

## Programação Concorrente

Trabalho Prático – Grupo 3

*Battle Royale*

- João Nogueira – A87973
- Miguel Gonçalves – A90416
- Paulo Costa – A87986
- Rui Baptista – A87989

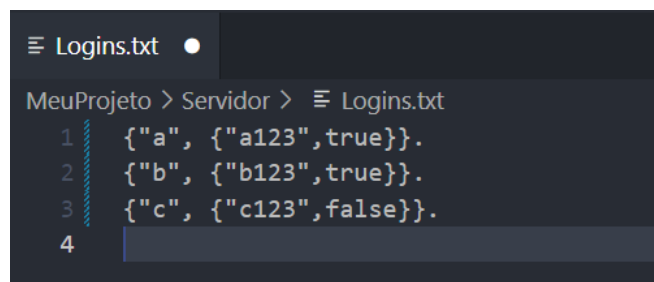
## Servidor

O servidor encontra-se dividido em 7 classes auxiliar, conversores, login\_manager, server, cristais, jogadores e estado.

**Auxiliares:** A classe auxiliar é composta por um conjunto de operações auxiliares tais como `multiplicaVector/2` que multiplica um vetor por número, `normalizaVector/1`, `adicionaPares/2` que adiciona dois vetores, `distancia/2` que indica a distância entre dois vetores, `posiciona/2` que indica uma posição sem nenhum obstáculo para o raio dado e `geraObstaculo/2` que adiciona um certo número de obstáculos a uma lista de obstáculos.

**Conversores:** A classe conversores tem as funções `formatarPontuacoes/1` que formata as pontuações para depois enviar para os clientes, `formataTecla/1` que formata a tecla enviada pelos clientes para indicar quais operações devem ser realizadas e a função `formatState/1`. Esta é responsável transformar o estado para enviar aos clientes para eles utilizarem essa informação para apresentar o jogo, e as suas funções auxiliares tais como, `jogadores_para_string/1`, `cristal_para_string/1` e `obstaculos_para_string/1`.

**Login\_manager:** A classe login\_manager é a classe responsável por criar uma conta, fechar uma conta e efetuar o login e logout de um jogador. Esta classe contém as seguintes funções `start_Login_Manager/0` responsável por iniciar o login\_manager, `create_account/2`, `close_account/2`, `login/2`, `logout/1`. Contendo este um loop que guarda as informações das contas tais como o nome, palavra-passe e se esta encontra-se online, esta recebe os comandos das outras funções e atualiza o Map onde estas informações são guardadas com eles.



```
Logins.txt
MeuProjeto > Servidor > Logins.txt
1 {"a", {"a123",true}}.
2 {"b", {"b123",true}}.
3 {"c", {"c123",false}}.
4
```

Figura 1- logins

Por exemplo, na figura acima apresentada o user “a” de password “a123” encontra-se com login efetuado, o user “b” segue a mesma medida enquanto o user “c” não se encontra logado (pode-se verificar com a variável false).

Este ficheiro .txt é gerado no login\_manager e é atualizado sempre que é feito um registo.

**Server:** A classe server é responsável por iniciar o servidor fazer conexão entre o cliente e o servidor indicando qual é a operação que o cliente pediu. A função start/0 é responsável por iniciar o servidor iniciando um processo com estado e outro com o login\_manager, criar um socket e de o enviar para a função acceptor/1, esta aceita uma requisição de conexão criando outro processo acceptor para podermos ter mais de um cliente e inicia o authenticator/1 com essa conexão.

O authenticator/1 é responsável por ler uma mensagem do socket e vai tentar efetuar a operação descrita caso ela seja de login, create\_account ou close\_account. Caso seja um login bem-sucedido vai para a função user/2, esta espera o utilizador ter uma vaga para entrar no jogo quando consegue chama o cicloJogo/3 este recebe e lida com as mensagens vinda do utilizador.

```
novaSala() ->
  [{spawn(fun() -> estado([],[]) end),[]}].

%Cria 4 salas com Pids diferentes
criaSala() ->
  novaSala() ++ novaSala() ++ novaSala() ++ novaSala().

lounge(Salas) ->
  receive
  {ready,Username, UserProcess} ->
    Sala = verificaSala(Salas),
    {Pid,ListaJogadores} = Sala,
    NovasSalas = remove(Sala,Salas),
    Pid ! {ready,Username,UserProcess},

    NovoJogadores = ListaJogadores ++ [{Username,UserProcess}],
    NovoSala = {Pid,NovoJogadores},

    lounge([NovoSala | NovasSalas]);

  {atualizaPontos, Username, UserProcess} ->
    onEstaJogador(Salas,UserProcess) ! {atualizaPontos,Username,UserProcess},
    lounge(Salas);
  {leave, Username, UserProcess} ->
    onEstaJogador(Salas,UserProcess) ! {leave,Username,UserProcess},
    lounge(Salas)
  end.
```

