

Programação Concorrente (3º ano de LCC)

**Trabalho Prático**

Relatório de Desenvolvimento

João Diogo Silva  
(a87939)

João Guilherme Reis  
(a84671)

19 de junho de 2022

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Análise e Especificação</b>	<b>3</b>
2.1	Descrição informal do problema . . . . .	3
2.2	Especificação dos Requisitos . . . . .	3
<b>3</b>	<b>Concepção/desenho da Resolução</b>	<b>4</b>
3.1	Servidor . . . . .	4
3.2	Cliente . . . . .	5
3.2.1	Classes . . . . .	5
3.2.2	Avatar . . . . .	5
3.2.3	Menu . . . . .	5
3.2.4	Main . . . . .	5
3.2.5	Interface Gráfica - Processing . . . . .	5
3.3	Estruturas de Dados . . . . .	5
<b>4</b>	<b>Codificação e Testes</b>	<b>6</b>
4.1	Alternativas, Decisões e Problemas de Implementação . . . . .	6
4.2	Testes realizados e Resultados . . . . .	7
<b>5</b>	<b>Conclusão</b>	<b>9</b>

# Capítulo 1

## Introdução

Este trabalho prático foi realizado no âmbito da cadeira de Programação Concorrente no decorrer do ano letivo de 2021/2022.

Foi-nos lançado o desafio de implementar um mini-jogo 2D onde vários utilizadores podem interagir usando uma aplicação cliente com interface gráfica, escrita em Java, intermediados por um servidor escrito em Erlang.

## Capítulo 2

# Análise e Especificação

### 2.1 Descrição informal do problema

Neste trabalho prático foi-nos pedido para implementarmos um mini-jogo 2D com interface gráfica em java através do software processing e de um servidor, escrito em erlang de forma a conseguir jogar em modo multiplayer com comunicações constantes entre o servidor e os clientes através de sockets TCP.

### 2.2 Especificação dos Requisitos

Começamos por um leve levantamento de requisitos:

1. O cliente deverá ter um menu, para que seja possível registar, fazer o login e sair;
2. O servidor em erlang deverá permitir jogar em multiplayer
3. O servidor deve ser capaz de receber as várias coordenadas de cada jogador e enviar em broadcast para todos menos para o mesmo.
4. O servidor deve simular uma partida de 3 a 8 jogadores
5. O servidor deve poder seleccionar um vencedor, e eliminar perdedores
6. O cliente deverá poder mostrar o estado atualizado de todos os jogadores
7. O cliente deve poder enviar a sua informação ao servidor

## Capítulo 3

# Concepção/desenho da Resolução

### 3.1 Servidor

O servidor foi escrito em Erlang, e tem a função de estabelecer uma conexão com os clientes que pretendem iniciar uma partida, prosseguindo a enviar a esses clientes a informação necessária para o funcionamento correto do BattleRoyale. Esta informação consiste em coordenadas dos outros players, cores e tamanho dos seus Avatares, como também qual o seu username.

Como base para o desenvolvimento do jogo, adaptamos um servidor de chat desenvolvido nas aulas práticas, onde a cada nova conexão, é criado um nodo para troca de mensagens. A função `room()` vai escutar a socket TCP desse cliente, e vai ser chamada em paralelo para todas as novas conexões. Se o servidor receber uma mensagem com as informações do estado do player, vai efetuar um pedido de broadcast "seletivo" desta mesma mensagem, já que não queremos enviar a mesma mensagem de volta para o cliente que a enviou, mas sim fazê-la chegar aos outros players.

Utilizamos compreensão de listas para este efeito, onde `Pid` representa o process ID do cliente que enviou a mensagem:

```
1 room(Pids) ->
2   receive
3     stop -> io:format("room OK~n", []);
4     {enter, Pid} ->
5       io:format("user entered ~n", []),
6       room([Pid | Pids]);
7     {line, Data, Pid} ->
8       io:format("received ~p ~n", [Data]),
9       [ UPid ! {line,Data} || UPid <- Pids, UPid /= Pid ],
10    ...
```

Listing 3.1: Função `room()` do Servidor

Seguidamente na função `user()`, ao receber o pedido `{line,Data}` enviamos para os clientes a mensagem pela socket TCP estabelecida.

```
1 ...
2 {line, Data} ->
3   gen_tcp:send(Sock, Data),
4   user(Sock, Room);
5 ...
```

Listing 3.2: Função `user()` do Servidor

## **3.2 Cliente**

### **3.2.1 Classes**

#### **3.2.2 Avatar**

A classe Avatar é a superclass de Cristal e de Player, onde guardamos a informação de cada um sobre o tamanho(que é fixo nos Cristais e que vai aumentando nos Players), a sua cor e as suas coordenadas.

