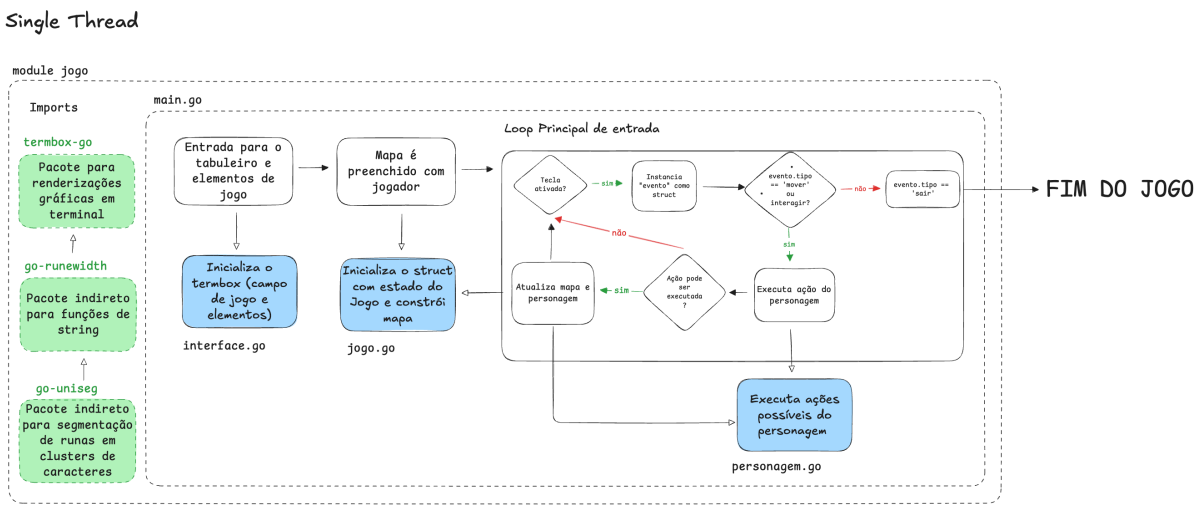
Relatório

T1 - Desenvolver elementos autônomos concorrentes para um jogo de terminal

Arthur Pereira Testa

O Jogo (Single Thread)

O jogo inicialmente funcionava com um loop Single Thread simples, onde o personagem é o unico elemento com interatividade do jogo. O diagrama abaixo explica de maneira simplificada como funciona o jogo e seus modulos principais.



Decisão de Arquitetura (Multi Thread)

Uma vez definida a arquitetura centralizando concorrência de ações de jogo em um coordenador, foram incluídos no código a interface *Cmd* e algumas structs (`[tipos.go]`), que serão utilizadas para definir o tipo de ação tomada (a vantagem de utilizar structs sendo que podemos facilmente iterar sobre o tipo da ação para chamar as funções necessárias em um *switch case* na instância do Coordenador), bem como declaramos os novos elementos e seus símbolos.

```
    Personagem    = Elemento{'☺', CorCinzaEscuro, CorPadrao, true}
    Inimigo       = Elemento{'☠', CorVermelho, CorPadrao, true}
    Parede        = Elemento{'▤', CorParede, CorFundoParede, true}
    Vegetacao     = Elemento{'♣', CorVerde, CorPadrao, false}
    Vazio         = Elemento{' ', CorPadrao, CorPadrao, false}
    PortalFechado = Elemento{'◉', CorVerde, CorPadrao, true}  // bloqueia
    PortalAberto  = Elemento{'◎', CorVerde, CorPadrao, false} // não bloquei
a
    AlavancaOff   = Elemento{'┴', CorCinzaEscuro, CorPadrao, true}
    AlavancaOn    = Elemento{'┬', CorVerde, CorPadrao, true}
    ArmadilhaOff  = Elemento{'^', CorCinzaEscuro, CorPadrao, false}
    ArmadilhaOn   = Elemento{'▲', CorVermelho, CorPadrao, false}
    SentinelaElem = Elemento{'§', CorVermelho, CorPadrao, true}
```

Os novos elementos e personagens podem ser descritos como:

