Desenvolvimento de uma Versão do Jogo "Pac-Man" em Sistemas Operacionais

Allan Carlos Figueiredo Echeverria e Vicente Knihs Erbs

1 Introduction

Sistemas operacionais representam os “programas mais fundamentais para um sistema, cuja responsabilidade é controlar os recursos dos computadores e fornecer uma base sobre a qual aplicações podem ser escritas” [1]. Durante o período de um semestre, na disciplina de Sistemas Operacionais do curso de Engenharia Mecatrônica da UFSC - Campus Joinville, foi desenvolvida uma parcela de um sistema operacional em nível de usuário sobre o sistema Linux.

A parte implementada consistiu em abordar o gerenciamento de *threads*, incluindo preempção baseada ou não no tempo. Também, foram implementados o uso de semáforos: mecanismos de coordenação para garantir a exclusão mútua entre tarefas [2].

Ao final, utilizando o sistema de gerência de *threads* desenvolvido e a biblioteca gráfica *SFML* [3], foi implementada uma versão do célebre jogo *Pac-Man*.

O presente documento relata como a aplicação foi estruturada e seu funcionamento básico. Primeiramente, a seção 1 discorre sobre as classes e seus componentes. Em seguida, a seção 2 comenta sobre as threads do programa. Por fim, há a conclusão do relatório.

2 Classes

2.1 Tiles

A classe Tiles é a menor e mais simples classe da aplicação. Basicamente, essa classe é responsável por criar a matriz de dados que contém os elementos do jogo e definir os diferentes elementos possíveis. Na figura 1, é possível visualizar uma parte da matriz (maze) com seus valores iniciais, sendo que ‘W’ representa um muro. Nesse caso, os personagens checavam essas posições e evitavam o contato com estes. O modo como o acesso a esse recurso compartilhado mantém a consistência entre as *threads* é discutido em seções posteriores.

Além disso, o único método de Tiles é chamado resetTiles(), e que apenas faz maze retorna aos seus valores iniciais; método esse que é chamado ao reiniciar o jogo.

A figura 2 demonstra um diagrama UML para essa classe.

2.2 Window

A classe Window é responsável por armazenar as texturas e sprites utilizados pela biblioteca gráfica SFML. A sua estrutura é apresentada na figura 3.

```
Tiles::tiles Tiles::maze[28][31] =
        {
                {W,W,W,W,W,W,W,W,W,W,W,W,u,u,u,W,P,W,u,u,u,W,W,W,W,W,W,W,W,W},
                {W,o,o,o,o,W,W,O,o,o,o,W,u,u,u,W,u,W,u,u,u,W,o,o,o,o,O,o,o,o,W},
                {W,o,W,W,o,W,W,o,W,W,o,W,u,u,u,W,u,W,u,u,u,W,o,W,W,o,W,W,W,o,W},
                {W,o,W,W,o,o,o,o,W,W,o,W,u,u,u,W,u,W,u,u,u,W,o,W,W,o,W,W,W,o,W},
                {W,o,W,W,o,W,W,W,W,W,o,W,u,u,u,W,u,W,u,u,u,W,o,W,W,o,W,W,W,o,W},
                {W,o,W,W,o,W,W,W,W,W,o,W,W,W,W,W,W,u,W,W,W,W,W,o,W,W,o,W,W,o,W},
                {W,o,W,W,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,o,W},
                {W,o,W,W,W,W,W,o,W,W,o,W,W,W,W,W,u,W,W,W,W,W,W,W,W,o,W,W,W,o,W},
                {W,o,W,W,W,W,W,o,W,W,o,W,W,W,W,W,u,W,W,W,W,W,W,W,W,o,W,W,W,o,W},
```

Fig. 1: Matriz "maze"

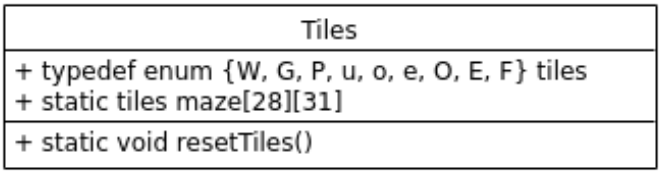

Fig. 2: Diagrama UML para a classe Tiles

Como é possível observar, seus atributos consistem em um semáforo, vetores de sprites que são utilizados para criar as animações e em muitas texturas e sprites variados que são utilizados ao longo do jogo - devido ao grande número nem todos foram apresentados.

