de complexidade ao anteriormente aprendido.

Outro aspeto relevante da programação de jogos é a necessidade de encontrar soluções para diversos desafios técnicos envolvendo também alguma criatividade e que tenham em conta o desempenho da aplicação, o aspeto e a experiência de utilizador.

Tendo em conta estes aspetos procede-se agora à explicação da ideia, um jogo multijogador de estratégia em tempo real[3], no qual cada jogador controla um ou mais barcos e tem como objetivo conquistar ilhas ao redor do mapa.

O objetivo ao longo do jogo é conquistar o maior número de ilhas possível pelo mapa uma vez que as mesmas são a principal fonte de rendimento dos jogadores. As ilhas conquistadas são uma fonte de rendimento passivo, ou seja, quem a conquistou recebe dinheiro de forma proporcional ao tempo passado na ilha até um máximo definido para incentivar a exploração. Estas ilhas podem, posteriormente, ser conquistadas por outros jogadores.

Este dinheiro serve para comprar mais barcos de forma a impulsionar a expansão do domínio do território do mapa. Através deste sistema quis-se criar um ambiente e um fluxo de jogo que mantenha o utilizador interessado na exploração e na dominação do jogo em que se encontra, ao invés de estabelecer um objetivo fixo de vitória que leva inevitavelmente ao desinteresse.

Existe também um sistema de lutas entre navios, quando um navio é enviado para procurar novas ilhas, é possível que este se cruze com navios de outros jogadores pelo caminho, o que provoca uma luta entre os mesmos.

O jogo funciona com um sistema de salas, nas quais o utilizador pode escolher entrar em uma já existente ou criar uma nova. É também possível um utilizador manter progresso em mais do que uma sala em simultâneo.

No servidor, os dados da transformação dos barcos e dos blocos têm uma perspectiva universal, estes dados são compostos pelo tamanho e coordenadas destas entidades.

Para a interface de cliente decidiu-se utilizar perspectiva isométrica alterando a transformação do servidor, desta maneira é possível oferecer ao utilizador uma sensação de 3D sem incorrer na complexidade de renderização e de desenvolvimento de 3D. Desta maneira conseguimos cumprir um dos objetivos estabelecidos, nomeadamente ter uma componente gráfica forte, rápida, responsiva e atrativa.

## 1.4 Requisitos

É importante notar que durante o desenvolvimento destes requisitos, foi seguido a técnica *Slice-Based development*. Nesta abordagem foram divididas as várias tarefas em pequenas "fatias" menores e iterativas, onde cada "fatia" representa uma ou mais funcionalidades que podem ser implementadas de forma independente. Este método tem a vantagem de dar uma melhor perspetiva do estado do trabalho, visto que, não se foca num só componente do desenvolvimento do início ao fim, mas sim em várias fatias dos componentes e de como estes se relacionam.

### 1.4.1 Requisitos essenciais

- Conceção e implementação dos visuais básicos do jogo da aplicação cliente, verificando que esta é capaz de representar os blocos do mapa na perspectiva 2.5D com uma boa performance.
- Criação de uma RESTful API para persistir e manipular os recursos do servidor, como também desenvolver a lógica do aplicação como a geração do mundo, os lobbies, e as várias ações do cliente, além de implementar *Server-Sent Events*[4].
- Desenvolvimento de uma aplicação cliente para dispositivos mobile com a Framework Flutter para representar o estado do jogo e interagir com o servidor, assim como a conceção e realização dos menus e das diversas funcionalidades.
- Planeamento e implementação de um sistema de eventos gerados por ações do jogador; Estes permitem observar os estados passados do jogo como também os estados futuros. Uma das utilidades deste sistema é permitir agendar notificações e ações na aplicação cliente e evitar scheduling no servidor.

### 1.4.2 Requisitos opcionais

- Estender a implementação básica do combate, implementar exércitos para popular os barcos e aumentar o componente de estratégia e risco do jogo atual.
- Melhorar a experiência de utilizador com componentes como efeitos visuais e um tutorial que assiste os novos jogadores a serem introduzidos aos conceitos principais do jogo e oferecer uma experiência mais *user friendly*.



## 1.5 Arquitetura

Neste capítulo são apresentados com maior detalhe os módulos e características do sistema que permitem atingir os objetivos previamente apresentados.

O sistema está dividido essencialmente em duas partes com igual importância para a solução final: A aplicação cliente e a aplicação servidor.

A aplicação cliente é a interface como o utilizador interage com o sistema e por esse motivo foi necessário ter um maior cuidado com a apresentação e responsividade do mesmo. O servidor é o módulo que serve como fonte de verdade e onde a aplicação cliente obtém todos os dados que necessita.

Como para qualquer servidor com recursos privados, existe também uma componente de autenticação utilizando OAuth com o serviço da Google, no entanto, este não é o foco do projeto e não será abordado. Foi implementado apenas com vista ao controlo de identidade dos jogadores.

Na figura seguinte é apresentado um esquema representativo da arquitetura do sistema, apresentando também as tecnologias utilizadas em cada módulo.

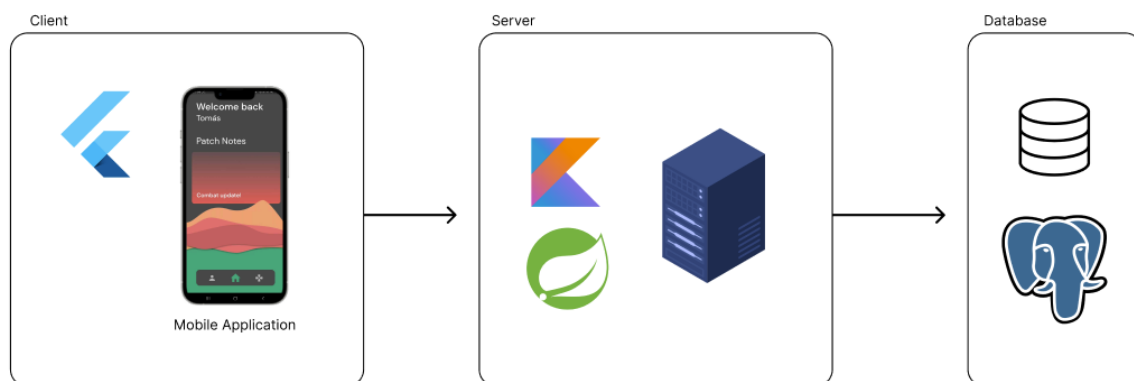


Figura 1.1: Arquitetura do sistema

### 1.5.1 Tecnologias

#### Cliente

Para o desenvolvimento da aplicação cliente escolheu-se a framework Flutter devido á sua funcionalidade multiplataforma, que permite manter a mesma base de código para uma aplicação Android e para uma iOS. Outro aspeto importante na tomada de decisão da framework de cliente a usar foi a qualidade da documentação encontrada na *internet*, assim como a experiência de desenvolvimento oferecida, ambas muito boas na framework escolhida. O último ponto relevante para a escolha foi a sua competência para interfaces complexas e animações. Inicialmente e aquando da escolha da framework foram testadas outras para ser possível tomar uma decisão informada e concluiu-se que o Flutter teve a melhor performance tendo

também em conta o tempo gasto no desenvolvimento.

## Servidor

No servidor decidiu-se manter um stack tecnológico conhecido, nomeadamente a framework Spring, em Kotlin e com base de dados PostgreSQL e que, falando da framework, permite ao programador abstrair-se de alguns detalhes de implementação do servidor e do código de infraestrutura, "passando a responsabilidade" para a framework. Exemplos desta abstração criada são a injeção de dependências e a definição de camadas do servidor, definidas por anotações.

O servidor está dividido em 4 peças, nomeadamente:

- *Interceptors*;
- *Controllers*;
- *Services*;
- *Repository*.

Os *interceptors* utilizados no servidor servem para autenticar o utilizador através do seu token pessoal.

Os *controllers* têm a responsabilidade de receber e interpretar os pedidos HTTP recebidos pelo servidor. São a única camada da aplicação que conhece o protocolo HTTP e comunica diretamente com a camada seguinte, os serviços.

