BEN YAAGOUB Nassim

RAPPORT INFO 42

BUGERGERTIME GAME

Java

Sommaire :

I. Explication et découpage du sujet :

Burgertime est un jeu vidéo créé par la société japonaise Data East pour sa borne DECO Cassette System, sorti en 1982. C'est un jeu de plates-formes à l'aspect d'un labyrinthe, où le joueur incarne un chef de cuisine et doit confectionner des hamburgers tout en évitant les autres personnages — représentés par de la nourriture — qui le poursuivent. (Wikipedia)

Afin de créer ce jeu, il faut :

- Une Interface Graphique
- Un joueur ou des joueurs
- Des monstres
- Un monde et des niveaux
- Un système de collision entre les monstres et le joueur
- Un réseau permettant de jouer en LAN

II. Description des structures de données:

- Une Interface Graphique :

Bien que le sujet précisait qu'il n'était pas obligatoire de créer une interface graphique, j'ai préféré en créer une pour une meilleur fluidité du jeu. Pour l'interface graphique, j'ai décidé d'utilisé les librairies *javax.swing.JFrame java.awt.Graphics* qui permettent de dessiner des images, des formes, et du texte. Mais j'avais un problème de rendu d'image, les images du jeu n'étaient pas assez fluides et des écrans noir apparaissaient entre chaque images. J'ai donc utilisé la classe *BufferedStrategy* du *java.awt.image.BufferStrategy*.

- Un joueur ou des joueurs :

Les joueurs devront être contrôlés par l'utilisateur, donc j'ai importé les librairies *java.awt.event.KeyEvent* et *java.awt.event.KeyListener*.

- Des monstres :

Ces monstres pourraient disposer d'une intelligence artificielle qui pourrait chasser le joueur.

- Un monde et des niveaux :

Je pense qu'il est préférable de représenter les niveaux du jeu avec une matrice de nombres, ce qui permet de réduire le temps d'écriture du projet et d'améliorer sa lisibilité. Les monstres et le joueur ne pourront se déplacer que sur les plateformes et les échelles.

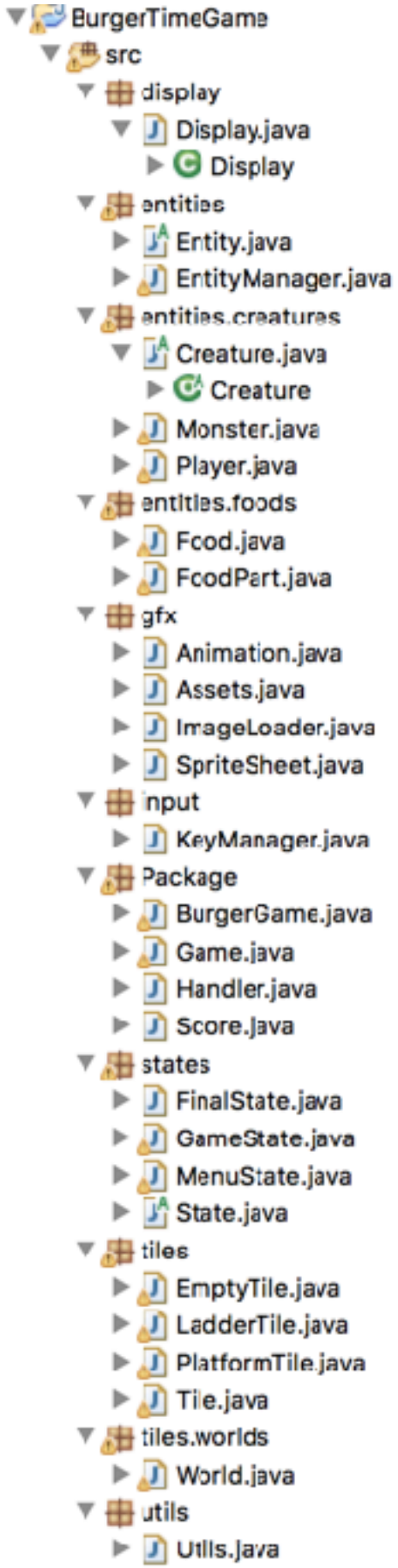
- Un système de collision entre les monstres et le joueur

Ce système se fera à l'aide de la classe *java.awt.Rectangle,* ainsi que de sa méthode *intersects()*

- Un réseau permettant de jouer en LAN

Je n'ai pas pu faire le réseau par manque de temps, mais il faudrait créer un serveur à partir d'un des deux ordinateur qui reçoit les touches pressée par les utilisateurs et renvoi les positions des objets à affiché au deux ordinateurs.