Figura 2- salas

Sempre que o servidor inicia, faz a criação das 4 salas que estarão disponíveis a receber novos jogadores (clientes), em que cada sala tem um PID diferente, isto para ser possível a identificação das mesmas e nas quais iniciamos um estado diferente em cada uma, de modo que estas sejam independentes entre si.

**Cristais:** A classe cristais alberga as funções relativas aos cristais tais como novoCristal/2 que gera um novo cristal e ainda verifica ColisoesCristalLista/2 que com o auxílio da verificaColisaoCristal/2 verifica a colisão entre um jogador e uma lista de cristais.

**Jogadores:** Cada objeto jogador contém os seus essenciais sendo alguns deles constantes: raio máximo, raio mínimo, arrasto, aceleração linear e aceleração angular. Também contém variáveis como cor atual, velocidade, raio, direção, agilidade, pontuação e posição, estes são criados com a função novoJogador/1.

Tal como era pedido, quando um jogador colide com um cristal fica com a cor desse cristal e ganha massa, para tal criamos as funções atualizaColisaoVerdes, atualizaColisaoVermelhos e atualizaColisaoAzuis que são usadas na função atualizaJogadores.

De forma a lidar com a colisão entre jogadores, desenvolvemos a função verificaColisaoJogadores2 que auxilia verificaColisaoJogadoresL a verificar se os jogadores colidem uns com os outros, se acontecer aplica a regras das cores, após isso caso a perda de massa seja menor do que a massa mínima o jogador é eliminado. O jogador que “vencer” a colisão ganha massa e será retirada massa ao jogador perdedor.

Em relação ao movimento do jogador, o erlang recebe do java a posição do rato e calculamos o vetor da direção da posição do rato e também o vetor da direção do “olhar” do jogador. Usando a função vaiParaCoordenadas basicamente substituímos o vetor do rato pelo vetor do jogador fazendo assim com que o jogador se movimente pelo mapa de jogo.

**Estado:** Esta classe é a responsável pelo funcionamento interno do jogo. A função start\_state/0 inicia os processos e o refresh/1 envia de tanto em tanto tempo uma mensagem ao gameManager/2 com o comando refresh para este atualizar o estado e enviá-lo aos jogadores. As funções adicionarVerdes/1, adicionarReds/1 e adicionarBlues/1 enviam de tanto em tanto tempo uma mensagem ao gameManager/2 para este ver se é possível adicionar cristais com as respetivas cores de cada um.

Nesta classe também foi implementado o requisito de suportar 4 partidas em simultâneo, no momento inicial criamos as 4 salas com um limite mínimo de 3 jogadores e máximo de 8. A inserção de jogadores é feita por ordem das salas, verificando primeiro se há slots livres na sala 1, depois a sala 2,3 e 4.

Temos também as funções novoEstado/0 que cria um novo estado para iniciar um jogo, adicionaJogador/2 que adiciona um jogador ao estado, removeJogador/2 que remove um jogador do estado e atualizaMelhoresPontos/2 que atualiza as melhores pontuações dos jogadores auxiliam o gameManager/2.

A função update atualiza o estado para isso, começa por verificar as colisões entre os jogadores e os cristais com verificaColisoesCristalLista/2, de seguida retira os cristais que colidiram com jogadores e atualiza os restantes na função atualizaListaCristal/2.

De seguida atualizamos os jogadores na função atualizaJogadores/4 para concluir filtramos os jogadores que perderam na função filtrar/2 que devolve a Lista dos jogadores que ainda continuam e os dados dos jogadores que perderam para indicar que já perderam. Aqui também temos a função updateBoost que é responsável pelo boost, o cliente pode clicar no botão esquerdo do rato para ativar o boost que fica ativo até o cliente deixar de clicar no botão ou até chegar à massa mínima.