Os *services* são responsáveis pela lógica do servidor; Estes validam os parâmetros recebidos pela camada superior, que conhece HTTP, aplicam qualquer lógica que seja necessária e acedem á camada mais abaixo, de acesso a dados. Esta camada é também responsável por criar e gerir as transações feitas com a base de dados.

O *repository* é a camada de acesso a dados e comunica diretamente com a base de dados PostgreSQL através da biblioteca JDBI.

## Capítulo 2

---

# Abordagem

Durante o desenvolvimento deste projeto foi encontrado um conjunto de problemas e conceitos novos que tiveram de ser pesquisados e resolvidos. Estes tópicos serão apresentados num formato mais teórico destacando a visão e abordagem escolhida, assim como, os desafios ligados a estes conceitos e como estes foram resolvidos.

### 2.1 Geração do mundo

Um dos elementos mais importantes de um jogo de exploração é o mundo a ser explorado. É fundamental que este seja interessante para manter o interesse dos jogadores, e único para permitir *"replayability"* infinita.

Criar um mundo interessante e único é um desafio, seja por meios manuais com uma equipa de *level designers* ou por meios automáticos como o uso de uma combinação de algoritmos com elementos de aleatoriedade.

Optou-se pela utilização de meios automáticos, visto que, o projeto não foi desenvolvido por level designers mas sim programadores. *Procedural Generation*[5] é um assunto de interesse em explorar dado as suas vantagens, que são o controlo dinâmico sobre os possíveis aspetos das ilhas, que dá a possibilidade de observar as alterações feitas no mundo inteiro instantaneamente, assim como as infinitas variações possíveis do mundo.

### 2.1.1 Como?

Uma das técnicas mais populares para gerar mundos é o uso de algoritmos como o *Perlin Noise*[6], que geram um campo de valores aleatórios que são interpolados suavemente entre si. No contexto deste trabalho, estes valores podem ser interpretados como a altura do terreno, como pode ser observado na figura seguinte.

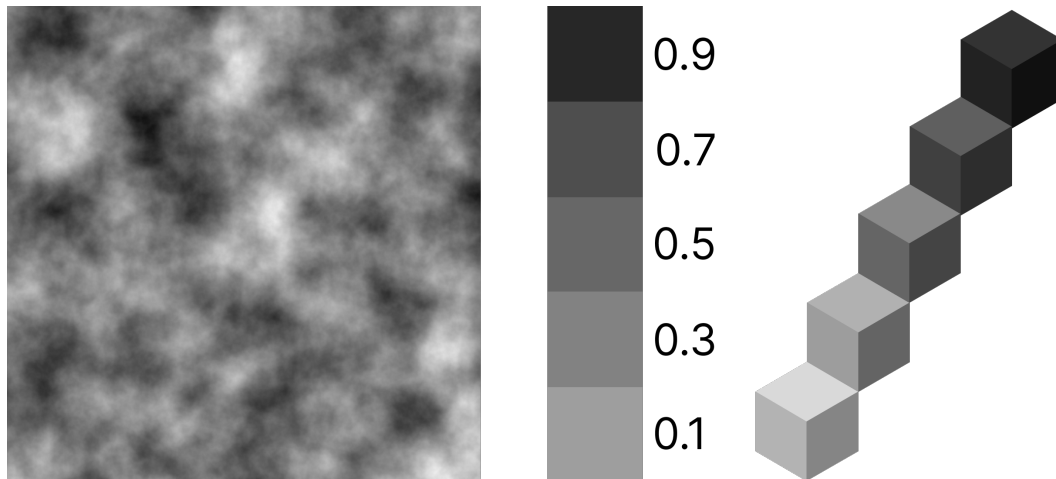


Figura 2.1: Ilustração do algoritmo Perlin Noise

Estas bases permitem a criação de terrenos simples onde facilmente se pode adicionar técnicas por cima para criar todo o tipo de terrenos, desde aplicar várias camadas de *Perlin Noise* com tamanhos diferentes para aumentar os detalhes e variações do terreno, como simular efeitos reais como a erosão no terreno base.

### 2.1.2 Desafios

Um dos desafios em gerar mundos aleatórios de forma automática é obter consistentemente os resultados esperados, isto pois, quanto mais dependemos de algoritmos aleatórios para gerar um terreno menos controlo se tem sobre o mesmo. Assim como também existe a possibilidade de anomalias como repetições e características não naturais nos terrenos.

### 2.1.3 A Nossa Abordagem

A visão do mundo a criar é um conjunto de ilhas em locais aleatórios, onde cada ilha tem um formato simples, mas distinto e único, composta por praias, planícies e montanhas.

Para isto, é fundamental o uso de mais técnicas, que tenham controlo sobre os algoritmos aleatórios para apenas produzir resultados de acordo com a visão em mente.

Descrevendo a abordagem escolhida para a geração do mundo, esta começa com:

1. Escolher pontos aleatórios com uma distância segura no mapa onde serão criadas as ilhas;

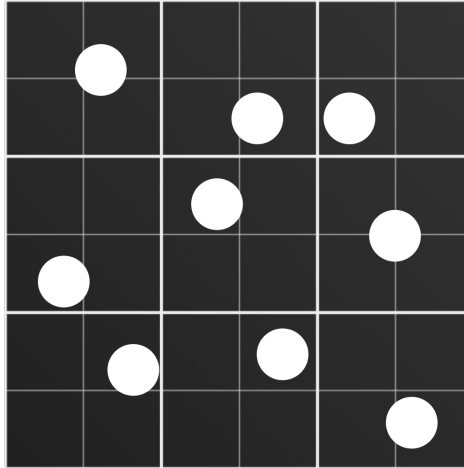


Figura 2.2: Exemplo de um conjunto de ilhas em pontos aleatórios

2. De seguida para cada ilha, escolhemos utilizar o algoritmo *Simplex Noise*[7] que é uma variante do algoritmo *Perlin Noise*, que gera um conjunto de valores aleatórios que vamos utilizar como o terreno da ilha;

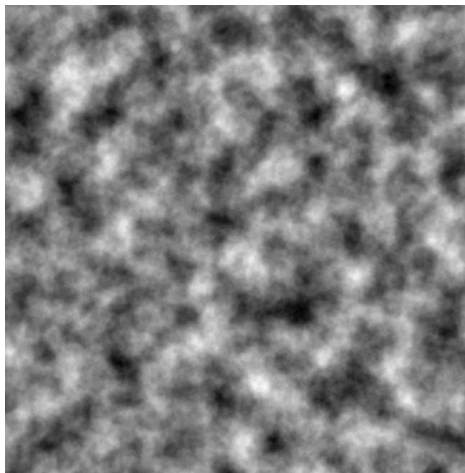


Figura 2.3: Resultado gerado pelo algoritmo Simplex Noise

Foi decidido usar o algoritmo *Simplex Noise* devido às suas vantagens, visto que este é uma versão aprimorada do Perlin Noise que é mais eficiente e gera resultados menos distorcidos a calcular os valores de ruído através da técnica *Simplex*.

3. No final para cada terreno é aplicado um transformação designado de Falloff, que serve para suavizar e reduzir a altura das bordas, garantindo assim que o terreno se conecta ao nível do mar, formando ilhas.

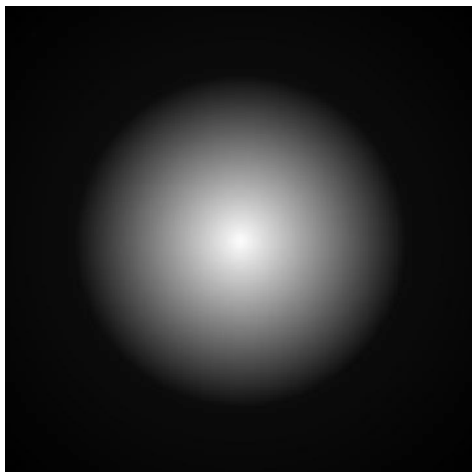


Figura 2.4: Exemplo de uma textura de Falloff gerada

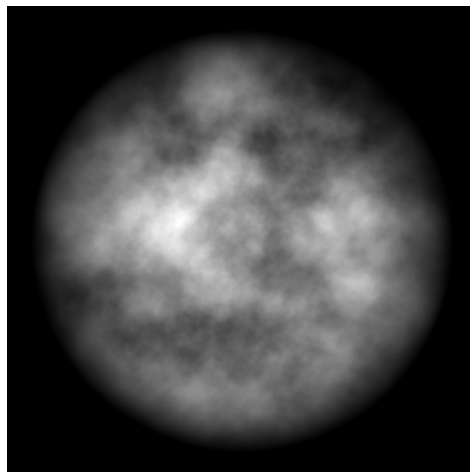


Figura 2.5: Resultado da transformação aplicada ao terreno

Como se pode observar na figura 5, foram gerados valores de 0 a 1 onde 0 é preto e branco é 1. A transformação consiste em multiplicar os valores do Falloff com os valores do terreno. O resultado pode ser observado na figura 6, interpretando os pixels a preto como o mar, pode observar-se como as bordas do terreno da ilha ficaram conectados ao nível do mar como é esperado de uma ilha.