#### **3.2.3 Menu**

A classe Menu é a que comanda o Menu em si, ou seja, através da classe Menu temos o menu inicial, que depois nos leva aos restantes menus, como o Login e o SignUp.

#### **3.2.4 Main**

A classe Main é a que trata do jogo em si, pois é lá que tratamos de desenhar os Players e Cristais, tratamos do caso de derrota e vitória e criamos as threads para que seja possível se efetuada a comunicação entre servidor e cliente.

#### **3.2.5 Interface Gráfica - Processing**

Para desenhar os Cristais e os Players foi utilizada a interface gráfica Processing, em que desenhámos elipses de raio fixo para os cristais e de raio permutável para os players (usando o size em ambos os valores de raio da elipse para esta se tornar uma esfera) e tratamos dos casos em que os cristais desaparecem, ou seja o cristal está dentro do Player, fazendo o mesmo para quando um player desaparece, mas comparando também as cores de cada.

## **3.3 Estruturas de Dados**

Para guardar a informação dos utilizadores que se registaram, criamos um ArrayList em java que vai ser povoado e depois guardado em ficheiro objeto, para que na próxima execução do programa, possa ser lido e trazido para memória. Reconhecemos que esta estratégia não é ideal, mais comentários na secção de Problemas de Implementação.

## Capítulo 4

# Codificação e Testes

### 4.1 Alternativas, Decisões e Problemas de Implementação

Visto que não conseguimos fazer com que os cristais fossem criados pelo servidor e transmitidos para os clientes de forma eficiente, decidimos que seria melhor cada cliente ter os seus cristais enviados em intervalos de tempo fixos, e randomizados pelo processing.

Trata-se de um modelo de renderização do lado do cliente, que não era pretendido na versão ideal do programa. Uma das tentativas de implementação no servidor consistiu em usar a função `send_after()` como também a `apply_interval()`, para enviar mensagens periodicamente para um determinado process ID, desta forma o servidor poderia criar floats randomizados a cada 5 segundos, com ajuda da função:

```
1 ...  
2 random_cristal() -> {rand:uniform()*rand:uniform(600),rand:uniform()*rand:uniform(600)}.  
3 ...
```

Listing 4.1: Função `random_cristal()` do *Servidor*

Infelizmente não conseguimos implementar estas metodologias, e optamos por renderizar os cristais em cada janela do processing, sendo que desta forma eles são independentes dos outros jogadores, e cada jogador tem cristais diferentes.

O processing tem peculiaridades no que toca a gestão de diferentes janelas num só computador. Fechar uma janela 2D (no modo P2D ou no JAVA2D) causa a instância do PApplet a chamar o método `exitActual()`, que contém a linha `System.exit(0)` (um método estático), que termina a Java Virtual Machine que está a ser executada, ou seja, ao fechar uma janela, todas as outras terminam por consequência.

A solução consiste em dar override ao método `exitActual()` na nossa classe PApplet (sem nenhum conteúdo), mas é introduzido outro problema, agora a Java Virtual Machine corre no background mesmo quando é fechada! Logo é importante fechar o sketch com o método `frame.dispose()`. Também há funções do processing que bloqueiam a execução do método `draw()`, pois já que este não pode ser chamado explicitamente, tem que haver forma de o parar, por exemplo para um menu de pausa. Verificamos novamente que ao executar `noLoop()` numa janela, também as outras bloqueavam. Para fazer com que o jogo continuasse após um jogador ser eliminado tivemos que apenas fechar a sua janela.

Um componente importante da aplicação é ser controlada maioritariamente pelo servidor, no que toca à troca de informação, simulação de uma partida, etc, mais conhecido por "Server-Side Rendering". Em algumas ocasiões, optamos por um "Client-Side Rendering", no que toca a guardar o ficheiro objeto com os

detalhes dos utilizadores, e à criação de cristais dinâmicos.

## 4.2 Testes realizados e Resultados

```
Contas: [{id=0, nome='Miguel', password='1234', n_vitorias=0, circulo=x:328.2308 , y: 730.35596}  
 , {id=0, nome='Joao', password='4321', n_vitorias=1, circulo=x:328.2308 , y: 730.35596}  
 ]  
  
Utilizador Atual: Anónimo  
Agar.IO  
  
1: Jogar  
2: Log in  
3: Registo  
4: Leaderbord  
5: Sair  
2  
  
Nome  
Joao  
  
Password  
4321  
Login Efetuado.  
Utilizador Atual: {id=0, nome='Joao', password='4321', n_vitorias=1, circulo=x:328.2308 , y: 730.35596}
```

