BARSONI Dorian **IE4-06**

Projet Info4B : BURGER TIME

SOMMAIRE :

Introduction :

Le jeu Burger Time est un jeu vidéo créé en 1982. Il s'agit d'un jeu de plates-formes dans lequel on retrouve un cuisinier qui a pour objectif d'aplatir différentes parties de burgers afin de les constituer. Ce dernier a un nombre limité de vies et est menacé par des ennemies qui cherchent à l'attraper. Le jeu se termine lorsque le cuisinier a assemblé tous les burgers ou lorsque qu'il est arrivé à court de vie

Le but de ce projet est de recréer ce jeu vidéo et d'y implémenter une dimension réseau afin de pouvoir jouer en multijoueur, ainsi que de sauvegarder les meilleurs scores. Dans ce compte rendu je vais vous détailler les manières dont je m'y suis pris afin de créer ses différents aspects.

Nous commencerons tout d'abord par voir l'analyse du sujet, l'architecture de l'application, l'idée de base pour recréer le jeu ainsi que les communications entre les différentes parties. Nous verrons ensuite l'implémentation concrète au sein des packages créés. Un fois ceci fait nous verrons l'utilisation du programme par un test concret illustré par des captures d'écran du jeu. Enfin nous terminerons par une conclusion sur le résultat final obtenu ainsi que les éventuelles évolutions et améliorations qui pourraient être apportées.

Projet Info4B : BURGER TIME

I/Analyse du sujet :

Avant la réalisation du projet il a fallu réfléchir à une architecture adéquate qui permettrait une bonne communication entre les différentes parties du programme. L'idée a donc été de créer 3 grandes parties :

- La plus grosse partie ainsi que la plus importante, le Noyau : C'est la partie créant les différents éléments, les mettant à jour eux ainsi que leur affichage. Cette partie a pour but de créer les structures du jeu vidéo, dynamiques *(les différentes entités comme les burgers ou les joueurs)* ou statiques (comme la carte). C'est elle qui les contiendra et qui, lorsqu'elle en recevra la demande, les mettra à jour *(demande de déplacement par exemple)*. C'est la partie la plus importante du programme.
- L'affichage utilisateur et les saisie clavier, les Entrés / Sorites : Cette partie a pour but de permettre à l'utilisateur d'interagir avec le programme. Il peut réaliser des entrées clavier qui seront directement envoyées au Noyau. Ce dernier en fonction de l'entrée reçu et en fonction de l'état dans lequel se trouvera le programme enverra une réponse. Cette réponse sera reçue par l'utilisateur sous la forme d'un affichage écran *(interface graphique)*.
- Le serveur et les clients, la partie Communication Réseau : Enfin cette dernière partie sert, comme son nom l'indique, à gérer les communications réseaux entre les serveurs et les clients. Cette partie n'est évidemment utile que pour le multijoueur. De plus il est important de préciser que cette partie ne contient pas le serveur et le client, elle ne gère <u>que la communication</u> entre eux, le serveur et les clients étant présents dans le noyau.

Comme dit plus haut le programme aura différents « états ». Ces états seront stockés et mis à jour dans le Noyau. Par exemple l'un des « états », que nous aborderons plus tard, est l'état *titleScreen*. C'est le premier état qui apparait lors du lancement du programme.

Durant le déroulement entier du programme l'utilisateur peut saisir des entrées claviers grâce à la partie Entrées / Sorties. Cependant ces entrées claviers n'auront pas le même impact en fonction du l'état du programme. Par exemple dans l'état *titleScreen* les seules entrées prises en compte sont les 4 touches directionnelles ainsi que la touche entrer.

Il y a donc une double communication entre le Noyau et l'utilisateur. En effet l'utilisateur peut réaliser des entrées clavier, par exemple en appuyant sur la flèche directionnelle du bas. Cette information sera directement envoyée au Noyau. Un fois reçu le Noyau appliquera une modification à une élément en fonction de l'état dans lequel se situe le programme. Il renverra ensuite une réponse à l'utilisateur sous la forme d'une modification de l'affichage.