Dentre os métodos há o load_and_bind_textures que primeiramente importa os arquivos com as imagens e cria os sprites, depois adiciona os valores aos vetores com os sprites dos personagens e números. Esse método é chamado no construtor da classe. Os *getters* apenas retornam os vetores para a classe do Pac-Man e dos fantasmas.

2.3 Personagem

A classe personagem é uma classe abstrata que define comportamentos básicos para os personagens do jogo (i.e. Pac-Man e fantasmas), principalmente no que concerne a posição e o movimento destes.

Seus atributos são: um mutex, posições x e y (distância em pixels a partir do canto superior esquerdo), a direção para a qual está se movimentando, os sprites e a quantidade destes e um mutex estático para a maze (matriz em Tiles).

Além dos *getters* e *setters*, há (em ordem) uma função para rotacionar os sprites do Pac-Man e uma função para movimentar os personagens com base na direção que retorna o valor de checkPosition(). Também, as funções vir-

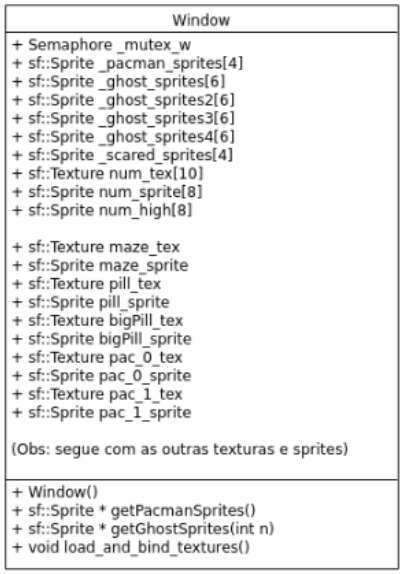

Fig. 3: Diagrama UML para a classe Window

tuais que dependem do personagem em questão que são responsáveis por retornar a sua posição na maze, retornar se algum evento deve ocorrer conforme sua posição (e.g. perder uma vida caso um fantasma encontre o Pac-Man) e atualizar o valor das variáveis estáticas que indicam sua posição e direção. Finalmente, funções estáticas de get e set para a maze e que utilizam o semáforo _mutex_maze.

2.4 PacMan

PacMan é uma classe que herda a classe Personagem e implementa os métodos virtuais desta. Além disso, há versões estáticas dos atributos que indicam a posição e a direção do Pac-Man que são usadas por outras classes.

Como a função para retornar a posição do Pac-Man na maze é diferente da dos fantasmas, há as funções pacmanGetTileX() e pacmanGetTileY(). Quando o personagem não está centralizado em um quadrado da matriz, a função retorna zero. Como às vezes é necessário pegar a posição deste na maze mesmo se não está centralizado (como quando os fantasmas verificam se tocaram o Pac-Man), foram criadas as funções pacmanGetNearTileX() e pacmanGetNearTileY().

2.5 Ghost

A classe Ghost é uma classe abstrata que também herda da classe Personagem, implementando funções bem semelhantes àquelas de PacMan.

As diferenças estão relacionadas aos sprites dos olhos dos fantasmas, ao destino que é objetivo do fantasma e segue uma lógica diferente para cada um e booleanos de estado para indicar se está "preso" na gaiola central (chamada *jail*) ou se está vulnerável ao Pac-Man (*scared*).

O método getTargetTile() é que implementa a lógica individual para cada fantasma e ajusta a direção para o personagem. Já a scareRunAway() implementa a mesma funcionalidade, mas para quando os fantasmas estão fugindo do Pac-Man. Nesse modo, todos os fantasmas assumem o mesmo comportamento.

2.6 Ghost1, Ghost2, Ghost3 e Ghost4

As classes Ghost1, Ghost2, Ghost3 e Ghost4 herdam da classe Ghost e são muito semelhantes entre si. Cada uma tem apenas atributos estáticos para sua posição e direção, setters para estes e implementam sua lógica própria para obter a sua *target tile*.

2.7 Jogo

A classe Jogo, como é possível observar na figura 6, é a maior e mais complexa de todas. É a classe principal do jogo, que cria as *threads* e guarda a estatísticas do jogo.

Nos atributos é possível perceber que ela detém os objetos das classes listadas anteriormente, estatísticas, *threads*, semáforos e booleanos de estado para indicar o início ou o pause do jogo. Vale ressaltar o atributo _window_render, que é um elemento do SFML representando a janela em que o jogo é apresentado.

