UNIVERSIDADE FEDERAL DE ITAJUBÁ – UNIFEI – CAMPUS ITABIRA
CURSO DE ENGENHARIA DA COMPUTAÇÃO - ECO
SISTEMAS OPERACIONAIS
PROF. JULIANO DE ALMEIDA MONTE-MOR

PROJETO 2: JOGO FREEWAY - ATAR

ALUNO:JEAN CLAUDIO DE SOUZA **RA:2017002917**
ALUNO:PEDRO HENRIQUE FIGUEIREDO COTA OLIVEIRA **RA:2017017769**
ALUNO:RODRIGO AUGUSTO LIMA **RA:2019010097**

1. INTRODUÇÃO

Neste projeto, será desenvolvido um programa em Java para o jogo "Game One: Lake Shore Drive". O objetivo é criar uma implementação do jogo utilizando threads e semáforos para o sincronismo das threads. Serão utilizadas estruturas de dados do tipo matriz para representar as possíveis posições das galinhas e dos veículos no jogo. Serão criadas threads separadas para representar os jogadores, os veículos e também para desenhar a tela do jogo.

O "Game One: Lake Shore Drive" é um jogo clássico no qual os jogadores controlam galinhas que precisam atravessar uma movimentada avenida de várias faixas de tráfego. O objetivo é guiar as galinhas em segurança até o outro lado da avenida, evitando colisões com os veículos em movimento.

O uso de threads permitirá que diferentes aspectos do jogo sejam tratados de forma assíncrona e simultânea, tornando a jogabilidade mais dinâmica. As threads dos jogadores serão responsáveis pelo controle das galinhas, as threads dos veículos gerenciarão o movimento dos carros na avenida e a thread de desenho será responsável por exibir a tela do jogo de forma contínua.

O uso de semáforos permitirá a sincronização adequada entre as threads, evitando condições de corrida e garantindo que as ações dos jogadores e veículos ocorram de forma segura e organizada.

Ao final do projeto, espera-se ter uma implementação funcional do jogo "Game One: Lake Shore Drive" em Java, com o uso de threads e semáforos para o sincronismo das ações.

2. OBJETIVOS

O objetivo principal deste projeto é desenvolver um programa em Java para o jogo "Game One: Lake Shore Drive", utilizando threads e semáforos para o sincronismo das threads. Além disso, serão utilizadas estruturas de dados do tipo matriz para representar as possíveis posições das galinhas e veículos no jogo.
Os objetivos específicos incluem:

1. Implementar as threads dos jogadores: Serão criadas threads separadas para representar os jogadores, que controlam as galinhas no jogo. Essas threads serão responsáveis pelo movimento das galinhas e pela interação com os veículos.
2. Implementar as threads dos veículos: Serão criadas threads separadas para representar os veículos em movimento na avenida. Essas threads serão responsáveis por gerenciar o movimento dos veículos e detectar colisões com as galinhas.
3. Utilizar semáforos para sincronização: Serão utilizados semáforos para garantir a sincronização adequada entre as threads dos jogadores e dos veículos. Os semáforos serão utilizados para controlar o acesso às posições da matriz que representam as posições das galinhas e veículos.
4. Criar uma estrutura de dados do tipo matriz: Será utilizada uma estrutura de dados do tipo matriz para representar as posições das galinhas e veículos no jogo. Essa matriz será atualizada pelas threads dos jogadores e dos veículos de forma coordenada.
5. Desenhar a tela do jogo: Será criada uma thread separada para desenhar a tela do jogo de forma contínua, mostrando as posições atualizadas das galinhas e veículos na matriz.

Além disso, foi preciso atender às regras dos jogos.