## 2.2 Renderização

De forma a manter uma interface visual apelativa assim como reduzir ao máximo as suas implicações na performance mostrou-se necessário procurar formas atuais e relevantes de renderizar os componentes da *UI*.

### 2.2.1 A Nossa Visão

Os visuais em mente para a aplicação cliente são uma interface gráfica atraente e responsiva, em específico os visuais do jogo são compostos por *voxels* em perspectiva 3D. O nome *Voxel*[8] tem origem na combinação de um pixel dentro de um volume (*Volumetric pixel*), este é um valor definido dentro de um volume como uma gride tridimensional.

São desenhados num *Canvas* infinito apenas os tiles que estão ao redor do navio atual selecionado. Este *Canvas* é interativo e permite dar Zoom In/Out e Drag para arrastar a câmara e observar o redor.

### 2.2.2 A Nossa Abordagem

A perspectiva isométrica[9] é um sistema de representação gráfica que permite representar objetos tridimensionais em duas dimensões, através de uma projeção paralela e ângulos fixos para criar a ilusão de profundidade e volume. Ao contrário de outras perspectivas, que usam pontos de fuga e linhas convergentes para criar a ilusão de profundidade, a perspectiva isométrica usa ângulos iguais de 30 graus para todas as três dimensões: altura, largura e profundidade.

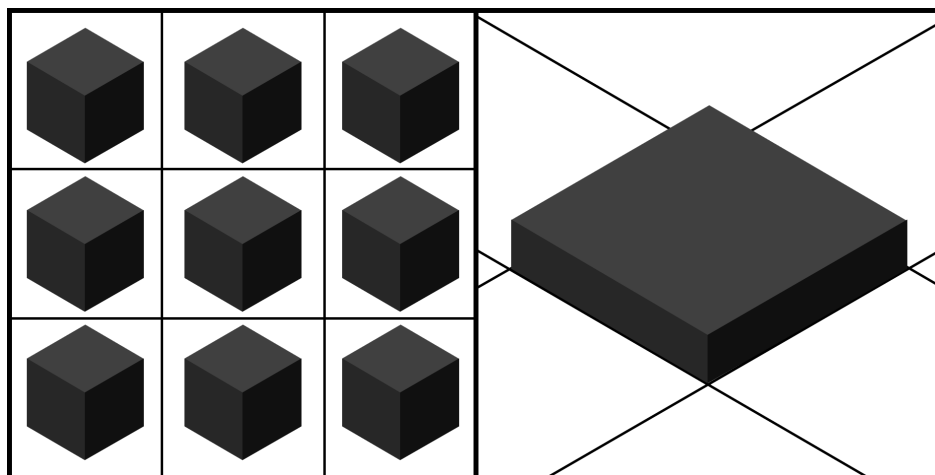


Figura 2.6: Perspetiva ortogonal vs perspetiva isométrica

De forma a representar uma matriz simples em perspectiva isométrica, como observado na figura anterior, é necessário aplicar as devidas transformações, visto que as posições associadas aos pontos encontram-se em perspectiva ortográfica. As transformações necessárias para

alterar uma sequência de blocos em projeção ortogonal para a mesma sequência em projeção isométrica, considerando *Width* como a largura do bloco e *Height* como a altura do bloco, temos:

$$X \begin{pmatrix} 1 * Width/2 \\ 0.5 * Height/2 \end{pmatrix} + Y \begin{pmatrix} -1 * Width/2 \\ 0.5 * Height/2 \end{pmatrix}$$

No que toca à grelha infinita, caso haja uma ilha, são renderizados primeiro blocos de água, e só posteriormente é atualizado o ecrã com os resultados do pedido de visão do redor do barco recebidos. Deste modo, a *UI* tem sempre blocos presentes, não esperando pelas respostas para renderizar o mundo. Na figura seguinte é apresentado um ecrã do jogo de forma a tornar mais claro o assunto tratado.

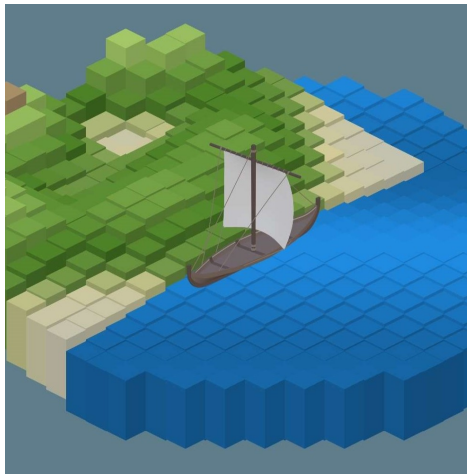


Figura 2.7: Exemplo aplicado da perspectiva isométrica

Um aspeto bastante relevante é o modo como os blocos são renderizados, nomeadamente, através do Flutter Canvas. Numa iteração inicial da solução, utilizou-se imagens para representar estes blocos, tanto em formato *PNG* (*Portable Network Graphics*), utilizando rasterização, como em *SVG* (*Scalable Vector Graphics*), através de vectorização, no entanto notou-se que existiam alguns problemas com esta implementação.

A rasterização[10] e a vetorização[11] são dois métodos de processamento de imagens amplamente utilizados na área da computação gráfica.

A rasterização é um processo que converte objetos em imagens compostas por pixels. Nesse método, os objetos são representados por uma grade de pontos coloridos, em que cada pixel possui uma cor específica. A principal vantagem da rasterização é a sua capacidade de exibir detalhes e efeitos realistas, como sombras, texturas e efeitos de iluminação. No entanto, a rasterização apresenta também algumas desvantagens, sendo uma delas a perda



de qualidade ao ampliar ou redimensionar uma imagem rasterizada.

Por outro lado, a vetorização é um método que utiliza fórmulas matemáticas para representar objetos gráficos através de pontos, linhas e curvas. Ao contrário da rasterização, as imagens vetorizadas são compostas por elementos geométricos definidos, como retas e curvas suaves, em vez de pixels. A principal vantagem da vetorização é a capacidade de edição precisa e flexível. É possível manipular cada elemento individualmente, alterando a sua cor, forma ou posição. Mas a vetorização também tem as suas limitações, podendo ser menos eficiente em termos de processamento para imagens com muitos detalhes, ou no caso deste projeto em específico, na existência de muitas imagens, uma vez que requer cálculos matemáticos complexos para renderizar os objetos.

A solução encontrada para este problema baseia-se na criação de formas de calcular as cores de um bloco baseadas na sua altura e através de um gradiente de cores definido na aplicação. Decidiu-se tomar inspiração e basear a implementação no *node ColorRamp*[12] do programa de design gráfico em 3D *Blender*[13]. Este node funciona como uma função de interpolação onde para um valor "T" é retornado um valor intermédio do gradiente definido. Para a nossa implementação, interpretamos o valor "T" como a altura do bloco. Na figura seguinte é apresentada uma captura de ecrã desta *ColorRamp* para mais fácil percepção.

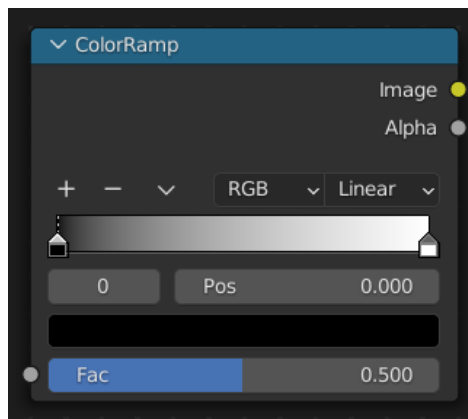


Figura 2.8: Node ColorRamp do Blender

Quer isto dizer que existe total controlo sobre as cores utilizadas no jogo, sem necessidade de carregar novas imagens ou outros *Assets*. Isto permite alterações de cor muito súbtis e interessantes conforme a altura do bloco, permitindo também até alterações da cor dos blocos de água conforme a sua altura, controlada internamente através de uma animação.