Entre os métodos há a função principal run, as funções para cada *thread* (iniciadas com run), *getters* e *setters* para os estados e funções para o controle do jogo em seus estados. Esses métodos ficarão mais claros na seção seguinte, em que são discutidas as *threads*.

3 THREADS

3.1 run()

Essa é a função passada em main.cc como a função principal do programa. Nela, antes de criar as outras *threads*, o recorde (ou *highscore*) é obtido de um arquivo de texto. Depois, são criadas as *threads* dos fantasmas, da janela e do Pac-Man, terminando com *joins* nestas. Desse modo, quando essa função termina, o destrutor é chamado e a memória alocada dinamicamente é liberada.

3.2 runWindow()

Essa *thread* é a responsável por criar a parte gráfica e criar animações. Primeiramente, esta cria o objeto _window_render, em seguida cria a *thread* encarregada de verificar os inputs e então entra em um loop que dura até que a janela seja fechada.

Nesse loop, são verificadas as diferentes condições de estado do jogo com as operações isPaused() e isStarting(). Essas operações são *getters* protegidos por semáforos, evitando *data races*. Para cada condição de estado ela realiza as operações necessárias para ilustrar o jogo.

Vale destacar a operação de pause do jogo, que é utilizada também nas outras *threads*, com exceção da que controla os inputs. Essa funcionalidade de interromper o jogo por um momento é feita com o uso de um mutex, sendo que sempre que as *threads* identificam que _isPaused é verdadeiro, estas chamam a operação p() e são adormecidas. Quando há um input para retirar as *threads* dessa condição, ocorre um wakeup_all() e o jogo retoma o fluxo normal.

3.3 runInput()

É esta que controla as entradas do teclado e repassa às outras *threads*. Basicamente, quando há uma entrada de nova direção para o pacman, esta chama _pacman.changeDirection(direção). Quando há um 'P', realiza a