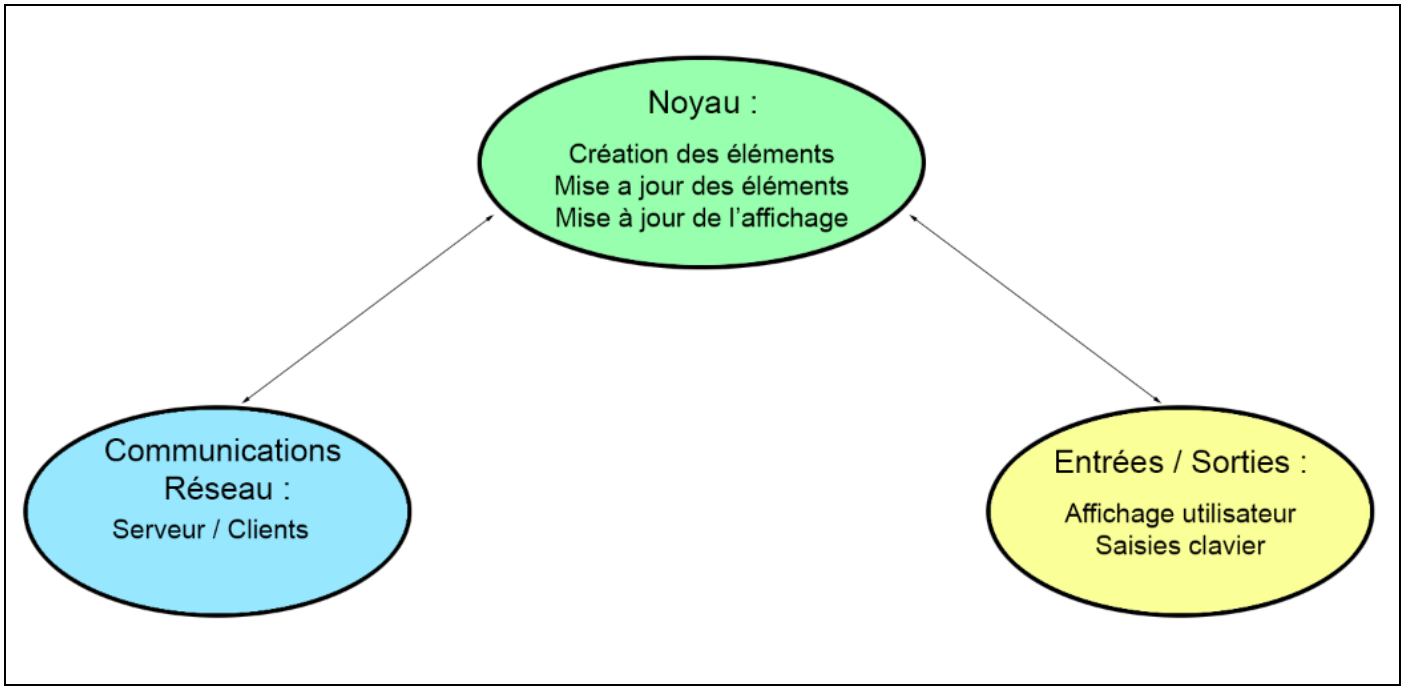
Dans notre exemple l'utilisateur a appuyé sur la flèche directionnelle du bas. Le noyau reçoit l'information. Il voit qu'il est dans l'état *titleScreen*. Cela implique le déplacement du bouton de sélection *(bouton servant à choisir son mode de jeu, de voir le tableau des scores ou encore de quitter le jeu)* vers le bas. Il met donc à jour la position du bouton qu'il a en mémoire, puis une fois cela fait il envoie la modification de l'affichage à l'utilisateur.

En ce qui concerne les parties Communication réseau et le Noyau on a aussi une double communication. L'idée est que le Noyau crée le serveur et les clients. Chaque personne ayant lancé le programme aura donc le choix : Soit héberger une partie, soit en rejoindre une.

La personne hébergeant la partie enverra donc au Noyau une demande de création de serveur et de client. Une fois le serveur créé un client lui sera automatiquement connecté, ce sera le cuisinier qui sera contrôlé par la personne hébergeant le serveur.

Pour les personnes souhaitant rejoindre une partie une demande à leurs Noyaux respectifs permettra de créer un client et de le connecter au serveur déjà existant. On aura une connexion maximale de 4 personnes : un cuisinier et 3 ennemis.