Na figura seguinte é apresentado o gradiente de cores utilizado, tendo em conta que este gradiente deve ser interpretado da esquerda para a direita, quer isto dizer que quanto mais á esquerda é a cor, mais baixos serão os blocos associados.



Figura 2.9: O nosso gradiente de cores

### 2.2.3 Desafios

Os maiores desafios no que toca á parte de renderização prenderam-se em encontrar maneiras de obter toda a interface visual idealizada na aplicação sem incorrer em riscos significativos de performance, visto que é necessário representar um número significativo de blocos com diferentes cores e animações, no caso de blocos representativos de água.

Outro desafio significativo foi conseguir alterar "dinamicamente" a cor dos blocos conforme a altura de cada um, tendo em conta que o carregamento de imagens da memória tem os seus custos, assim como o processamento para obter a imagem desejada.

## 2.3 O Movimento Dos Navios

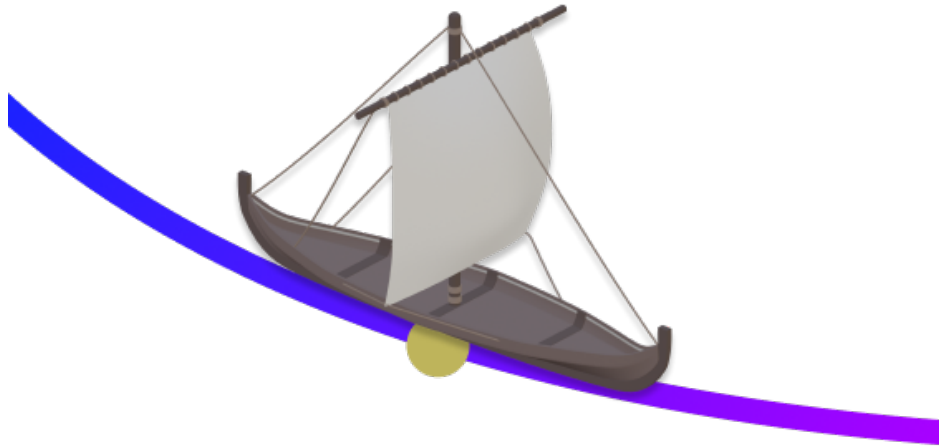


Figura 2.10: Ilustração de um navio

A frota de navios é a representação virtual do jogador no mundo; Cada navio é uma unidade independente que é controlada diretamente pelo jogador. Através dos navios é possível progredir e impactar o ecossistema assim como o mundo do jogo.

A principal função dos Navios é Navegar, esta ação permite tanto observar e descobrir novos territórios ao redor do mundo, como também interagir com outros jogadores e as suas frotas.

### 2.3.1 Movimento Em Jogos Multijogador

No movimento de jogos multijogador é preciso garantir que a posição das entidades controladas pelo jogador sejam confiáveis, sejam estas um personagem ou uma frota de navios.

Isto é alcançado através do modelo de autoridade do servidor, ou seja, o servidor é responsável por validar e calcular todas as ações e movimentos das entidades. Em vez de aceitar diretamente a posição enviada pelo jogador que pode ter sido manipulada, o servidor recebe apenas os inputs do jogador, como comandos de movimento ou ações específicas.

Para não existir *delay* entre estes cálculos, a aplicação cliente também realiza uma estimativa da posição com base nos inputs do jogador localmente. Isto é feito para fornecer feedback imediato ao jogador.

Assim que o servidor conclui e envia o cálculo da posição do jogador, a aplicação cliente verifica se esta difere da posição estimada pelo cliente e, se forem diferentes, altera a posição atual para a posição calculada pelo servidor.

No entanto, é importante ter em conta que enviar constantemente pedidos para alterar e observar a posição do jogador é pesado no servidor, e normalmente requer um maior número

de instâncias do servidor a correr.

### 2.3.2 A Nossa Visão

Ao contrário de muitos jogos de estratégia baseados em grelhas, optou-se por um movimento mais livre em qualquer direção que não respeitasse a grelha. Na escolha do tipo de movimento, foi valorizado este ser suave e fluído de um bloco ao outro, ter um método flexível e competente para caminhos elaborados, e que não causasse grande carga no servidor.

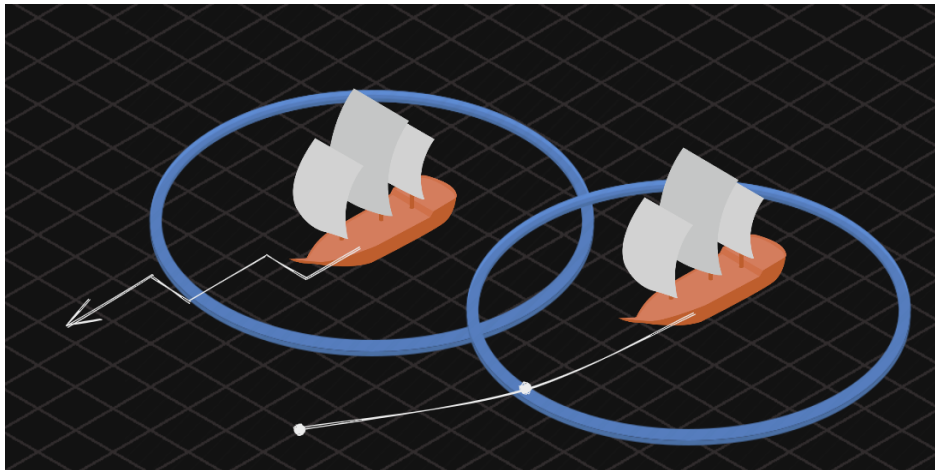


Figura 2.11: Tipos de movimento

Nesta figura é possível observar na esquerda um tipo de movimento que respeita a grelha do jogo, e na direita o tipo de movimento que não respeita a grelha e escolhemos implementar.

### 2.3.3 A Nossa Solução: Curvas De Bézier

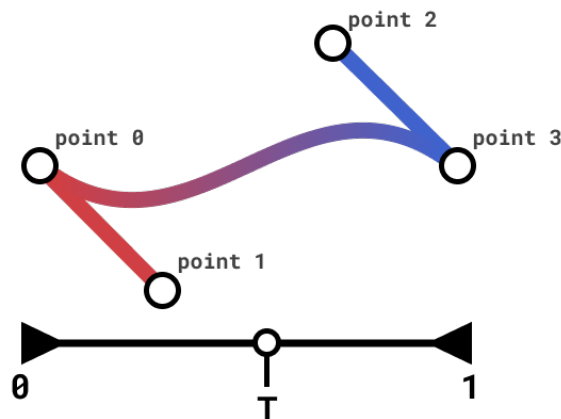


Figura 2.12: Ilustração de uma Curva de Bézier cúbica

Uma *Curva de Bézier*[14] é uma curva suave e flexível definida por uma função de interpolação que pode assumir diferentes formas dependendo do número de pontos que a define, como interpolação linear, interpolação polinomial ou interpolação por *splines*. As *Curvas de Bézier* são amplamente usadas em design gráfico, modelagem computacional e animações gráficas.

Para dar contexto, uma função de interpolação é uma função matemática utilizada para estimar valores intermédios entre pontos de dados conhecidos. A variável "T" é frequentemente utilizada nestas funções para representar um parâmetro que varia de 0 a 1, indicando a posição relativa entre os pontos de dados.

Com esta ferramenta podemos construir caminhos elaborados em qualquer direção, e interpretar a variável T como o tempo para animar objetos, no nosso caso navios, ao longo da curva.

Outra propriedade útil das *Curvas de Bézier* é que é possível obter o ângulo da tangente de um ponto ao longo da curva, e através deste ângulo, saber em que direção desenhar o navio.