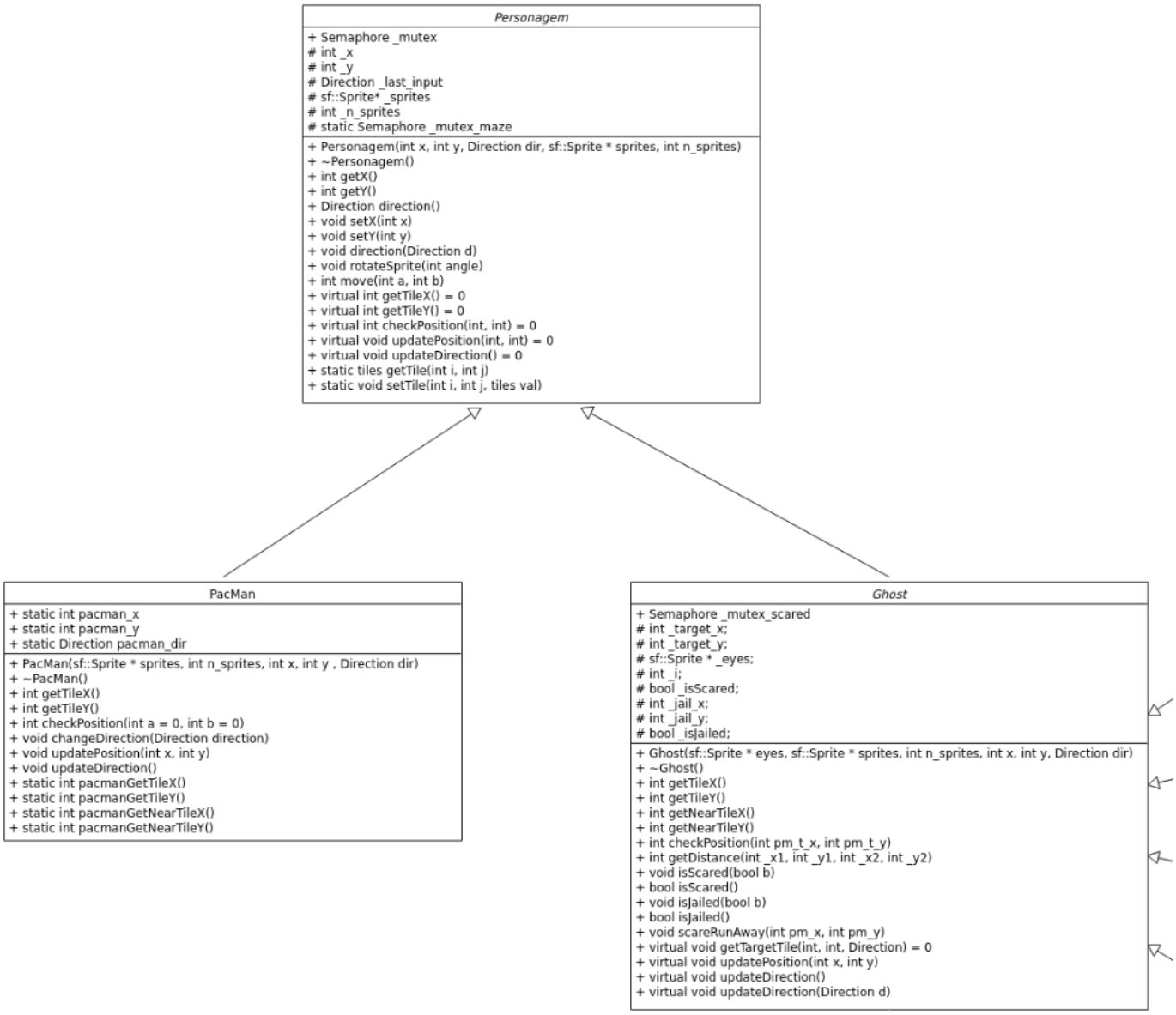

Fig. 4: Diagrama UML para a classe Personagem e classes filhas

operação isPaused(true), que é protegida por mutex. Desse modo as outras *threads* serão interrompidas até que outra entrada 'P' aconteça e wakeup_all() seja chamada. Também, se o jogo foi perdido, ao inserir uma entrada 'P' o jogo é reiniciado. Ao receber 'Q', chama finishGame(), que fecha a janela e encerra o jogo. Por fim, ao receber 'R', chama restartGame() e o jogo é reiniciado.

3.4 runGhost()

Essa thread apenas dá início às outras *threads* dos fantasmas, e é esta que é chamada pela run(). Além disso, nessa thread que ocorre um delay antes de o jogo iniciar, criando o intervalo em que o jogador se prepara para começar a jogar. Assim, ao final desse delay é chamada isStarting(false) e as outras *threads* correspondem à mudança.

3.5 runGhost1(), runGhost2(), runGhost3() e runGhost4()

Cada uma destas é responsável por criar as diferenças de tempo para que cada fantasma saia da *jail*, pelo movimento de cada fantasma - chamando getTargetTile() e depois move() - e por verificar se houve o choque entre um fantasma e o Pac-Man. Caso tenha ocorrido, a função chama loseLife().

A função loseLife() reposiciona os fantasmas em sua posição inicial, remove a fruta do mapa caso necessário e recria o delay inicial que ocorre através do atributo _isStarting.

3.6 runPacman()

De modo semelhante, controla o movimento do personagem Pac-Man por meio da função move(). Além disso, o resultado dessa função retorna se ele "comeu" um círculo