3. DESENVOLVIMENTO

Durante o desenvolvimento do trabalho prático, foi utilizado o Java Swing para a criação do jogo, possibilitando a criação de janela de execução. Para configurar o JFrame, foram utilizadas as funções no arquivo *main.java*:

```
JFrame window = new JFrame();
```

```
// Define a operação padrão de fechamento de janela para término do
jogo
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Impede que o usuário redimensione a janela
window.setResizable(false);
window.setTitle("FreeWay SO");
```

Feito a configuração do JFrame, o próximo passo foi configurar o GamePanel, que é uma classe personalizada que representa o painel do jogo.

```
GamePanel gamePanel = new GamePanel();
window.add(gamePanel);
// Redimensionamento de janela para componentes
window.pack();
// Centralização da janela
window.setLocationRelativeTo(null);
// Visibilidade da janela
window.setVisible(true);
// Inicialização da Thread do jogo
gamePanel.startGameThread();
```

Todo esse conteúdo apresentado acima pode ser encontrado facilmente na web, pois são modelos padrões para inicialização de GamePanel.
Com isso, foi inicializada a criação do jogo FreeWay para a disciplina de Sistemas Operacionais. A primeira etapa foi criar uma matriz do mapa, esta se chama *MeuMapa.txt* e tem o seguinte formato:

```
//Parede Superior
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
//Calçada Superior
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
//Rua Superior
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
//Meio da rua
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
//Rua Inferior
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
//Calçada Inferior
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
//Parede Inferior
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

Este arquivo foi carregado dentro de *TileManager.java* que podem ser encontrados modelos na web para carregamento de mapas.
Feito o mapa, foi carregado a textura com o seguinte trecho de código ao qual as texturas foram carregadas na matriz apresentada acima.

```
// Cria mapa de acordo com sprites da matriz
public void geTileImage() {
try {
tile[0] = new Tile();
tile[0].image =
ImageIO.read(getClass().getResourceAsStream("/piso.png"));
tile[1] = new Tile();
tile[1].image =
ImageIO.read(getClass().getResourceAsStream("/branco1.png"));
tile[2] = new Tile();
tile[2].image =
ImageIO.read(getClass().getResourceAsStream("/amarelo1.png"));
tile[3] = new Tile();
tile[3].image =
ImageIO.read(getClass().getResourceAsStream("/branco2.png"));
tile[4] = new Tile();
tile[4].image =
ImageIO.read(getClass().getResourceAsStream("/amarelo2.png"));
tile[5] = new Tile();
tile[5].image =
ImageIO.read(getClass().getResourceAsStream("/parede.png"));
} catch (IOException e) {
e.printStackTrace();
}
}
```

Com a textura carregada temos o mapa dimensionado e colorido. O segundo passo foi criar os carros que irão transitar pela rua, foi criado o arquivo *Carro.java* para

instanciar objetos carros que irão transitar nos níveis da rua, ao qual cada nível apresenta um limite diferente de velocidade, definido pelo código abaixo:

```
// Configura velocidade do carro
public void setSpeed() {
int Speeds[] = {1, 2, 3, 4, 6};
speed = Speeds[RelPosition];
}
```

Fazendo as devidas alterações de posições dos carros, foi instanciado a textura para cada um deles, mostrado no código abaixo:

```
// Sprite do carro
public void getPlayerImage() {
String colors[] = {"amarelo.png", "azul.png"};
String colors2[] = {"amarelo.png", "azul.png"};
Random rand = new Random();
int color = rand.nextInt(2);
try {
if(direction == 0) {
look =
ImageIO.read(getClass().getResourceAsStream("/"+colors2[color]));
} else {
look =
ImageIO.read(getClass().getResourceAsStream("/"+colors[color]));
}
} catch (IOException e) {
e.printStackTrace();
}
}
```

Atributos como a movimentação podem ser obtidos na web. Neste mesmo arquivo foi implementado o primeiro código de exclusão mútua, cujo objetivo é a sincronização para evitar problemas de concorrência em ambientes que utilizaram mais de uma Thread.

```
// Mutex para colisão inferior
private void alteraMatrizBaixo() {
try {
//Ativa região crítica
gp.mutex.acquire();
```

```
//Faz todo o tratamento de colisão
} catch (InterruptedException e) {
e.printStackTrace();
} finally {
//Libera região crítica
gp.mutex.release();
}
}

// Mutex para colisão superior
private void alteraMatrizAlto() {
try {
//Ativa região crítica
gp.mutex.acquire();
//Faz tratamento de colisão
} catch (InterruptedException e) {
e.printStackTrace();
} finally {
//Sai da região crítica
gp.mutex.release();
}
}