III. Spécification des classes principales



-> Game.java

Cette classe est la classe principale du jeu, elle est lancé au début du programme, et elle implémente l'interface Runnable

```
private Display display;
private static int DEFAULT_GAME_WIDTH = 1045;
private static int DEFAULT_GAME_HEIGHT = 1000;

private int width, height;
private String title;
private boolean end;

private boolean running = false;
private Thread thread;
private Timer timer1;

private BufferStrategy bs;
private Graphics g;


//States
private State gameState;
private State menuState;
private State finalState;

//Input
private KeyManager keyManager;

//Handler
private Handler handler;
```

Constructeur :

```
public Game(String title) {
        this.height = DEFAULT_GAME_HEIGHT;
        this.width = DEFAULT_GAME_WIDTH;
        this.title = title;
        timer1 = new Timer();
        end = false;
        keyManager = new KeyManager();
}
```

Méthodes Principales :

public void run()

Cette méthode est obligatoire pour faire fonctionner l'interface Runnable et est lancé directement à la création de l'instance. Elle possède une boucle infini ou les méthodes tick() et render() sont appelés.

public void render()
public void tick()

Ces méthodes sont présentes dans presque toutes les classes du programme, la méthode tick() permet de rafraîchir les données des classes (la position des monstres et du joueur, la position de la nourriture, les hitbox des entités…) et la méthode render() permet d'afficher l'interface graphique. Ces méthodes appellent les méthodes tick() et render(Graphics g) des classes de type *State.*

La méthode render est détaillé en commentaire dans le code.

public synchronized void start()
public synchronized void stop()

Ces méthodes permettent de démarrer le thread de la classe et de le stopper.

-> Handler.java

Cette classe permet d'acceder aux principales classes à partir de n'importe quelle classe, elle peut aussi permettre d'éviter des lignes de codes trop longues.

```
private Game game;
private World world;
private EntityManager entityManager;
private Score score;
```

-> State.java

Pour passer d'un écran à un autre, j'utilise la classe State. Cette classe possède un field de type State qui représente l'état actuel du Jeu.

-> MenuState.java

Hérite de la classe State et est lancée directement avec la classe Game, cette classe affiche l'écran de chargement. Je comptais utiliser cette classe pour implémenter le choix d'options tel que la rapidité du joueur, sa vie, etc….

-> GameState.java

Hérite de la classe State, une des classes principale, elle créer une instance de la classe World et EntityManager et appelle leurs méthodes tick() et render() et vérifie si le jeu doit se terminer afin de lancer le FinalState.

-> FinalState.java

Hérite de la classe State, elle affiche le résultat de la partie et gère les meilleurs scores.

Le meilleur score est chargé à partir du fichier score.txt, met à jour le meilleur score si il le faut et l'affiche.

-> World.java

```
private int width, height;
private int[][] tiles;
private Handler handler;
```

Cette classe charge la matrice présente dans le fichier world1.lvl, créer la map à partir de celle-ci, et la stock dans le tableau *tiles[][],* on affiche ensuite chaque parcelle de ce monde.

-> Tile.java

```
public static Tile[] tiles = new Tile[256];
public static Tile emptyTile = new EmptyTile(0);
public static Tile platformTile = new PlatformTile(1);
public static Tile ladder1Tile = new LadderTile(2);
public static Tile ladder2Tile = new LadderTile(3);
public static Tile ladder3Tile = new LadderTile(4);
public static Tile ladder4Tile = new LadderTile(5);
public static Tile emptySolidTile = new EmptyTile(6);

public static int       WIDTH = 55,
                        HEIGHT = 30;
private int width, height;
protected BufferedImage texture;
protected final int id;
```

Cette classe représente une parcelle de la map, elle possède une ID et en fonction de l'ID entré, créer une parcelle de type EmptyTile, PlatformTile ou LadderTile. La map est donc créé en entrant les numéros de la matrice comme ID dans le constructeur de cette classe. Donc pour un numéro 6 dans la matrice, c'est un emptySolidTile qui est crée.

Chaque sous-classe possède une texture, et une méthode isSolid(), si cette méthode renvoie true, le joueur et les monstres ne pourront pas marcher sur cette parcelle. Le détail de la collision seront effectué dans la classe Creature.