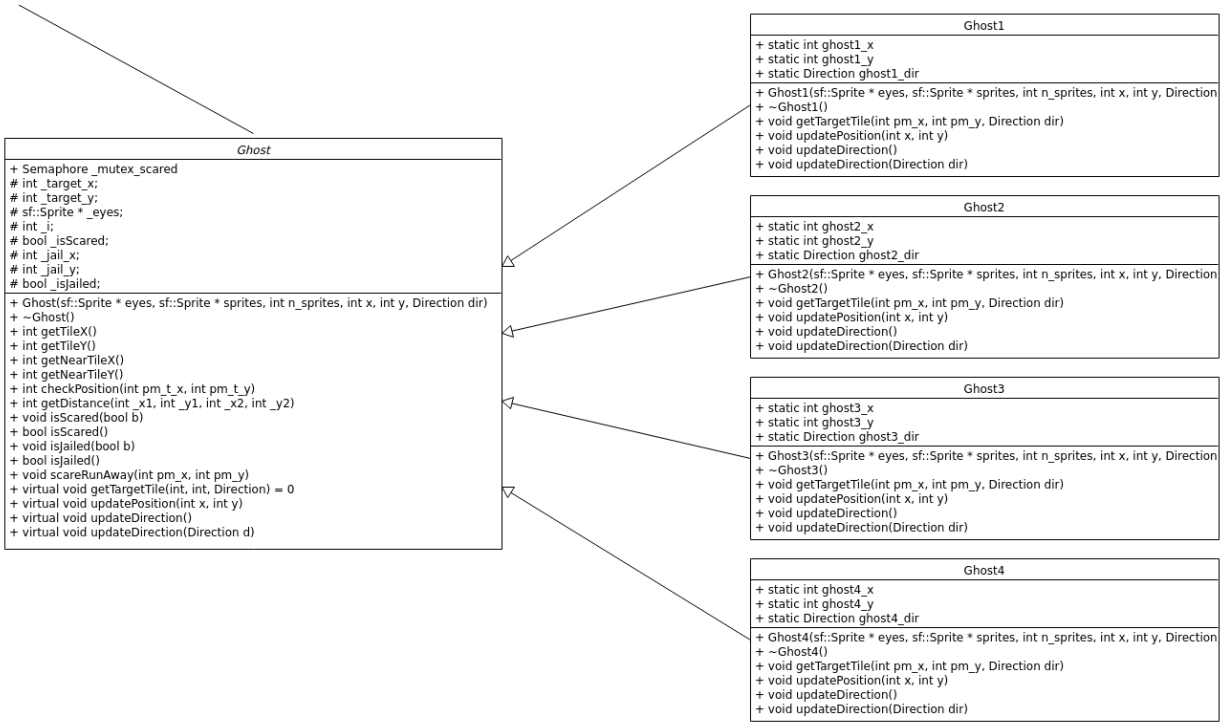

Fig. 5: Diagrama UML para as classes filhas de Ghost

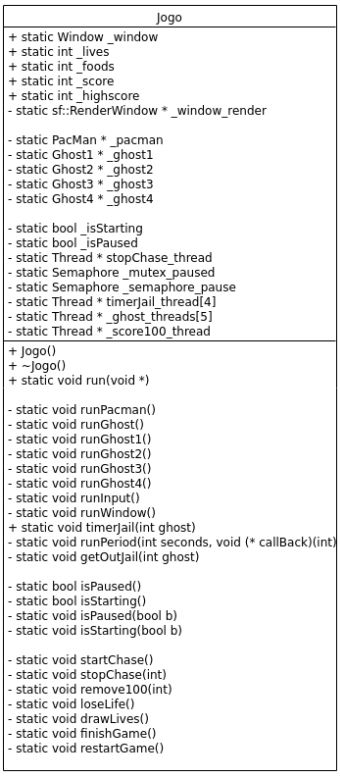

Fig. 6: Diagrama UML para a classe Jogo

pequeno, grande, ou uma fruta. Desse modo, controla a pontuação e a quantidade de "_foods" no labirinto.

São 240 "_foods" inicialmente. Quando esse valor atinge 170, é liberada a primeira fruta (a cereja). Quando atinge 70 é liberada a segunda (o morango). Também, se ele "comeu" uma fruta, é iniciada a animação de 100 pontos na tela. Essa animação funciona da seguinte maneira: na maze, o valor 'F' é substituído por um 'f' e, com isso, a runWindow() já fará a animação do sprite. Também, é lançada uma nova *thread* com uma função auxiliar, a runPeriod(), a qual chama uma função de callback passada depois de decorridos um certo tempo em segundos. Esse tempo pode ou não ser considerado enquanto o jogo está pausado. No caso em questão, após dois segundos a função remove100() - que remove o 'f' da maze - é chamada, e o tempo pausado não é considerado.

É muito semelhante quando este "come" um círculo grande. A função startChase() é chamada, e esta cria uma runPeriod() para a função stopChase(). Durante este período, o Pac-Man pode "comer" os fantasmas, que vão para a *jail*. Ao ir para a *jail*, o fantasma fica preso por um intervalo randômico entre um e sete segundos, seguindo um funcionamento muito semelhante usando a runPeriod(), a timerJail() e a getOutJail().

4 Conclusão

Portanto, através deste documento foi possível compreender o processo de implementação de um jogo Pac-Man utilizando um sistema de gerenciamento de *threads* desenvolvido ao longo do semestre na disciplina de sistemas operacionais.

Evidentemente, há diferentes abordagens para o problema, de modo que a implementação possa seguir outros caminhos e ser aprimorada.

5 REFERÊNCIAS

[1] TANENBAUM, Andrew S.; WOODHULL, Albert S. **Operating systems: design and implementation**. Englewood Cliffs: Prentice Hall, 1997.

[2] MAZIERO, Carlos A. Sistemas operacionais: conceitos e mecanismos. **Livro aberto. Acessível em: http://wiki. inf. ufpr. br/maziero/lib/exe/fetch. php**, 2014.

[3] SFML. Simple and Fast Multimedia Library. **Acessível em https://www.sfml-dev.org/**, 2020.