Le programme est donc décomposé en 3 parties majeurs : Le Noyau créant les éléments dynamiques et statiques du jeu, les mettant à jour et envoyant ces mises à jour à l'utilisateur et aux clients. Les Entrées / Sorties matérialisées par les entrées claviers de l'utilisateur et l'affichage du jeu sous forme d'une interface graphique. La Communication Réseau permettant à plusieurs joueurs de jouer sur une même partie.



Représentation des 3 grandes parties du programme ainsi que les communications entre elles

II/Implémentation :

Nous avons vu le fonctionnement global du programme dans la partie précédente, nous allons maintenant voir les fonctionnalités plus détaillées et la manière utilisée pour les implémenter. Pour ce faire nous allons voir les différents packages qui composent le programme, leur utilité, et le codes des parties les plus importantes.

Le programme est composé de 6 packages. Le package *Entity*, *Launching*, *Plateau*, *Position*, *Reseau* et *Window*. Chaque package est destiné à une fonctionnalité précise et est composé de plusieurs classes. Nous allons donc commencer par voir la composition du package *Launching*.

a) Launching :

Le package Launching est, comme son nom l'indique, le package principal lors du lancement du programme. Il contient les classes Main, Etat et Setup.

1) Main :

La classe Main est une classe très courte car ne possédant qu'une méthode *public static void main(String[] args)*. Le but de la classe Main est le lancement du programme, son déroulement lors d'une partie et lorsque l'utilisateur souhaite le fermer. La classe Main crée une instance de la classe Map *(classe vu plus tard)*, une de Fenetre *(classe vu plus tard)*, une de la classe Etat et une de la classe Setup puis entre dans une boucle infinie.

Cette boucle infinie utilise une instance de ScheduledExecutorService, permettant de créer une boucle se répétant pour une unité de temps fixée. Dans notre cas la boucle se répète toute les 3 millisecondes.

Dans cette boucle on vérifie en premier lieu si le cuisinier possède encore des vies ou si les burgers sont tous assemblés. Si l'une de ces conditions est vraie on ajoute 200 points au score par nombre de vies restantes puis on ferme le jeu. Si l'état du programme est « *inGame* » on appelle la fonction *updateInGame()* de la classe Fenetre *(mettant à jour les éléments en fonction des entrées claviers lors d'une partie)*. Sinon si l'état du programme est « *inGameMulti* » on appelle la fonction *updateInGmeMulti()* de la classe Fenetre *(mettant à jour les éléments en fonction des entrées claviers lors d'une partie multijoueur)*. Enfin sinon si l'état du programme est « *leaving* » on appelle *updateLeaving()* de la classe Fenetre *(fermant la fenêtre et arrêtant le programme)*.

```
//Boucle mettant à jour les évènements se déroulant sur la fenêtre avec un temps de 3ms entre chaque éxectution
ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
executor.scheduleAtFixedRate(() -> {
    if(etat.getVies() == 0 || map.isOver() == 3484) { //Si on tombe à court de vie ou que les burgers sont assemblés
        etat.addScore(etat.getVies()*200); //On ajoute 200pts au socre par vies restantes
        etat.setEtatLeaving(); //On passe l'état du programme à leaving
        System.out.println(etat.getScore()); //On affiche le score dans la console
        setup.end = true; //On passe l'attribut end de setup à true
    }

    if(etat.getEtat().equals("inGame")) { //Si l'état du programme est inGame
        fenetre.updateInGame(); //On appelle la fonction updateInGame() de la fenêtre
    }

    else if(etat.getEtat().equals("inGameMulti")) { //Sinon si l'état du programme inGameMulti
        fenetre.updateInGameMulti(); //On appelle la fonction updateInGameMulti()
    }

    else if(etat.getEtat().equals("leaving")) fenetre.updateLeaving(); //Sinon si l'état est leaving
                                                                       //On appelle updateLeaving()
}, 0, 3, TimeUnit.MILLISECONDS); //On fixe le temps de répétition à 3 millisecondes
```

Boucle de la classe Main permettant le fonctionnement du programme en jeu ou son arrêt

2) Etat :