-> Assets.java

```
public static BufferedImage platform, player, empty, emptySolid,
                            ladder1, ladder2, ladder3, ladder4,
                            meal1, meal2, meal3,
                            topBread1,topBread2,topBread3,
                            botBread1,botBread2,botBread3,
                            salad1, salad2, salad3,
                            tomato1, tomato2, tomato3,
                            egg1, egg2, egg3,
                            foodPlatform1, foodPlatform2, foodPlatform3,
                            loaderImage, newHighImage;

public static BufferedImage[]
                            player_down, player_left, player_up, player_right,
                            salad_down, salad_left, salad_up, salad_right,
                            pepper_down, pepper_left, pepper_up, pepper_right,
                            egg_down, egg_left, egg_up, egg_right;
```

Pour charger les images, j'ai recours à une classe qui permet de charger qu'une seule grande image et ensuite de découper cette image en petites images, ce qui permet d'augmenter considérablement le temps de chargement du programme. L'image est donc coupée et stocker dans plusieurs static variable afin d'y accéder à partir de tout les programmes.

-> Entity.java

Classe parent de toutes les entités du jeu (Player.java, Monster.java, Food.java), elle possède les méthodes abstraites render() et tick(). Toute ces classe possède un objet bounds de type Rectangle qui représente la zone matérielle de l'entité et grâce à cet objet, on peux créer un système de collision que je détaillerais dans la classe Creature.java.

Afin de placer tous les entités de manières plus simple, j'ai utilisé les variables Tile.WIDTH et Tile.HEIGHT, en effet, la map créer une matrice de parcelles, et pour placer les entités, il suffit de multiplier la largeur/longueur et la position d'une parcelle pour placer l'entité sur la parcelle en question.

-> Food.java

```
public FoodPart leftPart;
public FoodPart middlePart;
public FoodPart rightPart;
private Handler handler;

private int line;
private int column;
private int lineInArray, columnInArray;

private boolean down;
```

Cette classe représente la nourriture du jeu, elle possède 3 objets de types FoodPart qui constitue la nourriture. La nourriture est donc divisé en 3 parties et la nourriture tombe jusqu'à la prochaine plate-forme lorsque les 3 parties sont *down* ou lorsque un ingrédients tombe sur un autre. Les variables lineInArray et columnInArray représente la place de la nourriture dans le tableau de Food disponible dans la classe EntityManager.

-> FoodPart.java

```
public static enum TYPEFOOD {
        Meal,
        TopBread,
        BotBread,
        Tomato,
        Salad,
        Platform,
        None
}
public static enum PART {
        Left,
        Right,
        Middle
}
private PART part;
private TYPEFOOD typeFood;
private int line;
private int column;
private BufferedImage texture;
private boolean down;
```

Chaque FoodPart est représenté par un type de nourriture et sa partie dans la Food classe. En fonction de ces paramètres dans le constructeur, sa texture et sa position varient. Les variables *line* et *column* représente la position du FoodPart dans la matrice de la map.

-> Creature.java

```
protected int health;
protected Rectangle hitBox;
protected float speed;
protected float xMove, yMove;
protected TYPECREATURE typeCreature;
public enum TYPECREATURE {
        Player,
        Salad,
        Pepper,
        Egg,
}
```

Cette classe représente toutes les entités mobile (monstres et joueur), c'est dans cette classe que les mouvement de ces entités sont gérer : la méthode *move()* appelle les méthodes *moveX()* et *moveY()* pour le déplacement dans la map, et les méthodes *checkCollisionWithPlayer()* et *checkCollisionWithFood()* en fonction du type de créature.

Les méthodes sont détaillé et expliqué dans le code source et seront expliqué dans la partie détail des algorithmes.

-> Animation.java

Cette classe permet de changer la texture des monstres et du joueur toute les 250 mili secondes grâce à un timer. Cette classe prend en paramètre un tableau d'image, les images à afficher, et renvoie grâce à sa méthode *getCurrentImage()* l'image à affichée.

-> Monster.java

Comme les autres classes, cette classe possède les méthodes *tick()* et *render()*. La méthode tick() appelle la méthode chasePlayer() qui récupère la position du joueur, et en fonction de sa position par rapport au monstre, déplace le monstre vers le joueur.