// Posição para colisão
private void inicializaMatriz() {
try {
//Ativa região crítica
gp.mutex.acquire();
//Inicializa matriz com todo tratamento de colisão
} catch (InterruptedException e) {
e.printStackTrace();
} finally {
//Libera toda a região crítica
gp.mutex.release();
}
}
```

Com os mapas e carros prontos, foi criado o arquivo *player.java* para criar o player e seu controle de colisão, sempre mantendo a região crítica entre *gp.mutex.acquire()* e *gp.mutex.release()*. Diferente dos carros, o objeto player será controlado pelos usuários, além de que cada player, tanto o 1 quanto o 2, estarão rodando em Threads diferentes, o que geralmente causa diversos problemas caso o conteúdo não esteja bem definido dentro da região crítica do código. O código para criação do player pode

ser encontrado em diversos exemplos na web, mas a parte de exclusão mútua é apresentado abaixo:

```
// Criação do player na tela
private void inicializaPosicaoMatriz() {
try {
//Proteção da região crítica
gp.mutex.acquire();
//Liberação de cada player para uma thread por região crítica
gp.cc.matriz[linhaAtual][colunaAtual] = idPlayer;
} catch (InterruptedException e) {
e.printStackTrace();
} finally {
//Liberação da região crítica
gp.mutex.release();
}
}
```

O controle de colisão do player também tem que ser bem definido, pois cada Thread terá o direito de acessar o código de colisão. O código abaixo mostra como deve ser essa implementação:

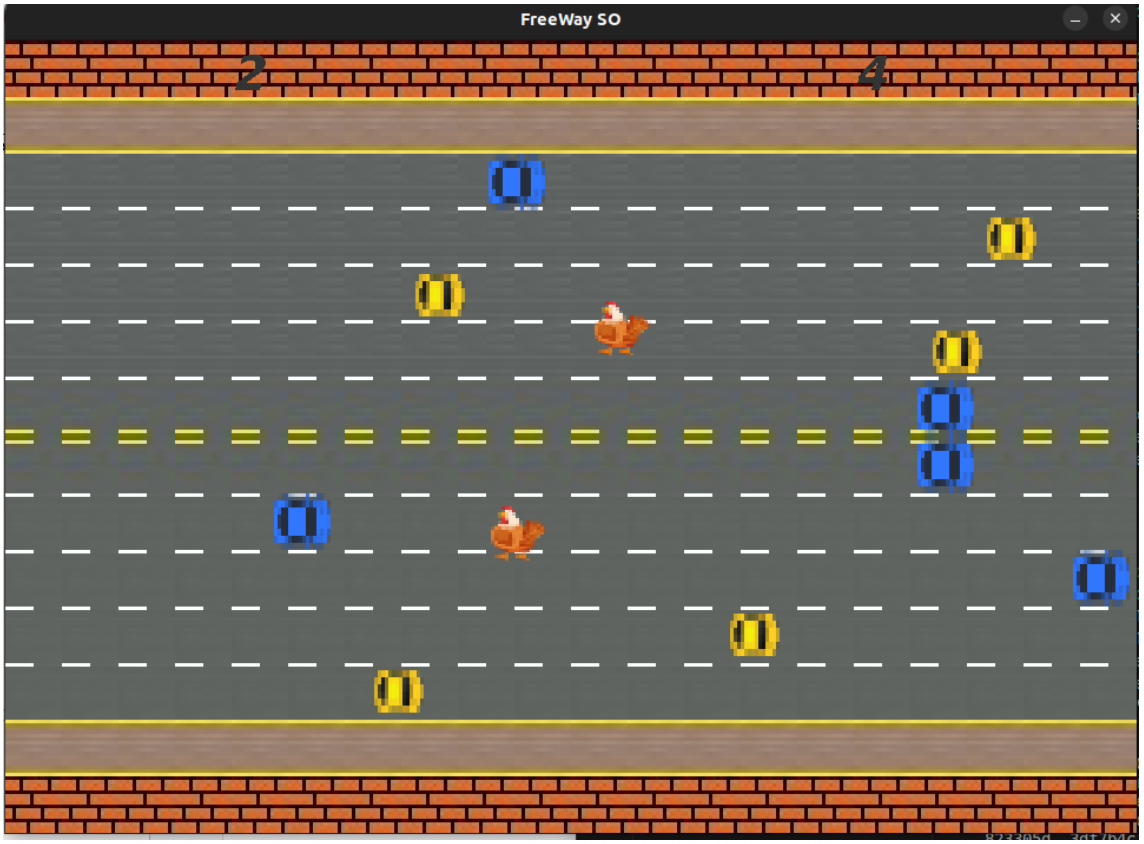
```
try {
//Entra na região crítica
gp.mutex.acquire();
//Acesso de uma thread por vez para player 1 ou 2
gp.cc.matriz[linhaAtual][colunaAtual] = idPlayer;
} else {
//Tratamento de colisão por Thread
if (idPlayer == 1) {
gp.cc.colision1 = true;
} else {
gp.cc.colision2 = true;
}
}
}
} catch (InterruptedException e) {
e.printStackTrace();
} finally {
//Liberação da região crítica
gp.mutex.release();
}
```

O mesmo deve ser feito caso detecte uma colisão do player com o objeto e este precise ser resetado para a posição inicial, lembrando que apenas uma Thread pode acessar a região crítica por vez.