La classe Etat est la classe permettant au programme de savoir dans quel état il se trouve. Plusieurs états existent : L'état initial (dont on a déjà parlé) *titleScreen* qui signifie qu'on est à l'écran titre, l'état *titleScreenSoloMulti* qui signifie qu'on est à l'écran titre solo/multijoueur, l'état *HebergerRejoindre* qui signifie qu'on est à l'écran titre héberger / rejoindre, *WaitingRoom* qui signifie qu'on est dans la salle d'attente pour lancer une partie multijoueur, *inGame* qui signifie qu'on dans une partie solo, *inGameMulti* qui signifie qu'on est dans une partie multijoueur et leaving qui signifie qu'on souhaite quitter le programme.

Ces états sont stockés sous la forme d'un String représenté par l'attribut etat de la classe Etat. Il est modifié à l'aide de fonctions *public void setEtatTitleScreen()* qui elle, par exemple, fixe l'état *titleScreen* au programme.

La classe Etat possède également deux attributs int vies et score, pour stocker respectivement la vie et le score. On possède les méthodes *getVies()* et *getScore()* pour obtenir leurs valeurs, ainsi que *retireVie()* et *addScore(int score)* pour enlever une vie ou ajouter des points au score.

Enfin la classe Etat possède également un attribut Setup setup. Ce dernier est le même que celui créé dans Main et qui lui est passé dans son constructeur. Grâce à setup on peut, à chaque fois qu'on retire une vie ou qu'on rajoute du score, mettre à jour sur l'interface graphique la modification.

```
//Fonction passant l'état à leaving
    public void setEtatLeaving() {
        etat = "leaving";
    }

//Fonction retirant une vie
public void retireVie() {
    vies--;
    setup.updateVies(vies);
}

//Fonction augmentant le score
public void addScore(int points){
    score += points;
    //System.out.println("Score : " + score);
    setup.updateScore(score);
}
```

Quelques fonctions de la classe Etat

3) Setup :

La classe Setup est une des classes les plus importante du programme. C'est elle qui crée les éléments dynamiques du jeu, à savoir les différentes parties des burgers, le cuisinier et les ennemies. Elle contient également un bon nombre des JLabel permettant de mettre à jour les évènements à l'écran.

On y retrouve donc beaucoup d'attributs dont nous parlerons lorsque nous étudierons les packages associés, comme Pain_inferieur, Cuisinier ou encore Oeuf. Mais on y retrouve également le JLabel vies, affichant le nombre de vies et le JLabel[] score affichant le score. La classe Setup possède aussi un tableau Client[] clients qui stocke les différents clients connectés lors d'une partie multijoueur.

La majorité des méthodes présentent dans cette classe consiste à mettre à jour les évènement à l'écran, comme la méthode *setupTitleScreen()*, qui affiche l'écran titre initial avec le bouton de sélection, ou encore *removeTitleScreen()*, qui supprime l'écran titre et le bouton de la fenêtre.

```
//Fonction mettant en place l'écran titre
public void setupTitleScreen() {
    fenetre.fenSelectButtonSetup(535, 474); //Ajout du bouton de sélection
    fenetre.fenTitleSetup(); //Ajout du fond
    fenetre.setVisible(true); //Visibilité de la fenêtre à true
    fenetre.repaint(); //Mise à jour des labels (inutile au lancement mais utile en cas de retour en arrière
}

//Fonction supprimant les éléments de l'écran titre
public void removeTitleScreen() {
    fenetre.fenSelectButtonRemove(); //Suppression du bouton de sélection
    fenetre.fenTitleRemove(); //Suppression
}
```