Construindo o caminho no servidor conseguimos garantir que este caminho é confiável e partilhando este com a aplicação cliente, cada um consegue observar a posição atual ao longo do caminho independentemente, sem pedidos intermédios. Este método é seguro, eficiente e reduz consideravelmente a carga no servidor.

#### 2.3.4 Obstáculos

Para construir caminhos complexos é necessário usar vários pontos, no entanto, a performance das *Curvas de Bézier* é proporcionalmente menor quantos mais pontos a definirem, pois a complexidade da função de interpolação aumenta.

Para além da performance, outro problema é a falta de controlo local, ou seja, a alteração de um único ponto, tem influência ao longo de toda a curva, o que não é o desejado, é importante termos controlo local e apenas manipularmos um segmento da curva, caso contrário, torna-se um desafio criar caminhos complexos.

A nossa solução para estes obstáculos foi de em vez de usar uma única curva com  $n$  pontos, juntar várias curvas de menor complexidade, no nosso caso decidimos usar as curvas cúbicas definidas por 4 pontos que são o tipo de curva mais usado pela sua simplicidade e competência.

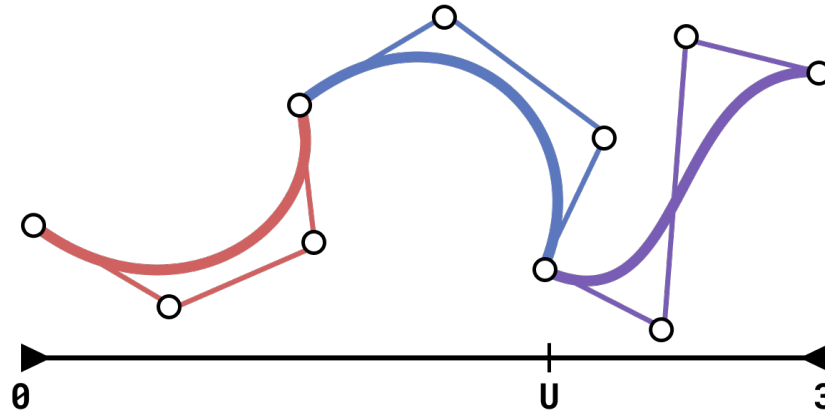


Figura 2.13: Exemplo de um caminho complexo

Nesta figura, podemos observar 3 curvas cúbicas e uma variável "U" que varia entre 0 e o número de curvas cúbicas. Esta variável é o valor de interpolação global, e serve para obter os valores intermédios entre o conjunto das curvas. Para obter os valores intermédios da curva através da varivel "U" fazemos:

$$T = \text{remainder}(U, 1);$$

$$I = \text{floor}(U);$$

Através da variavel "T" temos o valor de interpolação local da curva com index igual ao da variavel "I".

Assim resolvemos o problema do controlo local, visto que alterar um ponto apenas tem influência na curva a que esse ponto pertence, como o problema da performance, pois não importa o número de pontos, a complexidade de obter os valores intermédios permanece constante.

Anteriormente falamos de como era possível usar a variável "T" para animar objetos ao longo da curva, calculando a posição correspondente ao valor atual de "T" que incrementamos com o tempo. Um detalhe importante que não pode ser negligenciado é que com este método, a distância dos pontos entre si influencia diretamente a velocidade do objeto, ou seja, se ao longo da curva, dois pontos estiverem mais distantes um do outro, então a velocidade do objeto quando estiver entre esses dois pontos vai ser maior.

Isto é um problema, para o movimento dos nossos barcos é importante que o movimento seja uniforme, como seria num barco de verdade em condições ideais e constantes.

Considerámos duas maneiras de resolver este problema:

- Alterar os calculos de obtenção dos pontos intermédios para ter em conta a distância entre os pontos;
- Certificar-nos que as caminhos construídos tem pontos com distâncias equivalentes entre si.

Decidimos escolher o segundo método, pois não vimos mérito em alterar e aumentar a complexidade dos cálculos de obtenção dos pontos intermédios quando podemos resolver este problema com um método mais simples e eficaz.

## 2.4 A Criação dos Caminhos

A necessidade da criação das *Curvas de Bezier* provém da navegação dos navios. A ação de navegar desencadeia o cálculo de um caminho composto por pontos entre o início e o destino, mas de modo a obter estes pontos mostrou-se necessária a utilização de um algoritmo de procura de caminhos eficiente e competente.

### 2.4.1 Como?

Um dos algoritmos mais utilizados na indústria dos jogos para resolver este problema é o  $A^*$ [15], um algoritmo de pesquisa informada e que procura a solução de forma mais curta, tendo em conta não só os caminhos possíveis, mas também os caminhos mais apropriados. Este é calculado utilizando uma heurística, que depende de caso para caso e guia a busca priorizando os caminhos que parecem ser mais promissores. Isto reduz significativamente o número de nós a serem explorados, o que faz deste algoritmo um dos mais eficientes para este tipo de problemas.

### 2.4.2 Requisitos e Abordagem

Um dos requisitos para a navegação foi que o utilizador tivesse um bom nível de controlo sobre as rotas traçadas, para isto criou-se um ponto intermédio que pode ser alterado e que tem influência no caminho final. Outro requisito também importante foi encontrar um algoritmo que permitisse desviar o caminho de alguns obstáculos, que neste contexto, são ilhas. Escolheu-se por uma questão de experiência de utilização que os caminhos traçados se desviassem das ilhas já conhecidas pelo jogador.

Através do algoritmo  $A^*$  mostrou-se possível não só encontrar o caminho mais rápido, que é de forma geral o seu propósito principal, como também influenciar este caminho com o ponto intermédio. Para isto criou-se uma heurística que tem em conta não só a distância desde o ponto atual até ao ponto final, mas também a distância do ponto atual até ao ponto intermédio, embora que com um peso para o valor final da heurística mais reduzido.

Desta forma é possível aplicar uma *penalty* aos pontos vizinhos encontrados que não se aproximem do ponto intermédio e fazer com que o caminho encontrado se desvie em direção do ponto desejado.



Na figura seguinte, é apresentado um exemplo do caminho traçado pelo algoritmo  $A^*$ , com base no caminho escolhido pelo utilizador, desviando-se de uma ilha existente e conhecida. A curva presente na figura começa no barco e termina no ponto final da viagem e representa a curva escolhida pelo utilizador e visualizada na aplicação. O conjunto de pontos a preto representa o caminho gerado pelo algoritmo.

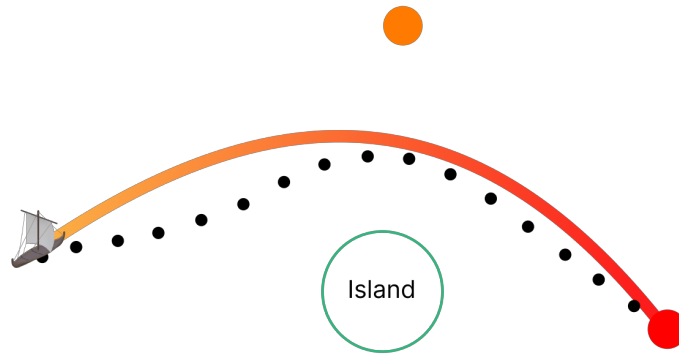


Figura 2.14: Exemplo do resultado do algoritmo de procura

É importante destacar que foram realizadas algumas alterações ao algoritmo  $A^*$  genérico. Foi implementado na criação de nós vizinhos uma distância de segurança com obstáculos, para que o caminho não se aproxime demasiado das ilhas. Da mesma forma, na criação de nós vizinhos foi adicionado um *step* que salta os nós vizinhos mais próximos, pois chegámos a conclusão que não era necessário uma grande frequência de nós para os nossos caminhos. Isto resulta num algoritmo customizado para a nossa solução mais rápido e também mais simples.

O resultado da execução deste algoritmo contém o caminho encontrado como um conjunto de pontos a cada *step*, no entanto estes pontos não estão prontos para serem usados como *Curvas de Bézier*. É fundamental que o conjunto de pontos seja divisível por quatro para serem criadas curvas de Bézier cúbicas pois estas são compostas por quatro pontos. Para isto, é dividido o caminho encontrado em segmentos de 4 pontos todos a igual distância uns dos outros. Tendo então o caminho encontrado e dividido em segmentos de igual comprimento, é possível representá-lo através de um conjunto de Curvas de Bezier.

