# Manual tecnino ejercicio 3

### Partes del programa en donde existieron múltiples procesos trabajando de forma concurrente y/o paralela.

Se utilizaron ciclos para ejecutar la lógica de juego para ciertas entidades importantes:

* Jugador

public class RunnableJugador **implements** Runnable**{**

@Override

public void run**()** **{**

// correr cycle

long lastTime **=** System**.**currentTimeMillis**();**

long timeToWait**;**

**while(!**getRemove**()** **&&** game**.**getRunning**())** **{**

cycle**();**

long deltaTime **=** System**.**currentTimeMillis**()** **-** lastTime**;**

timeToWait **=** game**.**DELAY **-** deltaTime**;**

**if(**timeToWait **<** 0**)** **{**

timeToWait **=** 2**;**

System**.**out**.**println**(**"WARNING: timeToWait menor que 0!!!"**);**

**}**

**try** **{**

Thread**.**sleep**(**timeToWait**);**

**}** **catch** **(**InterruptedException e**)** **{**

String msg **=** String**.**format**(**"Thread interrupted: %s"**,**

e**.**getMessage**());**

JOptionPane**.**showMessageDialog**(null,** msg**,**

"Error"**,**

JOptionPane**.**ERROR\_MESSAGE**);**

**}**

lastTime **=** System**.**currentTimeMillis**();**

**}**

**}**

**}**

* Spawner

public class RunnableSpawner **implements** Runnable**{**

//@Override:

public void run**()** **{**

// correr cycle

long lastTime **=** System**.**currentTimeMillis**();**

long timeToWait**;**

**while(**game**.**getRunning**())** **{**

cycle**();**

long deltaTime **=** System**.**currentTimeMillis**()** **-** lastTime**;**

timeToWait **=** game**.**DELAY **-** deltaTime**;**

**if(**timeToWait **<** 0**)** **{**

timeToWait **=** 2**;**

System**.**out**.**println**(**"WARNING: timeToWait menor que 0!!!"**);**

**}**

**try** **{**

Thread**.**sleep**(**timeToWait**);**

**}** **catch** **(**InterruptedException e**)** **{**

String msg **=** String**.**format**(**"Thread interrupted: %s"**,**

e**.**getMessage**());**

JOptionPane**.**showMessageDialog**(null,** msg**,**

"Error"**,**

JOptionPane**.**ERROR\_MESSAGE**);**

**}**

lastTime **=** System**.**currentTimeMillis**();**

**}**

**}**

**}**

Un problema que se tuvo es que, por la naturaleza de como se dibujan graficos en el JPanel, teníamos que dibujar todas las entidades de juego en ‘una pasada’. Tambien necesitábamos correr el juego a un frame-rate razonable y constante. Por ello, se implemento un thread que se encargaría de redibujar el juego 67 veces por segundo.

@Override

public void run**()** **{**

long lastTime **=** System**.**currentTimeMillis**();**

long timeToWait**;**

long lastTimeFps **=** System**.**currentTimeMillis**();**

int fpsCount **=** 0**;**

**for(**Jugador jugador **:** jugadores**)** **{**

jugador**.**thread**.**start**();**

**}**

spawner**.**thread**.**start**();**

**while(true)** **{**

repaint**();**

fpsCount**++;**

**if(**getRunning**())** **{**

cycle**();**

**}**

long deltaTime **=** System**.**currentTimeMillis**()** **-** lastTime**;**

timeToWait **=** DELAY **-** deltaTime**;**

**if(**timeToWait **<** 0**)** **{**

timeToWait **=** 2**;**

System**.**out**.**println**(**"WARNING: timeToWait menor que 0!!!"**);**

**}**

**try** **{**

Thread**.**sleep**(**timeToWait**);**

**}** **catch** **(**InterruptedException e**)** **{**

String msg **=** String**.**format**(**"Thread interrupted: %s"**,**

e**.**getMessage**());**

JOptionPane**.**showMessageDialog**(this,** msg**,**

"Error"**,**

JOptionPane**.**ERROR\_MESSAGE**);**

**}**

// Este deltaTime es bastante impresiso pero no importa porque los

// fps son solo para debugging

long timeDiffFps **=** System**.**currentTimeMillis**()** **-** lastTimeFps**;**

**if(**timeDiffFps **>=** 1000**)** **{**

System**.**out**.**println**(**"fps: " **+** fpsCount**);**

lastTimeFps **=** System**.**currentTimeMillis**();**

fpsCount **=** 0**;**

**}**

lastTime **=** System**.**currentTimeMillis**();**

**}**

**}**

Por ultimo se utilizo el thread principal para instanciar la ventana en donde dibujaremos nuestro juego .