Fonction mettant en place l'écran titre et l'enlevant

On peut cependant trouver d'autres méthodes, comme *updateScore(int score)*, qui permettent de mettre à jour l'affichage du score sur la fenêtre. Pour cela on décompose l'entier score passé en paramètre en 5 entiers, puis à l'aide d'un switch on affiche le bon chiffre au bon endroit.

```java
//Fonction mettantà jour le score
public void updateScore(int score) {
    int[] chScore = new int[5];

    //Pour i allant de 4 à 0
    for(int i=4; i>-1; i--) {
        chScore[i] = score%10; //chScore[i] prend la valeur du score modulo 10
        score /= 10; // On divise le score par 10

        //On prend l'entire chScore[i]
        switch(chScore[i]) {
            case(0) : //Si il vaut 0
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/0.png"))); //On charge 0.png
                break;
            case(1) : //Si il vaut 1
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/1.png"))); //On charge 1.png
                break;
            case(2) : //Si il vaut 2
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/2.png"))); //On charge 2.png
                break;
            case(3) : //Si il vaut 3
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/3.png"))); //On charge 3.png
                break;
            case(4) : //Si il vaut 4
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/4.png"))); //On charge 4.png
                break;
            case(5) : //Si il vaut 5
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/5.png"))); //On charge 5.png
                break;
            case(6) : //Si il vaut 6
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/6.png"))); //On charge 6.png
                break;
            case(7) : //Si il vaut 7
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/7.png"))); //On charge 7.png
                break;
            case(8) : //Si il vaut 8
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/8.png"))); //On charge 8.png
                break;
            case(9) : //Si il vaut 9
                this.score[i].setIcon(new ImageIcon(getClass().getResource("/images/chiffres/9.png"))); //On charge 9.png
                break;
        }
    }
}
```

Fonction de mise à jour du score

Enfin dans les fonctions purement multijoueur on peut également trouver des fonctions comme *creationServeur()* qui crée un serveur, *creationClient()* qui crée un client, *getMyPosX(int id) getMyPosY(int id)* qui retournent respectivement la position x et y du joueur, ou encore *updateEntities(int idRecu, int x, int y)* qui met à jour la position d'une entité en fonction de l'id qu'on reçoit et des coordonnées x et y.

```java
public void updateMyEntities(int idRecu, int x, int y) {
    switch(idRecu) { //En fonction de l'id reçu
        case(0) :
            cuisinier.setPixelPos(x, y);
            break; //Mise à jour des pos du cuisinier
        case(1) :
            oeuf.setPixelPos(x, y);
            break; //Mise a jour des pos de l'oeuf
        case(2) :
            piment.setPixelPos(x, y);
            break; //Mise a jour des pos du piment
        case(3) :
            saucisse.setPixelPos(x, y);
            break; //Mise à jour des pos de la saucisse
    }
}
```

Fonction mettant à jour la position d'une entité en fonction de l'id et des positions reçues

La classe setup est donc très importante, que ce soit pour l'affichage de l'interface graphique, la création d'un serveur ou d'un client ou bien la mise à jour de chaque client sur chaque fenêtre.

b) Plateau :

Plateau est le package contenant les structures statiques du programme, c'est-à-dire la carte du jeu. Elle est composée des classes Blocs, dont Mur, Vide, Echelle et Border héritent, ainsi que de Map.

1) Blocs :

Blocs est une classe toute simple. Elle contient 3 méthode booléenne abstraites caractérisant un bloc : *estAccessible()* pour savoir si on peut traverser le bloc, *estEscaladable()* pour savoir si on peut monter sur le bloc, *estVide()* utile pour détecter une chute *(nous verrons cela plus tard)*. Les classes Mur, Vide, Echelle et Border héritent de Blocs pour définir leurs propres caractéristiques. Par exemple Mur n'est ni accessible, ni escaladable, ni vide.

```
package Plateau;

public class Mur extends Blocs{

    //Bloc empechant le joueur de passer ou  de tomber

    @Override
    public boolean estAccessible() {
        return false;
    }

    @Override
    public boolean estEscaladable() {
        return false;
    }

    @Override
    public boolean estVide() {
        return false;
    }
}
```

Classe Mur

2) Map :