### 2.4.3 Desafios

Um dos problemas típicos dos algoritmos de procura de caminhos em espaço de estados é o problema do mínimo local. Este problema é mais concretamente relacionado com a procura de pontos vizinhos (ou pontos sucessores).

Dado que este é um algoritmo de pesquisa informada e que procura os pontos sucessores através de um valor calculado segundo a sua distância ao ponto inicial e o seu valor de heurística, é possível que os pontos vizinhos possíveis para avançar no caminho não sejam elegíveis para o algoritmo, por serem um retrocesso no caminho. Na sua implementação mais comum, o algoritmo não permite retrocessos, e é possível que fique bloqueado num ponto cujos sucessores não são elegíveis, que neste contexto, poderão ser pontos de ilha. Devido a este facto, foi necessário ter em conta este problema na implementação do algoritmo.

## 2.5 Eventos

A ação primária do jogador é a de "Navegar" para um destino determinado pelo jogador, esta ação pode ter consequências, como:

- Encontrar uma ilha pelo caminho, o que resulta no barco ficar imóvel até uma nova ação de navegar.
- Encontrar um barco inimigo, resultando numa interação de luta enquanto continuam a navegar para os seus destinos.

Estes são os dois tipos de eventos que podem ser produzidos pelos jogadores.

A utilidade de um sistema de eventos cuja estrutura descreve as consequências das ações feitas, é podermos obter as estatísticas, construir o ranking do jogador a partir destes dados, observar o estado passado do jogo e consequentemente ter também a possibilidade de fazer um replay desde o início do jogo até ao momento presente.

### 2.5.1 Como?

Para um barco "Encontrar" outra entidade seja este uma ilha ou um navio, este precisa de estar perto desta entidade.

No entanto, como eventos resultam da ação de "Navegar", então o navio vai encontrar-se em movimento até chegar ao seu destino.

*Como é que podemos verificar se o navio está perto de alguma entidade durante o seu trajeto quando algumas destas entidades, os barcos, se encontram em movimento?*

Uma solução considerada seria usar um sistema de *Job Scheduler*[16], que é uma componente para criar e controlar um conjunto de tarefas a um determinado tempo e frequência. Para criar uma tarefa (*Job*) que fosse executada muito frequentemente e verificasse se algum dos barcos se encontra perto de alguma entidade.

### 2.5.2 Desafios

Como referido ao longo deste documento, um dos nossos maiores objetivos para este projeto é fazer deploy, o que significa ter um servidor hospedado. Para enfrentar este objetivo realisticamente é preciso pensar nos seus custos.

Hoje em dia, a maioria dos serviços de hospedagem cobram pela quantidade de pedidos recebidos ou pela quantidade de tempo que um servidor permanece ligado.

Estes serviços desligam os servidores quando se encontram inativos, ou seja, quando não são recebidos pedidos dentro de um certo intervalo de tempo ou não existam jobs a serem executados.

Recuando á solução anteriormente referida, utilizando um sistema de *Job Scheduler*, esta seria não só cara pois obrigaria o servidor a permanecer sempre ligado devido a *Jobs* executados muito frequentemente, como pela quantidade de pedidos que teriam de ser feitos para observar o estado atual dos eventos, como também pesada, visto que o servidor teria de realizar constantemente várias operações não triviais como obter as posições atuais dos barcos e calcular as suas distâncias.

### 2.5.3 A Nossa Abordagem

Depois de muita reflexão neste assunto, decidimos abordar o problema de uma maneira completamente diferente.

Em vez de verificar a criação de eventos ao longo do caminho de um barco, calcula-se uma única vez ao efetuar-se um *Navigate*, caso ao longo do caminho forem criados eventos.

Cada evento é composto pelos seus detalhes e o instante quando este ocorre.

Esta técnica torna a utilidade de usar um *Job* obsoleto, otimizando e permitindo ao servidor ficar inactivo. Outra vantagem deste método é utilizar os eventos para agendar tarefas na aplicação cliente, como:

- Observar o estado atual do barco apenas quando existe um estado novo, ou seja, depois do instante do evento, assim deixa de ser necessário realizar *Pooling*;
- Agendar notificações de eventos no dispositivo mobile.

No entanto a complexidade dos cálculos para verificar a criação dos eventos aumentou consideravelmente, já não é tão simples como *verificar se algum barco está perto de outra entidade*.

Para construir os **Island Events** usamos as seguintes técnicas:

1. Simplificamos a *Curva De Bézier* num conjunto de linhas sem qualquer curvatura para simplificar os cálculos de interseções, verificar interseções em *curvas de Bezier* é uma operação cara que envolve matrizes complexas e concluímos que conseguimos ter resultados equivalentes usando uma técnica mais simples;

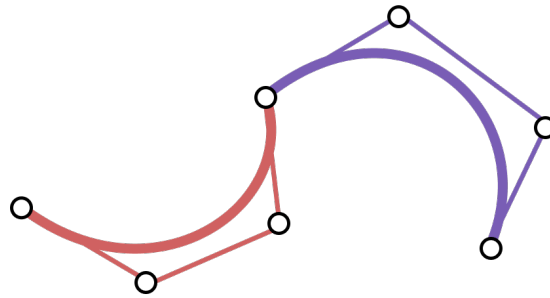


Figura 2.15: Ilustração da simplificação da Curva de Bézier

2. Procuramos o primeiro ponto de interseção desde o início do caminho até ao fim, entre o conjunto de linhas e o conjunto de ilhas;
3. Isolamos o segmento da *Curva De Bezier* interseçada e tiramos uma amostra de 10 pontos pertencentes a esse segmento;

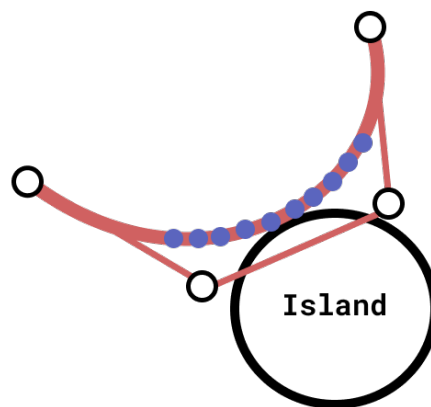


Figura 2.16: Exemplo de uma amostra de pontos da interseção

4. Procuramos o ponto (e o instante) que está mais próximo da ilha, mas não dentro dela. Fazemos isto, em vez de simplesmente pegar no ponto de interseção, para obter uma posição perto o suficiente da ilha mas fora dela, para evitar o navio de subir a ilha.

Para construir os **Fight Events** usamos a seguinte técnica:

1. Obtemos o movimento atual, ou seja, o movimento construído para o instante atual do nosso e do barco inimigo;
2. Apartir do movimento de cada um, simplificamos a *Curva De Bézier* num conjunto de linhas sem qualquer curvatura e depois a partir dessas linhas construímos planos.

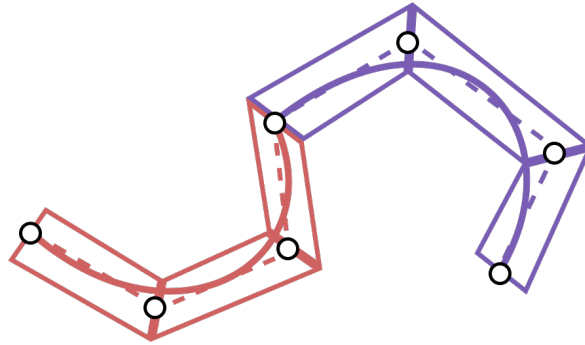


Figura 2.17: Ilustração dos planos de colisão construídos para a Curva de Bézier

Como referido anteriormente, para verificar interseções foi escolhido usar uma técnica mais simples e menos custosa e por isso optámos por simplificar para um conjunto de linhas sem curvatura. No entanto diferente de anteriormente, é construído para cada linha um plano.

Desta forma no caso de uma interseção entre dois caminhos que não se cruzam mas passam extremamente perto um do outro na mesma altura, não vai ser criado nenhum evento. Como isto não é o resultado desejado, são usados planos em vez de linhas para existirem interseções nestes casos também.