-> Player.java

La classe qui gère le joueur et ses mouvement , sa méthode tick() est la même que celle du monstre à l'exception du déplacement du joueur : la méthode *getInput().* Cette méthode récupère les touches entrées par l'utilisateur par l'intermédiaire de la classe KeyManager. En fonction du code de la touche pressé, le joueur essaye de se déplacer.

La méthode hit() est appelé lorsqu'un monstre touche le joueur, le joueur devient alors inactif et tous les monstres sont immobilisés pendant 3 secondes. La vie du joueur décrémente ensuite de 1.

-> EntityManager.java

```
private Handler handler;
private Player player;
private ArrayList<Entity> entities;
private ArrayList<Food> foods;
private ArrayList<Creature> creatures;
private Food[][] foodManager;
```

Cette classe créer toutes les entités du jeu et les range dans des *ArrayList<>()* et fait appel à leurs méthodes *tick()* et *render()*. Afin de gérer les déplacements de la nourriture, j'ai créer un tableau de Food à deux dimension afin de stocker toutes les positions possibles de la nourriture. Chaque Food du tableau connait sa position dans le tableau grâce aux variables lineInArray et columnInArray.

La méthode FallFood est appelé lorsque les trois parties de la nourriture sont down, la méthode prends en paramètre la nourriture à faire tomber.

-> Utils.java

Cette classe permet d'effectuer des tâches, un peu longue et pas facile à codé, en les simplifiants. Ces fonctions sont accessibles à partir de n'importe quelle classe. Elles servent à lire et écrire dans un fichier, charger une image ou encore transformer une chaîne de caractères en nombre

IV. Descriptions des principaux algorithmes

1. L'Image Par Seconde :

Certains Ordinateurs fonctionnent plus vite que d'autres, il est donc nécessaire de mettre en place un système permettant de contenir la rapidité des enchainements d'image, et d'avoir un même nombre d'images par secondes pour tous les ordinateurs. Cet algorithme est difficile à assimiler.

```
int fps = 60;                    // Détermine le nombre d'images par secondes voulu
double timePerTick = 1000000000 / fps; // Transforme les secondes en nano secondes
                                       // pour plus de précisions
double delta = 0;
long now;
long lastTime = System.nanoTime(); // Récupère la temps en nanoSeconde Actuel
long timer = 0;
int ticks = 0;

while(running) {
        now = System.nanoTime(); // Récupère la temps en nanoSeconde Actuel
        delta +=(now-lastTime) / timePerTick;
// Différence entre le temps les deux dernières récupération du temps divisé par le nombre d'IPS, lorsque ce nombres sera égal ou supérieur à 1, cela voudrait dire que le nombre qu'il s'est passé assez de temps pour pouvoir rafraîchir les données et afficher le jeu afin qu'il y ai 60 images par secondes
        timer += now - lastTime;
        lastTime = now;
// Réinitialise le temps en nano secondes Actuel
        if(delta>=1) {
                tick();
                render();
                ticks++;
                delta--;
        }

        if(timer >=1000000000) {
                System.out.println("Images Par Secondes :" + ticks);
// Permet de vérifie si on obtiens bien 60 IPS
                ticks = 0;
                timer = 0;
        }
}
```

2. Mouvement des entités mobiles :

```
public void moveX(){
        if(xMove > 0){ // Déplacement à droite
                int tx = (int) (x + xMove + bounds.x + bounds.width) / Tile.WIDTH;
                // Récupère la position x du joueur
```

