Desenvolver Elementos Autônomos Concorrentes para um Jogo

Amanda Wilmsen[*], Killian Domingues[*], Luís Trein[*] e Maria Rita Rodrigues[*]

Escola Politécnica — PUCRS

24 de abril de 2025

Objetivo

Este trabalho tem como objetivo a implementação de três novos elementos autônomos e concorrentes para um jogo de terminal baseado em interface textual. Essa experiência serviu como aplicação prática de conceitos de paralelismo e concorrência no contexto de jogos, e está disponível no repositório Trabalho 1.

1. Descrição do Jogo

O jogador deve explorar um mapa em **busca de quatro tesouros escondidos em caixas misteriosas**. Cada caixa pode conter um tesouro, uma armadilha ou estar vazia. Visualmente, todas as caixas são idênticas no início (amarelas e inofensivas), e apenas durante a interação seu conteúdo é revelado: **verde para tesouro**, **vermelho para armadilha** e **cinza para caixa vazia**.

A interação só é possível quando o jogador está a uma célula de distância da caixa, e o jogador não pode atravessá-las — elas são obstáculos físicos no mapa. A cada 20 segundos, todas as caixas mudam de posição aleatoriamente, e o número de caixas em movimento diminui dinamicamente conforme o jogador coleta as caixas.

Além disso, um NPC Guia auxilia o jogador fornecendo dicas baseadas na proximidade de objetos:

- **Quente:** indica um tesouro próximo;
- **Frio:** sugere armadilha por perto;
- **Morno:** aponta presença de algum objeto desconhecido.

Se os quatro tesouros não forem encontrados em 1 minuto, um **monstrinho ladrão** será liberado no cenário. Este inimigo se movimenta de forma autônoma e, ao se aproximar dos tesouros

[1] amanda.wilmsen@edu.pucrs.br
[2] killian.d@edu.pucrs.br
[3] luis.trein@edu.pucrs.br
[4] maria.rita04@edu.pucrs.br

— tanto os ainda escondidos quanto aqueles já coletados pelo jogador —, realiza sua captura. A partida é encerrada em derrota caso todos os tesouros sejam perdidos.

2. Representação Visual

Jogador	Caixa Misteriosa	Tesouro	Armadilha	Vazia	NPC	Monstro
☺	■	■	■	■		¥

3. Requisitos Mínimos de Concorrência e Sincronização

3.1. Novos elementos concorrentes (mínimo 3 tipos)

Foram implementados os seguintes elementos concorrentes, cada um com comportamento próprio:

- **Tesouros:** ao serem coletados, somem do mapa e incrementam a contagem;
- **Armadilhas:** eliminam o jogador instantaneamente;
- **Caixas vazias:** não alteram o estado do jogo;
- **NPC Guia:** segue o jogador e fornece dicas em tempo real;
- **Monstrinho ladrão:** surge após 1 minuto e tenta roubar os tesouros restantes.

Todos os elementos acima operam em goroutines independentes, permitindo que seus comportamentos ocorram mesmo sem a interação direta do jogador. As caixas são inicializadas em **caixa.go**, onde também está a função **Iniciar()** que define sua lógica concorrente – e o mesmo se aplica ao NPC e ao monstrinho.

3.2. Exclusão mútua

Para garantir a integridade da matriz do mapa e evitar condições de corrida, utilizamos o **sync.Mutex** para proteger seções críticas. A exclusão mútua é aplicada durante:

- Movimentação de elementos autônomos (caixa.go, monstro.go, npc.go);
- Atualizações da interface com interfaceDesenharJogo() (interface.go);
- Interações com as caixas.

- **Exemplo do código:**

```
c.Mutex.Lock()
(*c.Mapa)[c.Y][c.X] = Vazio
c.Mutex.Unlock()
```

No arquivo **jogo.go**, a struct principal Jogo possui o **MutexMapa** para proteger a matriz Mapa.

3.3. Comunicação entre elementos por canais

As interações entre o jogador, NPC e monstrinho foram realizadas via canais. Essa abordagem garante a troca segura de informações sem que elementos concorrentes precisem acessar diretamente estruturas compartilhadas, prevenindo comportamentos inesperados.

Exemplo: o campo **Interacao chan bool** na struct **Caixa (em caixa.go)** permite que o jogador envie:

```
caixa.Interacao ← true
```

A caixa então recebe e executa **efeito()**, o que nos garante o desacoplamento e segurança concorrente.

3.4. Escuta concorrente de múltiplos canais

O NPC Guia escuta simultaneamente canais relacionados ao movimento do jogador e à posição de elementos no cenário, reagindo dinamicamente para fornecer dicas. O monstrinho ladrão também monitora múltiplos canais, alternando entre comportamento livre e perseguição direcionada aos tesouros, conforme os sinais recebidos. As caixas são outro exemplo, pois elas utilizam o **select** para reagir a interação ou **timeout**:

```
select {
case ← time.After(20 * time.Second):
 c.mover()
case ← c.Interacao:
 c.efeito()
}
```

Esse padrão nos garante a responsividade no comportamento dos elementos.

3.5. Comunicação com timeout

A movimentação periódica das caixas e o surgimento do monstrinho são controlados com timeout usando **time.After**. Caso o tempo limite seja atingido sem eventos externos, as caixas se movem automaticamente e o monstrinho é ativado, o que mantém a dinâmica do jogo constante.

Exemplo no jogo.go:

```
jogo.MonstroSpawn = time.Now().Add(30 * time.Second)
if time.Now().After(jogo.MonstroSpawn) {
jogo.Monstro = monstroNovo()
jogo.Monstro.Iniciar(jogo)
}
```