# Cliente

De forma a desenvolvermos a nossa interface gráfica em Java utilizamos tal como sugerido o Processing, e para a criação de botões e janelas usamos mais concretamente o g4p que é uma livreria de Processing que possui um builder onde podemos criar e personalizar a nossa janela e o código é nos gerado automaticamente o que facilitou o processo da construção da interface gráfica.

O nosso menu disponibiliza 5 opções:

- **Registar:** escolher nome e palavra-passe pretendida, nenhum dos campos pode ser deixado em branco nem o nome da conta pode já estar a ser utilizado.
- **Cancelar:** fecha o nosso menu.
- **Opções:** onde podemos mudar a porta e o endereço IP a onde nos vamos conectar.
- **Login:** digitamos o nome e a palavra-passe correspondente.
- **Nova Partida:** se possível entra numa nova partida.

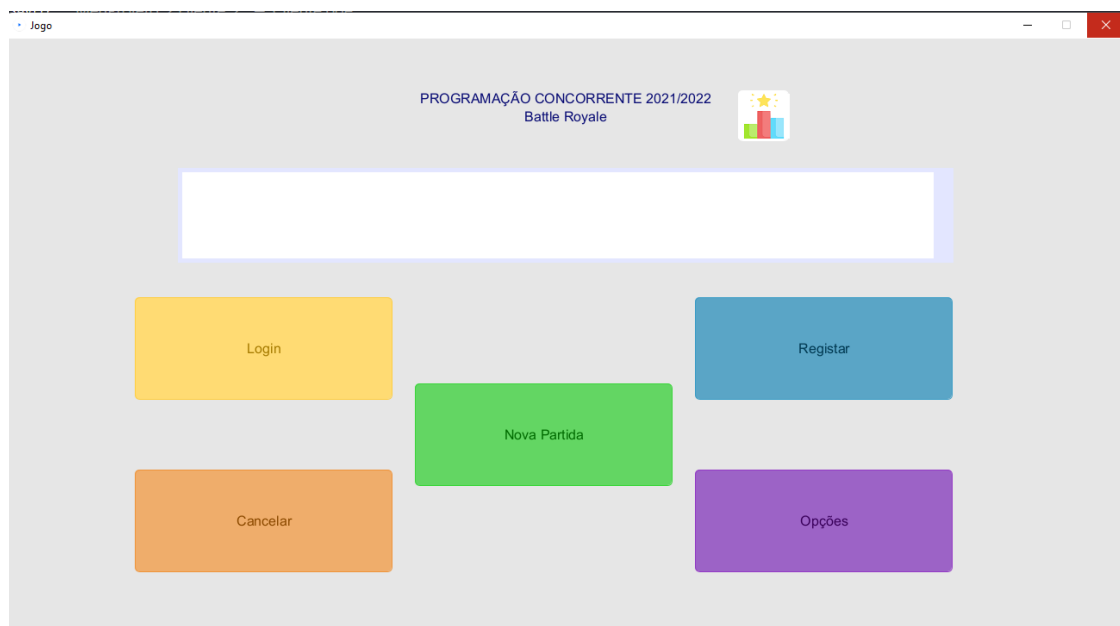


Figura 3- menu

Cada botão do menu é mediado pelo seu handler, onde comunicámos com a socket as nossas ações.

Para a parte de desenhar a tela, temos duas **threads**, uma que lê e atualiza o estado corrente do jogo e a outra thread atualiza as pontuações globais a cada 5 segundos sendo o ciclo de vida desta última desde quando se abre a janela pontuações globais até ao fecho dessa mesma janela.

## Classes Java:

**Conector:** classe que nos permite conectar com o servidor via Sockets TCP, nesta alocamos a Socket, o BufferedReader e o PrintWriter. Temos 4 métodos nesta classe, o connect que recebe um endereço IP e uma porta e cria uma Socket, o nosso BufferedReader será então criado a partir da inputStream da socket, e o PrintWriter criado a partir da outputStream da socket.