3. Através dos caminhos construídos são procuradas interseções no mesmo período temporal entre cada plano. Sabendo que o navio é uma entidade em movimento ao longo do caminho, não basta existir uma interseção entre os planos para existir um *Fight Event*.

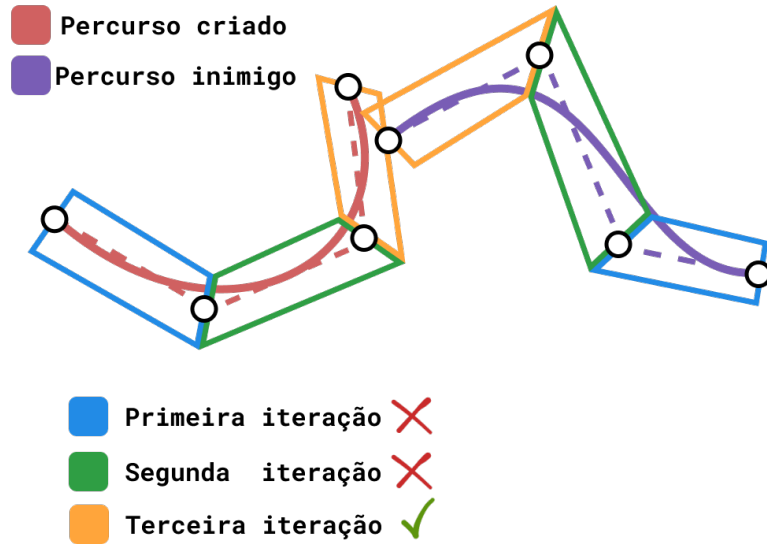


Figura 2.18: Exemplo entre a interseção de dois caminhos

Como se pode observar nesta figura, os planos dos caminhos são agrupados por ordem temporal, isto é, o grupo azul é o plano onde os navios atualmente se encontram, e eventualmente vão estar nos grupos verde e amarelo. Desta forma são apenas verificadas interseções entre os planos do mesmo grupo, pois seria incorreto comparar dois grupos com uma diferença de espaço temporal entre eles, ou seja, acontecem em instantes separados.

Isto é uma técnica chamada *Space Partitioning*[17] que é comum ser utilizado para otimizar algoritmos de colisão em simulações.

4. Para cada interseção, é obtida uma amostra de 5 pontos de cada plano intersetado onde pesquisamos os dois pontos que estão mais perto um do outro no espaço tal como no tempo.
5. Apartir deste ponto, temos o ponto onde o **Fight Event** ocorre e o seu instante, é depois escolhido aleatoriamente um vencedor.
6. Repetir os mesmos passos para os restantes barcos.

## 2.6 Histórico dos pontos visitados

Dado que um grande foco de projeto é a exploração do mapa por parte dos jogadores, é necessário ter em conta tanto a experiência de navegação como também persistir a informação de onde os barcos de um utilizador já passaram, de modo a que seja possível manter um histórico de progresso e de navegação na sala em que o utilizador está a jogar.

Neste tipo de projeto, é também preciso ter em conta a experiência a que o utilizador está acostumado de modo a não ser uma mudança que o faça não gostar da solução. Para garantir este aspeto foi feito um estudo de outros jogos do mesmo género, sabendo que estes têm sucesso e uma *UX* agradável e conhecida.

### 2.6.1 Como?

De modo a apresentar fielmente os caminhos feitos pelo utilizador, é utilizado o conjunto de pontos que define os caminhos traçados para cada barco, guardado na base de dados, o que significa que é possível obter as rotas por onde este passou e enviá-las para o cliente.

De forma a conseguir representar as ilhas já conhecidas pelo utilizador, é utilizado o sistema de eventos previamente explorado no documento. Deste modo é possível saber quais os eventos de ilha que já aconteceram até ao momento em que o utilizador obtém o minimapa e enviar na resposta as ilhas encontradas.

### 2.6.2 A Nossa Abordagem

Persistir o histórico de navegação de forma eficiente em termos de complexidade de espaço ocupado assim como em termos de complexidade de tempo de computação necessário para o obter é um desafio complexo e é necessário encontrar um equilíbrio entre estes dois aspetos de forma a obter uma solução otimizada, apropriada e, o mais importante, correta.

Elaborando um pouco na forma como os dados são tratados, à aplicação de cliente chega um objeto que contém os caminhos percorridos representados através de pontos com 2 coordenadas, e o conjunto de ilhas descobertas, caso existam.

As ilhas são representadas através de objetos que contêm 3 coordenadas, de modo a ser possível representar a sua altura. Para obter as ilhas conhecidas, é comparado o instante no qual acontecerá o evento de ilha com o instante corrente e verifica-se se o evento já aconteceu.



A tarefa do cliente é apenas representar as ilhas recebidas e através do conjunto de caminhos, fazer *pulse* em cada um destes pontos. Um *pulse* define-se visualmente como um círculo cujo centro é o ponto recebido. Este *pulse* pode ser feito visto que que em torno de cada um daqueles pontos, não existe mais nenhuma informação, e portanto pode ser imediatamente considerado como água.

Um aspeto bastante importante sobre este sistema de guardar os pontos, que se relaciona também com o uso das curvas de Bezier é que um utilizador pode alterar o caminho de um barco antes do mesmo terminar o caminho corrente. Isto implica que não é correto representar no mini mapa todos os pontos do caminho mas sim apenas os que foram vistos até à criação do novo caminho. Com a utilização das *Curvas De Bézier* tornou-se possível recortar a curva no ponto que corresponde ao início do próximo caminho e desta maneira apenas representar os caminhos que foram efetivamente visitados.

Para abordar o problema de obtenção das ilhas optou-se pela utilização do sistema de eventos previamente abordado. Através deste sistema é possível ao servidor saber quais são as ilhas visitadas pelo utilizador.

Nas figuras seguintes são apresentados o minimapa atual, com os caminhos recortados á medida certa, e o minimapa sem tratamento.

Ambas têm uma ilha representada e é de notar que são exatamente o mesmo caminho, sendo a única diferença, que a figura da esquerda apresenta os caminhos recortados nos sítios devidos e a figura da direita apresenta os caminhos completos sem qualquer tipo de tratamento.



Figura 2.19: Mini mapa com os caminhos recortados



Figura 2.20: Mini mapa sem tratamento do comprimento dos caminhos

### 2.6.3 Desafios

Os principais desafios da implementação deste sistema prenderam-se com a forma de guardar e obter as informações acerca do caminho.

Mostrou-se necessário encontrar uma forma de enviar apenas os pontos mínimos necessários a uma boa representação em vez de todos os presentes num caminho, isto porque num caso extremo em que o utilizador já conhece o mapa inteiro, seria necessário enviar todos os pontos do mapa. Isto seria bastante custoso visto que um mapa terá em média  $600 \times 600$  *tiles*, o que resulta em 360.000 pontos. Mesmo embora não se enviassem exatamente todos, seria um número bastante considerável e que tornaria as respostas aos pedidos *HTTP* demasiado grandes e morosas de processar.

Numa iteração inicial existiu um sistema que guardava pontos visitados aquando de um pedido por parte do cliente que devia apenas ser respondido com os blocos relevantes á volta da posição do barco. Nesta iteração o cliente seria obrigado não só a realizar um *pulse* por ponto, mas sim a arranjar uma representação visual para os ligar. Isto foi feito criando elipses de 2 em 2 pontos, no entanto, não seria uma representação perfeita visto que as elipses se apresentariam mais largas do que deviam no eixo maior, e demasiado próximas ao ponto nos extremos.

Também foi abordada e testada outra aproximação ao problema da representação, através de *pill shapes*. Estas seriam visualmente mais fieis ao caminho e apresentam-se como fazendo um *pulse* em cada ponto, e unindo-os através de um retângulo. Ambas estas abordagens foram abandonadas devido á complexidade de cálculos desnecessária.

Isto provou-se altamente ineficiente e incorreto visto que, em primeiro lugar os pontos só seriam guardados se a aplicação cliente estivesse aberta ao longo do caminho, em segundo envolviam vários cálculos a cada pedido de modo a verificar se o ponto presente do barco deveria ser guardado, ou se estaria demasiado perto do último guardado. Este sistema não permitia também saber com exatidão que ilhas já teriam sido visitadas.