Figura 4.1: Menu da aplicação

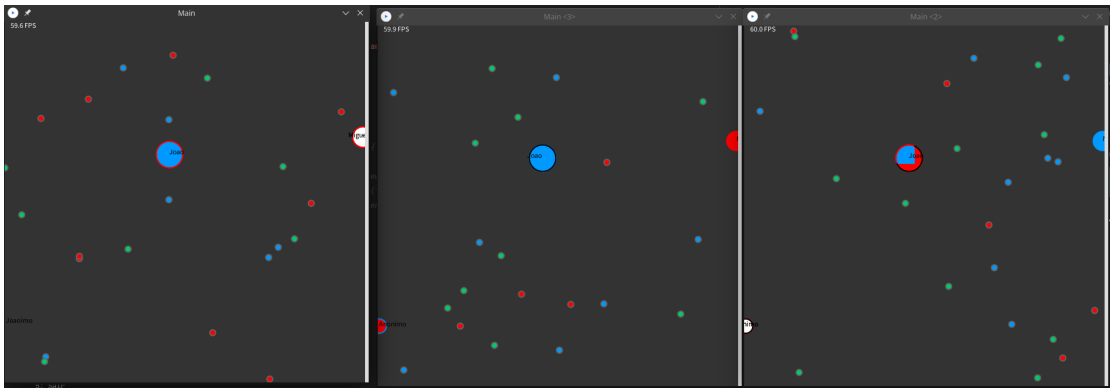
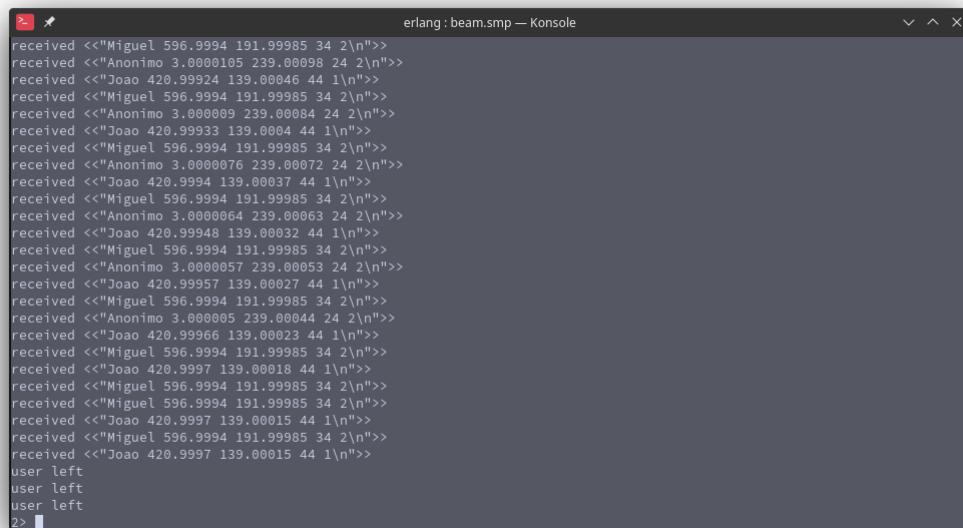


Figura 4.2: Jogo com 3 jogadores





```
erlang : beam.smp - Konsole
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Anonimo 3.0000105 239.00098 24 2\n">>
received <<"Joao 420.99924 139.00046 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Anonimo 3.000009 239.00084 24 2\n">>
received <<"Joao 420.99933 139.0004 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Anonimo 3.0000076 239.00072 24 2\n">>
received <<"Joao 420.9994 139.00037 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Anonimo 3.0000064 239.00063 24 2\n">>
received <<"Joao 420.99948 139.00032 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Anonimo 3.0000057 239.00053 24 2\n">>
received <<"Joao 420.99957 139.00027 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Anonimo 3.000005 239.00044 24 2\n">>
received <<"Joao 420.99966 139.00023 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Joao 420.9997 139.00018 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Joao 420.9997 139.00015 44 1\n">>
received <<"Miguel 596.9994 191.99985 34 2\n">>
received <<"Joao 420.9997 139.00015 44 1\n">>
user left
user left
user left
2>
```

Figura 4.3: Estado do servidor

## Capítulo 5

# Conclusão

Estamos cientes de que era pedido uma renderização do lado do servidor, porém em algumas ocasiões usamos "Client-Side Rendering", devido a alguns problemas na resolução da aplicação, sabendo que este não seria o método mais aconselhado. Tendo isto em conta, pensamos que as funcionalidades de concorrência e paralelismo foram alcançadas, podendo também ser alvo de algumas melhorias.

Foi um trabalho interessante que nos ajudou a desenvolver aptitudes de programação concorrente, como serviu também para uma das primeiras experiências no desenvolvimento de videojogos.