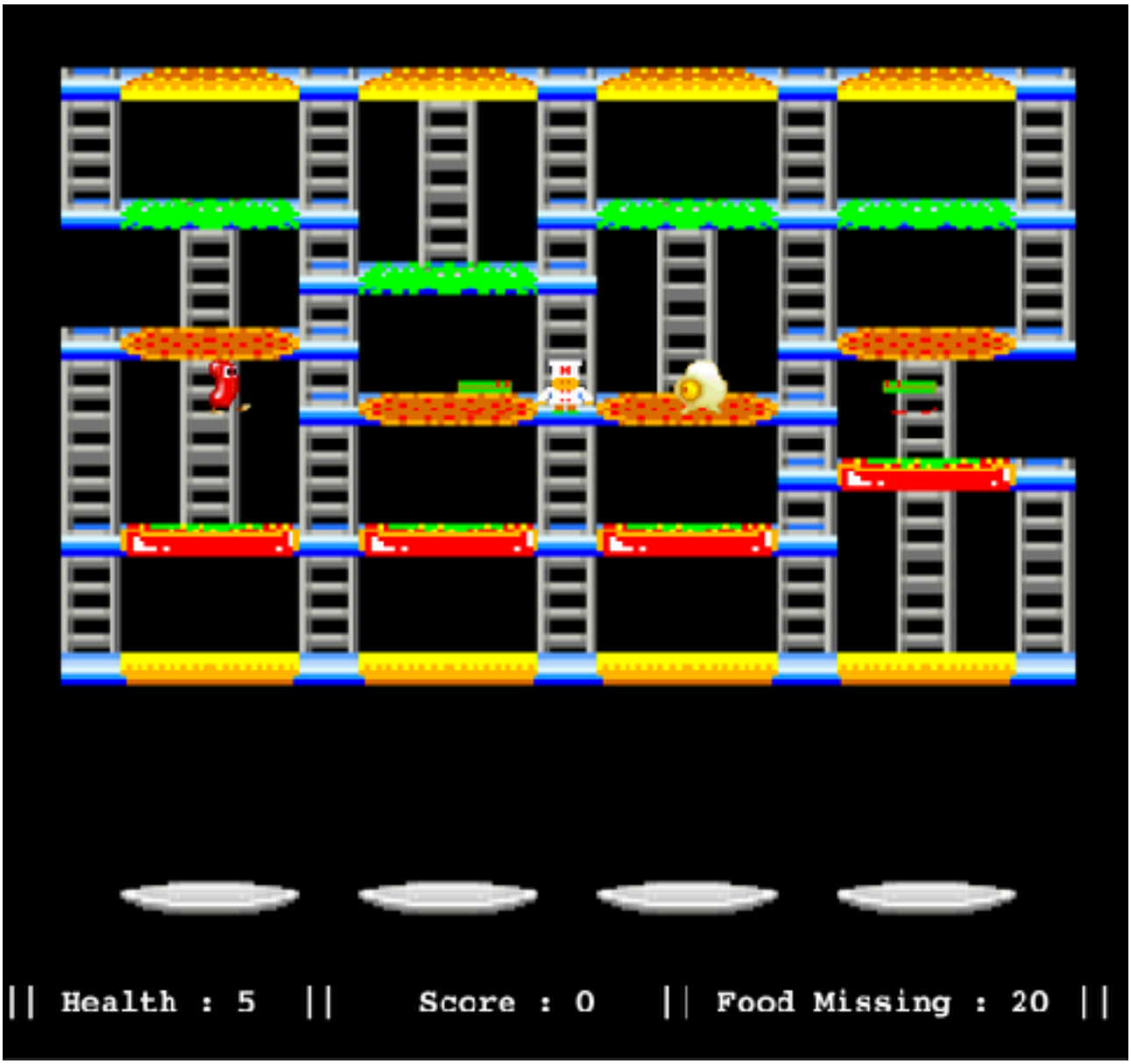
```
                if(!collisionWithTile(tx, (int) (y + bounds.y) / Tile.HEIGHT) &&
!collisionWithTile(tx, (int) (y + bounds.y + bounds.height) / Tile.HEIGHT)){
                        // Vérifie si la prochaine position rentre en collision avec une parcelle
solide
                        x += xMove;
                        // Si non, la créature se déplace

                }else if(xMove < 0){ // Déplacement à gauche
                        int tx = (int) (x + xMove + bounds.x) / Tile.WIDTH;
                        // Même raisonnement
                        if(!collisionWithTile(tx, (int) (y + bounds.y) / Tile.HEIGHT ) &&
                                        !collisionWithTile(tx, (int) (y + bounds.y +
bounds.height) / Tile.HEIGHT)){
                                x += xMove;
                        }else{
                        }
                }
        }
```