Nome	Descrição	Observação
Portal	Quando ativado, transporta o jogador para uma posição determinada automaticamente, e depois fecha após 5 segundos	*Comunicação com timeout* - o canal recebe uma chamada de função para fechar o portal, mudando seu status e símbolo.
Alavanca	Ativa o portal, possibilitando teletransporte do jogador	*Alterna entre estados (ligado/desligado)*
Armadilha	Ativada em períodos diferentes, repentinamente. Se o jogador passa sobre enquanto ativada, pode receber dano	*Causa efeitos sobre o personagem. Alterações simultâneas no mapa. Modificação de estado do jogo.*
Sentinela	As Sentinelas transitam o mapa, patrulhando em um raio predefinido até que estejam em proximidade do jogador - ponto em que o *modoPerseguir* é ativado. A partir daí, as Sentinelas perseguem o jogador até	*Comunicação entre elementos por canais* - o canal responsável pela sentinela recebe a informação da posição atual do jogador e, caso esteja dentro do parametro definido como raio de

	que ele esteja fora de um raio pré-definido (*raioPerseguir) - calculado por distancia de Manhattan.*	perseguição, ativa o modo de perseguição.

```
        case p := ←chPosPlayer:
            ultimoVisto = p
            if distManhattan(pos, p) <= raioPerseguir {
                modoPerseguir = true
            } else if modoPerseguir && distManhattan(pos, p) > (raioPerseguir
+2) {
                modoPerseguir = false
            }
```

Criamos também algumas funções auxiliares que o coordenador pudesse usar para facilitar a implementação dos padrões de movimento dos demais elementos e NPCs (non playable characters).

Nome	Descrição	Observação
`jogoDentro` : bool	Função auxiliar que verifica se o ponto desejado para movimentação no mapa é válido (dentro dos limites de mapa).	Função booleana retorna “True” caso o ponto esteja dentro do grid do mapa, e “False” caso contrário. Não verifica se celula é bloqueante ou não.
`jogoCelula` : Elemento	Verifica o conteúdo de determinada celula (se celula é válida, ou seja, `jogoDentro() == True`), retornando o objeto do tipo Elemento (previamente definido).	Tipo Elemento possui atributos: `{ simbolo rune cor Cor corFundo Cor tangivel bool // Indica se o elemento bloqueia passagem }`
`jogoSetCelula` : void	Atribui determinado elemento à posição do mapa, desde que a celula seja válida (`jogoDentro() == True`)	

Assim, construímos a fundação que será utilizada na implementação do Coordenador e dos canais que irão transmitir as informações do jogo e dos seus elementos de forma concorrente. As funções auxiliares, apesar de não necessariamente terem uma atuação direta na implementação de concorrência, ajudam a reduzir o código e facilitar a sua leitura, especialmente considerando a natureza “multi-tarefa” do coordenador.

Coordenador.go

O coordenador é o componente central da arquitetura concorrente do jogo, responsável por gerenciar a comunicação entre elementos do jogo e garantir a correta sincronização das ações. Abaixo detalhamos sua implementação e os mecanismos de concorrência utilizados.

Estrutura do Coordenador

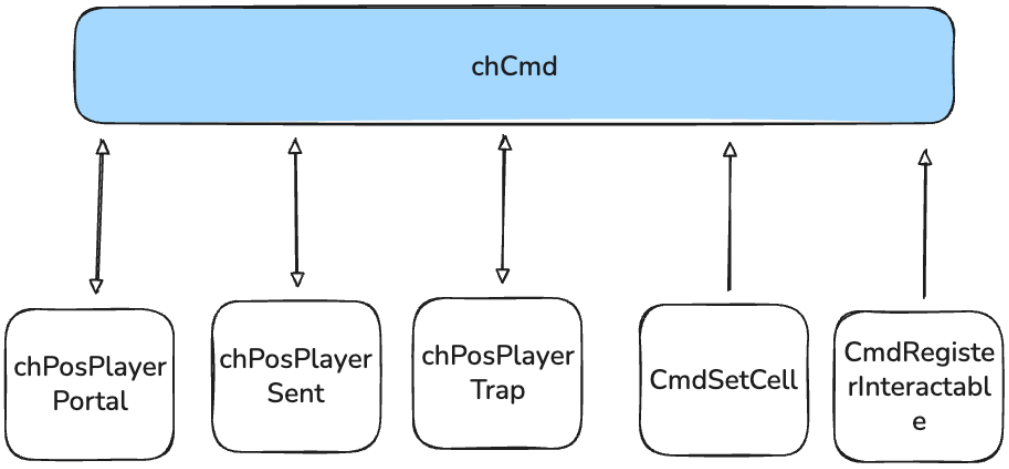
A struct `coordenador` possui os seguintes campos:

Nome	Tipo	Descrição
`jogo`	*Jogo	Ponteiro para a instância principal do jogo, contendo o estado atual
`chTeclado`	chan termbox.Event	Canal de leitura para eventos de teclado
`chCmd`	chan Cmd	Canal de leitura para comandos enviados por outros componentes
`subsPos`	[]chan Ponto	Slice de canais para publicar a posição do jogador (padrão Observer)
`interactables`	map[[2]int]chan bool	Mapa de elementos interativos, usando coordenadas [x,y] como chave e canais como valor

O Coordenador.go uma goroutine “chefe” que é a **única** a mexer no estado do jogo (mapa, posição do jogador, desenho).