public static void main**(**String**[]** args**)** **{**

EventQueue**.**invokeLater**(()** **->** **{**

GameContainer gameContainer **=** **new** GameContainer**();**

gameContainer**.**setVisible**(true);**

**});**

**}**

### Como se realizo la comunicación y sincronización entre procesos.

Se utilizaron principalmente ReentrantReadWriteLocks, ya que, había varias funciones que solo necesitaban leer los datos y no escribir sobre ellos.

private void lasersCycle**()** **{**

lasersLock**.**writeLock**().**lock**();**

**{**

**for** **(**Laser laser **:** lasers**)** **{**

laser**.**cycle**();**

**}**

**for** **(**int i **=** 0**;** i **<** lasers**.**size**();** i**++)** **{**

**if(**lasers**.**get**(**i**).**remove**)** **{**

lasers**.**remove**(**i**);**

i**--;**

**}**

**}**

**}**

lasersLock**.**writeLock**().**unlock**();**

**}**

También se utilizaron métodos sinchronized, porque a veces la coordinación era mas simple y no necesitábamos la complejidad de tener que determinar si un bloque de código solo escribía o leía sobre un recurso compartido

public synchronized void setRemove**(**boolean b**)** **{**

**this.**remove **=** b**;**

**}**

public synchronized boolean getRemove**()** **{**

**return** **this.**remove**;**

**}**

### Situaciones en las cuáles era posible que se dieran:

#### Deadlocks:

game**.**spawner**.**enemigosLock**.**readLock**().**lock**();**

**{**

**for** **(**Enemigo enemigo **:** game**.**spawner**.**enemigos**)** **{**

**if(**enemigo**.**collides**(this.**x**,** **this.**y**,** hitboxWidth**,** hitboxHeight**))** **{**

game**.**spawner**.**enemigosLock**.**writeLock**().**lock**();**

enemigo**.**takeHit**();**

**this.**remove **=** **true;**

game**.**spawner**.**enemigosLock**.**writeLock**().**unlock**();**

**}**

**}**

**}**

game**.**spawner**.**enemigosLock**.**readLock**().**unlock**();**

Lo solucionamos implementa un simple writeLock a todo el bloque de codigo

#### Condiciones de carrera:

El problema principal era el hecho que el thread encargado de hacer render tenia que leer todas las coordenadas de todos los ‘game objects’ del juego y otro thread se encargaba de mover y colisionar los ‘game objects’ . Esto se soluciono agregadon readLock cuando hacíamos render y writeLock cuando modificábamos las coordenadas

public class RunnableSpawner **implements** Runnable**{**

//@Override:

public void run**()** **{**

// correr cycle

long lastTime **=** System**.**currentTimeMillis**();**

long timeToWait**;**

**while(**game**.**getRunning**())** **{**

cycle**();**

long deltaTime **=** System**.**currentTimeMillis**()** **-** lastTime**;**

timeToWait **=** game**.**DELAY **-** deltaTime**;**

**if(**timeToWait **<** 0**)** **{**

timeToWait **=** 2**;**

System**.**out**.**println**(**"WARNING: timeToWait menor que 0!!!"**);**

**}**

**try** **{**

Thread**.**sleep**(**timeToWait**);**

**}** **catch** **(**InterruptedException e**)** **{**

String msg **=** String**.**format**(**"Thread interrupted: %s"**,** e**.**getMessage**());**

JOptionPane**.**showMessageDialog**(null,** msg**,** "Error"**,** JOptionPane**.**ERROR\_MESSAGE**);**

**}**

lastTime **=** System**.**currentTimeMillis**();**

**}**

**}**

**}**

public void render**(**Graphics g**)** **{**

enemigosLock**.**readLock**().**lock**();**

**{**

**for** **(**Enemigo enemigo **:** enemigos**)** **{**

enemigo**.**render**(**g**);**

**}**

**}**

enemigosLock**.**readLock**().**unlock**();**

**}**

### Variables o datos que era necesario compartir entre procesos.

public class Game **extends** JPanel **implements** Runnable**{**

//...

private boolean running**;**

//...

**}**

// sobre cada instacia

public class Enemigo**{**

//...

int x**;**

int y**;**

int hp**;**

//...

**}**

// sobre cada instancia

public class Jugador**{**

//...

int x**;**

int y**;**

int hp**;**

ArrayList**<**Laser**>** lasers**;**

//...

**}**

public class Spawner**{**

ArrayList**<**Enemigo**>** enemigos**;**

**}**