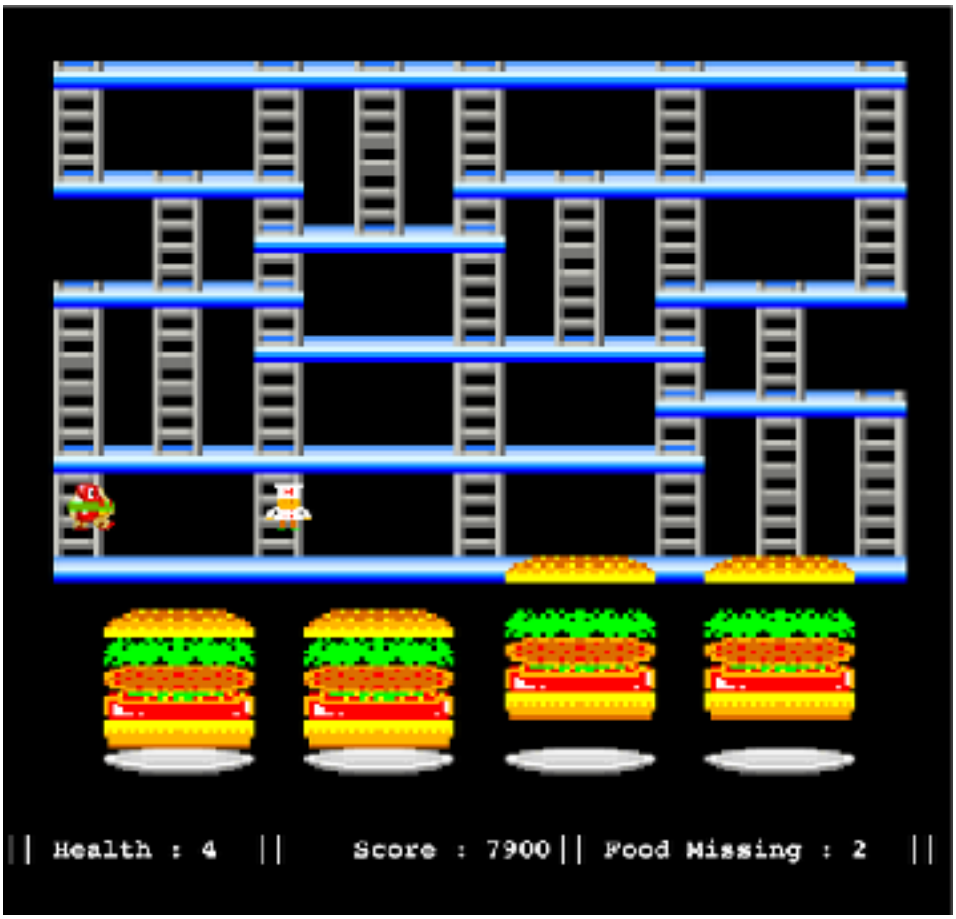
3. Fonctionnement de méthode FallFood(Food f, boolean cascade) :

```
int c = f.getColumnInArray();
int l = f.getLineInArray();

if((foodManager[l+1][c].getLeftPart().getTypeFood() != TYPEFOOD.Platform &&
f.getLeftPart().getTypeFood() != TYPEFOOD.None)) {

        System.out.println(f.getLeftPart().getTypeFood() +" falls at Line = " +
        f.getLineInArray() + " and Column = " + f.getColumnInArray());
        Food actualFood = f;
        Food nextFood = foodManager[l+1][c];
        if(!cascade)
                foodManager[l][c] = new Food(handler,
actualFood.getColumn(),actualFood.getLine(), TYPEFOOD.None, l, c);
        while(f.getLine() != nextFood.getLine()) {
                f.setLine(f.getLine() + 1);
        }
        f.setLineInArray(f.getLineInArray() + 1);
        foodManager[l+1][c] = f;
        f.setDown(false);
        if(l == 4)
                handler.getScore().foodPartMissed--;
        if(!cascade)
                handler.getScore().score+=100;
        handler.getScore().score+=50;
        f.tick();

        fallFood(nextFood,true);
}
```

- On récupère la position dans le tableau de la nourriture à faire tomber, on vérifie ensuite si cette nourriture n'est pas de type NONE et si la nourriture suivante c'est pas la plateforme de fin (dans quelle cas la nourriture aura trouver sa place final.
- On récupère la nourriture suivante pour donner a la nourriture actuel ses donnés dans le tableau.
- Si ce n'est pas une cascade on créer une nouvelle instance de nourriture de type NONE et on la place à celle de la nourriture qui doit tomber.
- On fait tomber la nourriture jusqu'a la position de la nourriture suivante
- Et on fait tomber la nourriture suivante avec comme paramètre true pour la cascade.

V. Captures d'écran :




|| Health : 4 ||
Score : 7900 ||
Food Missing : 2 ||


New High Score
YOU WIN THIS FABULOUS GAME !
Your score : 8600
Best score : 8600