```
// reseta player
public void resetPosition() {
//Verifica qual Thread causou a colisão
if (gp.cc.matriz[linhaAtual][colunaAtual] == idPlayer) {
//Faz o tratamento (não precisa de região crítica pois o código
acima permite que apenas uma Thread solicite colisão por vez
}
//Reseta a Thread que causou a colisão
gp.cc.matriz[linhaDefault][colunaDefault] = idPlayer;
}
```

Com isso foi feito o solicitado para o trabalho prático da disciplina de Sistemas Operacionais. Os demais arquivos:

- *ControleColisão.java* serve para controlar a colisão de cada Thread que solicitou a colisão nos códigos anteriores, seguindo a mesma receita de *gp.mutex.acquire()* e *gp.mutex.release()*.
- *Entity.java* serve para definir a entidade posição e velocidade do buffer de imagem.
- *GamePanel.java* serve para definir quais Threads terão acesso a região crítica por vez, além de definir a variável mutex do tipo Semaphore (semáforo): *mutex = new Semaphore(1,true);* com o nível de permissão 1 e valor booleano True. Este arquivo apresenta também o código *player1.startThread();* e *player2.startThread()*, o que garante que cada player terá acesso a uma Thread.
- *KeyHandler.java* serve para definir os botões que cada player usará para o jogo, o player 2 use Arrow_Up e Arrow_Down, o player 1 use W e S.
- *Ruas.java* serve para configurar qual direção cada carro irá percorrer em qual rua.
- *Score.java* mostra o placar do jogo, quando o jogador chega ao final da rua sem nenhuma colisão.
- *Tile.java* serve para configurar a sincronização da textura com o objeto que está sendo atribuída a ela.



4. CONCLUSÕES

A implementação adequada de semáforos, exclusão mútua, Threads e sincronismo é fundamental para garantir a integridade dos dados, a estabilidade do jogo e uma experiência fluente para o usuário. O uso correto destes conceitos permite que diferentes aspectos como jogo, gráficos, lógica e entrada do jogador trabalhem harmoniosamente, resultando em uma experiência de jogo mais envolvente e livre de problemas de concorrência. Alḿe disso, a abordagem multi thread pode aproveitar recursos de hardware moderno para melhorar o desempenho geral e proporcionar uma jogabilidade mais imersiva e responsiva.

5. REFERÊNCIAS BIBLIOGRÁFICAS

Activision Freeway Instructions. Disponível em <https://www.gamesdatabase.org/Media/SYSTEM/Atari_2600/Manual/formated/Freeway_-_1981_-_Zellers.pdf>. Acesso em 27 jun. 2023.

Lesson: Overview of the Java 2D API Concepts. Disponível em <https://docs.oracle.com/javase/tutorial/2d/overview/index.html>. Acesso em 27 jun. 2023.

ANEXOS

Inserir como anexo os códigos fontes dos programas desenvolvidos, bem como quaisquer outros materiais complementares. Os anexos precisam ser numerados e referenciados no texto.