- **Como funciona:** recebe **comandos** por um canal “barramento” (`chCmd`) e usa `select` para:
  - mover o jogador (`CmdMovePlayer`),
  - acionar interações perto do player (`CmdInteragir`),
  - mudar células do mapa (`CmdSetCell`),

    - teleportar (`CmdTeleportPlayer`),
    - registrar inscritos para receber a posição do player (`CmdSubscribePlayerPos`),
    - encerrar (`CmdQuit`).
- **Por que existe:** garante **exclusão mútua via canais** (sem `mutex`). Só o coordenador altera o estado; os demais apenas **pedem** mudanças.
- **Extras:** publica a posição do jogador para quem precisar e redesenha periodicamente (ticker).



Mecanismos de Concorrência

1. Canais de Comunicação

O sistema implementa **três canais principais** para gerenciar os eventos de jogo:

- **Canal de eventos de teclado (`chTeclado`):** Recebe eventos de teclado capturados em uma goroutine separada
- **Canal de comandos (`chCmd`):** Recebe comandos de outros componentes do jogo (NPCs, elementos, etc.)
- **Canais de publicação (`subsPos`):** Distribui a posição do jogador para elementos que precisam reagir a mudanças
- **Canais de interação (`interactables`):** Permite que elementos interativos sejam notificados quando o jogador interage com eles

2. Comunicação Não-bloqueante

O código implementa comunicação não-bloqueante em vários pontos usando a declaração `select` com cláusula `default`. Por exemplo:

```
// Em publicarPosPlayer
for _, ch := range c.subsPos {
    select { // não bloquear o coordenador
    case ch ← p:
    default:
    }
}
```

Esta abordagem garante que o coordenador nunca fique bloqueado ao tentar enviar uma mensagem para um canal que pode estar cheio ou sem leitores.

3. Temporizadores (Tickers)

O sistema utiliza um ticker para atualizações periódicas da renderização:

```
tickerRender := time.NewTicker(120 * time.Millisecond) // refresh suave
defer tickerRender.Stop()
```

Este mecanismo garante que a interface gráfica seja atualizada em intervalos regulares, independentemente de outros eventos.

4. Exclusão Mútua Lógica

O padrão arquitetural implementado oferece uma exclusão mútua lógica, já que o coordenador é o único componente que modifica diretamente o estado do jogo. Não é necessário *mutex* pois a exclusão é aplicada pelos próprios canais que chamam o coordenador. Outros componentes devem enviar comandos para o coordenador, que os processa de forma serializada:

```
// Exclusão mútua: apenas o coordenador altera o mapa
if jogoDentro(c.jogo, m.X, m.Y) {
    jogoSetCelula(c.jogo, m.X, m.Y, m.Elem)
}
```

Loop Principal e Multiplexação

O método `loop()` do coordenador implementa uma multiplexação de eventos usando `select`, permitindo que ele responda a múltiplas fontes de eventos concorrentemente:

```
for {
    select {
    case ev := ←c.chTeclado:
        // Tratamento de eventos de teclado
    case cmd := ←c.chCmd:
        // Tratamento de comandos
    case ←tickerRender.C:
        // Redesenho periódico
    }
}
```

Padrões de Concorrência Implementados

O código implementa diversos padrões de concorrência:

- **Observer:** Através de `subsPos`, elementos podem "assinar" para receber atualizações da posição do jogador
- **Command:** O canal `chCmd` recebe diferentes tipos de comandos (polimórficos através de interface) que são processados centralmente
- **Actor Model:** Cada elemento interativo funciona como um ator independente, comunicando-se por troca de mensagens
- **Multiplexador:** O coordenador funciona como um multiplexador de eventos, centralizando o processamento

Elementos.go

O arquivo "elementos.go" implementa diversos componentes interativos do jogo que operam de forma concorrente através de goroutines independentes, comunicando-se com o coordenador via canais. Cada componente utiliza técnicas específicas de concorrência em Go para criar comportamentos distintos:

1. Alavanca

A função `iniciarAlavanca` cria um mecanismo interativo que pode ser acionado pelo jogador:

- **Mecanismo principal:** Registra um canal de interação com o coordenador e executa uma goroutine dedicada

```
        if ligada {
            chCmd ← CmdSetCell{X: x, Y: y, Elem: AlavancaOn}
```

- **Comunicação não-bloqueante:** Usa `select` com `default` ao enviar sinal de abertura (evita bloqueios)

```
            select {
            case outAbrir ← sinal{}:
```

- **Padrão Actor:** Goroutine mantém estado interno (ligada/desligada) e responde a mensagens recebidas

2. Portal com Timeout

A função `iniciarPortal` implementa um portal que se abre temporariamente e teleporta o jogador:

- **Multiplexação avançada:** Escuta concorrentemente três canais distintos (abertura, posição do jogador, timeout)
- **Temporizador dinâmico:** Implementa lógica sofisticada com `time.Timer` para controlar o estado do portal
- **Prevenção de vazamento:** Usa técnicas de drenagem de canal para evitar "fogo fantasma" do timer
- **Manipulação de timeouts:** Funções `resetTimer` e `stopTimer` gerenciam o ciclo de vida do temporizador

3. Sentinela

A função `iniciarSentinela` cria um inimigo que patrulha entre waypoints e persegue o jogador quando próximo:

- **Temporizador periódico:** Usa `time.Ticker` para movimento regular independente da entrada do jogador
- **Máquina de estados:** Alterna entre patrulha e perseguição baseado na distância do jogador
- **Select multiplexado:** Escuta concorrentemente entre o ticker de movimento e atualizações da posição do jogador

4. Armadilha

A função `iniciarArmadilha` implementa um elemento que alterna entre estados ativo/inativo e causa dano ao jogador:

- **Múltiplos temporizadores:** Usa dois `time.Ticker` separados para controlar os períodos de ativação e desativação
- **Multiplexação tripla:** Escuta concorrentemente três canais (ticker de ativação, ticker de desativação, posição do jogador)
- **Detecção de colisão:** Verifica se o jogador está na posição da armadilha enquanto ativa

Funções Auxiliares

O código também implementa funções utilitárias para cálculos de posicionamento:

- **`passoRumo`:** Determina o próximo passo para movimentação em direção a um alvo
- **`distManhattan`:** Calcula a distância de Manhattan entre dois pontos (soma de diferenças absolutas)
- **`abs`:** Função auxiliar para valor absoluto

Este design demonstra o uso eficiente de goroutines e canais para criar um sistema concorrente onde múltiplos elementos do jogo operam independentemente, mas de forma coordenada, comunicando-se através de mensagens assíncronas com o coordenador central.

Main.go

- **Inicializa** a interface, carrega o mapa e cria os **canais**.
    - Aqui não há mudança com relação à versão anterior do jogo
- **Sobe** o **coordenador** (goroutine) e a goroutine do **teclado** (transforma teclas em **comandos**).

```
go coord.loop()

go capturarTeclado(chCmd)
```

- **Assina** canais para receber a posição do jogador (elementos usam isso).

```
chPosPlayerPortal := make(chan Ponto, 4)
chPosPlayerSent := make(chan Ponto, 4)
chPosPlayerTrap := make(chan Ponto, 4)
chCmd ← CmdSubscribePlayerPos{Ch: chPosPlayerPortal}
chCmd ← CmdSubscribePlayerPos{Ch: chPosPlayerSent}
chCmd ← CmdSubscribePlayerPos{Ch: chPosPlayerTrap}
```

- **Cria** os elementos (alavanca, portal, sentinela, armadilha).

```
alavancaPos := Ponto{X: jogo.PosX, Y: jogo.PosY + 2}
portalPos := Ponto{X: jogo.PosX, Y: jogo.PosY + 4}
destino := Ponto{X: jogo.PosX + 15, Y: jogo.PosY + 10}

chAbrirPortal := make(chan sinal, 1)
iniciarAlavanca(alavancaPos.X, alavancaPos.Y, chCmd, chAbrirPortal)
iniciarPortal(portalPos.X, portalPos.Y, destino, chCmd, chAbrirPortal, ch
PosPlayerPortal)
```

- **Espera** o jogo terminar (recebe sinal de **done** quando chega `CmdQuit`).

    ←done

Loop Principal

Uma das principais mudanças da versão “Single Threaded” e a versão “Multi Threaded” é que o loop principal de jogo passa a ser movido para o modulo

Coordenador.go, e não mais no modulo Main.go, que fica responsável apenas por “subir” as go routines do coordenador e de captura do teclado.