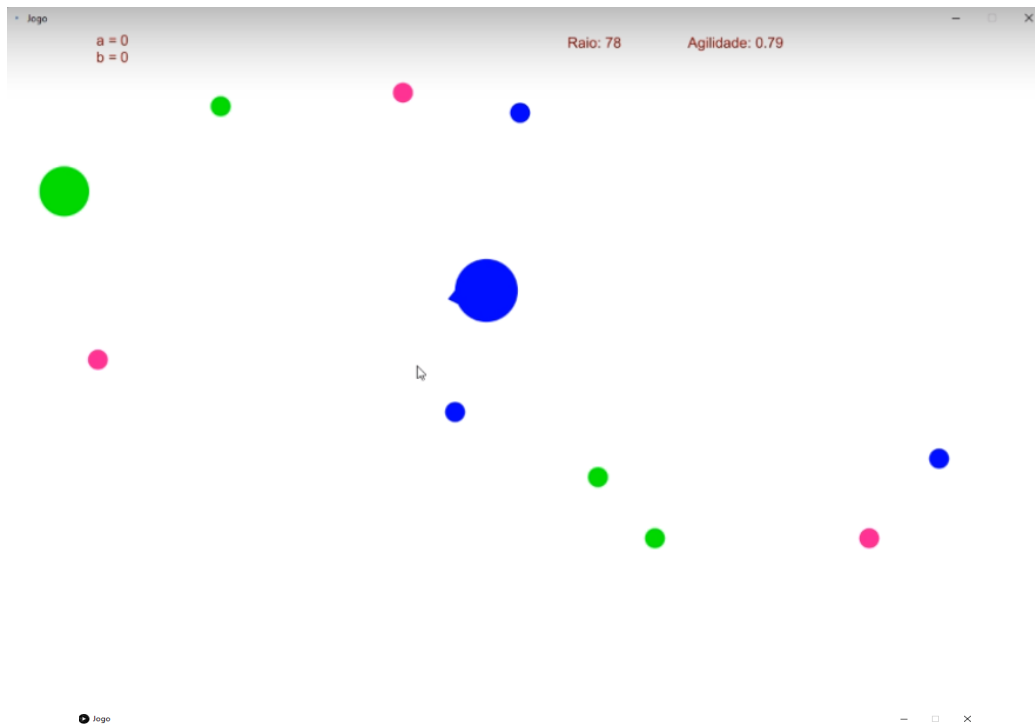
O método read vai ler a informação da stream a partir do nosso BufferedReader, o método write escreve para a stream a partir do nosso PrintWriter, o método disconnect fecha a nossa socket.

**Cristal:** classe onde alocamos informação do cristal que apenas interessa para a apresentação na tela de jogo, como a posição, tamanho e cor. Temos um método draw onde dependendo do tipo do cristal a sua cor vai variar, ou seja, tipo = 0 é um cristal verde, tipo 1 é vermelho e tipo 2 é azul.

**Jogador:** classe que aloca o nome, a pontuação do jogador, a sua posição, direção, raio, agilidade e o nome do jogador do cliente que nos ajuda a saber se este jogador é o jogador do cliente ou é um adversário temos um método draw onde desenhamos o jogador do cliente que dado o respetivo tipo atribuímos a sua cor, tal como fizemos para os cristais. O jogador vai ter um pequeno triângulo em cima para o cliente saber qual dos jogadores está a controlar.

**Jogo:** classe onde alocamos toda a informação que queremos que seja representada na tela, desde um array com os cristais e outro array com os jogadores. Temos também um HashMap para as pontuações atuais do jogo. Esta classe tem um método de update onde utilizamos um **ReentrantLock** garantindo que temos **exclusão mútua**. Temos também um método draw onde apenas percorremos os arrays e chamamos as funções draw das outras classes.

## Resultados:



PERDEU

VENCEU

## Conclusão:

Achamos este projeto de grande importância visto que abordou praticamente todos os conceitos lecionados ao longo deste semestre, onde tivemos de aplicá-los num desafio de maior escala. Conseguimos concluir todas as funcionalidades, deixando-nos assim com um sentimento de dever cumprido. Desta forma estamos contentes com o resultado final, no entanto temos noção que existe sempre espaço para melhorar.

Já fora deste contexto, gostaríamos de agradecer ao professor Paulo Almeida pelo tempo disponibilizado.