Outro desafio relevante, já previamente explorado, foi apenas representar os caminhos efetivamente visitados, recortando os mesmos até ao início do próximo caminho ou até a um possível evento de ilha.

## Capítulo 3

---

# Conclusão

### 3.1 Considerações

Com a conclusão deste trabalho foram aplicadas e demonstradas as habilidades técnicas, conhecimento e pensamento crítico para enfrentar desafios complexos como os apresentados ao longo deste documento. Foi conseguido não só encontrar soluções para os problemas encontrados, como também justificar e documentar a solução tomada em comparação com outras soluções considerando as vantagens e desvantagens de cada um. Este processo envolveu a pesquisa e aprofundamento de conhecimento de conceitos relevantes para o trabalho documentado como para a área de engenharia de Software no geral. No desenvolvimento do projeto foram trabalhadas as habilidades de desenhar e planear o desenvolvimento de um projeto dentro de um tempo limite, o pensamento crítico para tomar decisões e justificá-las, e a gestão do tempo das tarefas por completar. Alguns dos conceitos explorados neste trabalho foram *Procedural Generation*, Algoritmos pseudo aleatórios de ruído como o *Perlin Noise* e o *Simplex Noise*, funções de interpolação como as *Bézier Curves*, algoritmos como o  $A^*$  e "builders" de recursos.

Tendo em conta que se optou por não utilizar uma *game engine*, mas sim uma *framework* de desenvolvimento de aplicações multiplataforma, é necessário ter em conta os limites da *framework*, visto que a tarefa de representar visualmente um número muito grande de elementos não é tarefa trivial este não é o foco principal da *framework*. Mesmo utilizando as formas mais rápidas de representação, como desenhar diretamente no *Canvas*, pode trazer problemas de performance.

Após o desenvolvimento percebeu-se rapidamente que a quantidade de blocos desenhados e animados tem um impacto relevante na performance e no número máximo *FPS* (*frames per second*) atingidos. Devido a este facto mostrou-se necessário ter atenção redobrada à quantidade de informações e blocos presentes no ecrã. Seria incomportável, por exemplo, ter uma grelha de blocos visíveis que não acabasse ou até que ocupasse toda a extensão do ecrã, visto que caso fosse o caso e o utilizador fizesse *zoom out*, o número de blocos renderizados

escalaria consideravelmente.

É relevante também referir que todas as decisões foram tomadas dentro de um tempo limite, tendo em conta o seu impacto, devido à pequena janela temporal de desenvolvimento do projeto. Antes de tomar qualquer decisão é importante explorar diversas opções e resoluções do problema apresentado, de forma a ser possível tomar uma decisão informada. Pelo mesmo motivo, e por não ser um foco atual não foi priorizada uma experiência de jogo equilibrada, nem mais funcionalidades opcionais, que num projeto de engenharia, têm sempre espaço.

## 3.2 Dificuldades

Foi encontrado ao longo do desenvolvimento deste trabalho um conjunto de dificuldades, algumas comuns neste tipo de projeto e outras mais específicas a este projeto.

Para uma melhor compreensão do trabalho realizado, são apresentadas as dificuldades que merecem maior ênfase:

- Para a aplicação cliente foi usada a framework Flutter, que embora fosse a (nossa) primeira interação com esta framework o verdadeiro desafio não foi a "aprender", mas sim construir um jogo e os seus componentes visuais nela. Esta dificuldade já era expectada, visto que esta framework não foi feita especificamente para jogos.
- Foram também encontradas dificuldades na matemática espacial especificamente os cálculos que envolviam o espaço como posições. Ao longo do desenvolvimento deste projeto foi necessário trabalhar com conceitos como posições globais e locais, posições em perspetiva global e em perspetiva isométrica e as suas características como tamanho e transformações.
- Uma dificuldade contínua ao longo deste trabalho foi encontrar soluções realistas para os problemas encontrados. Soluções que se possam aplicar em meios e condições reais, dentro das limitações tecnológicas e económicas. Estas dificuldades são principalmente devido à carga que jogos RTS (Real Time Strategy) impõem dos servidores.

Estas dificuldades foram anteriormente exploradas e documentadas em maior detalhe através de problemas reais e as suas soluções.

### 3.3 Trabalho Futuro

Devido à dimensão do projeto escolhido e aos constrangimentos de tempo inerentes ao mesmo ser realizado num período de tempo de um semestre e a equipa de desenvolvimento ser pequena para um projeto deste género, mostrou-se impossível concluir todas as tarefas opcionais e adicionais que haviam sido planeadas.

Como já foi referido no presente documento, este projeto foi pensado com intenção em continuar a ser desenvolvido após o término da unidade curricular. Para seguir em frente com o projeto desenvolvido está planeado implementar mais funcionalidades para construir um produto mais completo e indicado para ser publicado nas lojas digitais. Enumera-se então uma lista de tarefas para desempenhar no futuro.

- Aumentar a cobertura dos testes automáticos tanto no back-end, como testes de *UI*;
- Aumentar a complexidades das ilhas com outros tipos de ilhas e mais interações.
- Sistema de amigos, de modo a ser possível ver o progresso dos amigos do jogador e ser mais fácil entrar em salas em que amigos já estão a jogar;
- Animações ou efeitos visuais durante batalhas entre navios;
- Adicionar música e efeitos sonoros de modo a tornar a experiência mais agradável;
- Testar com um maior número de jogadores para compreender e conseguir equilibrar o jogo e a sua economia;
- Adicionar uma loja dentro de jogo com *skins* e alterações de aspeto, utilizando micro-transações, de forma a tornar o jogo autossustentável ou até rentável;
- Distribuir o jogo pelas principais lojas digitais.
- Implementação da aplicação Web, também em Flutter;

A maioria dos pontos enumerados anteriormente são melhorias à experiência de jogo dos utilizadores e não fazem parte do produto mínimo pensado aquando do início do desenvolvimento do projeto.



---

# Referências

- [1] Flutter documentation. <https://docs.flutter.dev/>. Accessed: 2023-05-31.
- [2] Spring boot documentation. <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>. Accessed: 2023-05-31.
- [3] Rts games. <https://en.wikipedia.org/wiki/Real-time>. Accessed: 2023-05-31.
- [4] Server sent events documentation. <https://developer.mozilla.org/en-US/docs/Web/API/Server-sent/Using-sent>. Accessed: 2023-05-31.
- [5] Procedural generation. [https://en.wikipedia.org/wiki/Procedural\\_generation](https://en.wikipedia.org/wiki/Procedural_generation). Accessed: 2023-05-31.
- [6] Perlin noise algorithm. <https://en.wikipedia.org/wiki/Perlin>. Accessed: 2023-05-31.
- [7] Simplex noise algorithm. <https://en.wikipedia.org/wiki/Simplex>. Accessed: 2023-05-31.
- [8] Voxel. <https://pt.wikipedia.org/wiki/Voxel>. Accessed: 2023-05-31.
- [9] Isometric perspective. <https://en.wikipedia.org/wiki/Isometric>. Accessed: 2023-05-31.
- [10] Rasterização. <https://en.wikipedia.org/wiki/Rasterisation>. Accessed: 2023-05-31.
- [11] Vectorização. <https://en.wikipedia.org/wiki/Vector>. Accessed: 2023-05-31.
- [12] Blender color ramp node. <https://docs.blender.org/manual/en/latest/render/shader/converter/color.html>. Accessed: 2023-05-31.
- [13] Blender project. <https://www.blender.org/>. Accessed: 2023-05-31.
- [14] Bézier curve. <https://en.wikipedia.org/wiki/Bézier>. Accessed: 2023-05-31.
- [15] A\* search algorithm. [https://en.wikipedia.org/wiki/A\\*](https://en.wikipedia.org/wiki/A*). Accessed: 2023-05-31.
- [16] Job scheduler. <https://en.wikipedia.org/wiki/Job>. Accessed: 2023-05-31.
- [17] Space partitioning. [https://en.wikipedia.org/wiki/Space\\_partitioning](https://en.wikipedia.org/wiki/Space_partitioning). Accessed : 2023 – 05 – 31.