La classe Map est la classe la plus important du *Plateau*. Elle a pour but de créer la carte du jeu. Elle possède dont un String définissant la disposition de la carte. Un X est un mur, un . est du vide, un E une échelle et un B une bordure. Dans le constructeur de Map ce String et positionné dans un tableau de char, puis à l'aide d'un switch on crée la grille de jeu qui se résume à un tableau à double entré de Blocs. Si le caractère lu est un X on crée un mur, ect…

```
public Map() {

    //Tableau contenant le modele
    char[] tab_modele = modele.toCharArray();

    //On fixe la hauteur de chute des aliments
    limitedHeight = new int[4];
    for(int i=0; i<4; i++) {
        limitedHeight[i] = 931;
    }

    grille = new Blocs[this.largeur][this.longueur];

    //Initialisation de la grille
    for(int lig=0; lig<largeur; lig++) {
        for(int col=0; col<longueur; col++) {
            char c = tab_modele[lig*longueur + col];
            switch(c) {
                case 'X' :
                    grille[lig][col] = new Mur();
                    break;
                case '.' :
                    grille[lig][col] = new Vide();
                    break;
                case 'E' :
                    grille[lig][col] = new Echelle();
                    break;
                case 'B' :
                    grille[lig][col] = new Border();
                    break;
                default :
                    System.out.println("Bloc inexistant");
                    break;
            }
        }
    }

}
```

Constructeur de Map

On retrouve également dans Map des fonctions permettant d'obtenir le bloc de la grille en fonction d'une Point *(classe abordée plus tard)* donné. Mais on y retrouve aussi l'attribut *int[] limitedHeight*. Celui-ci est d'une importance majeure. Il permet de bloquer la chute des aliments à la hauteur fixée. C'est un tableau de 4 int car on possède 4 burgers. A chaque fois qu'un aliment tombe dans le carré, l'assemblant donc, la colonne i où il se situe change de valeur de hauteur limite *(on la remonte de 15 pixels)*.

La fonction isOver() permet de sommer les 4 valeurs de limitedHeight. Avec les valeurs initiales données on sait que les 4 burgers sont assemblés (donc que la partie est terminée) si la somme vaut 3484, d'où la condition dans la boucle du Main.

```java
//Fonction permettant d'obtenir la hauteur maximum de chute d'une colonne d'aliment
public int getLimitedHeight(int i) {
    return limitedHeight[i];
}

//Fonction arrêtant le jeu une fois tous les burgers empilés
public int isOver() {
    return limitedHeight[0] + limitedHeight[1] + limitedHeight[2] + limitedHeight[3];
}

//Fonction permettant de modifier la hauteur de chute maximale d'une colonne d'aliments
public void setLimitedHeight(int nb, int i) {
    limitedHeight[i] = nb;
    etat.addScore(200); //On ajoute 200pts au score quand un aliment tombe dans un rectangle
}
```

Fonction de Map donnant la hauteur limite, vérifiant si les burgers sont assemblés, mettant à jour la hauteur limite

c) *Entity :*

Entity est un package très riche, nous ne regarderons donc pas en détaille toutes les classes. Il possède au total 12 classes. Chacune hérite de la classe initiale Entitee.

1) Entitee :

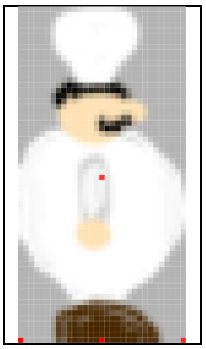
Entitee est très simple, elle possède 3 attributs protégés, Map *map,* Fenetre *fen (classe abordée plus tard),* Point *pos (classe abordée plus tard)*. Ces trois éléments sont la base de ce que représente une entité.

Par la suite les entités se scindent en 2 : Les entités Joueur et les entités Aliment, ce sont les 2 classes principales du package. Comme leurs noms l'indique, on va séparer les joueurs *(utilisateur ou IA)* des composants des burgers.

2) Joueur :

Deux classes héritent de Joueur : Cuisinier, et Ennemi. On va s'appuyer sur Cuisinier pour donner quelques exemples.

Chaque joueur possède 9 attributs protégés : 4 Pixels (classe vu plus tard), 1 JLabel *sprite*, 2 String contenant le nom du sprite gauche et le nom du sprite droit, et 2 int *x* et *y* représentant la position de l'entité.



Voici le sprite droit du cuisinier. Sur ce dernier on peut remarquer la présente de 4 pixels rouges. Ce sont les 4 attribut Pixel. Sans rentrer dans le détail la classe Pixel représente les coordonnées d'un pixel. On a donc le pixel hg *(pour horizontal gauche)* qui est le pixel le plus à gauche, le pixel hd *(pour horizontal droit)* qui est le pixel le plus à droite, le pixel vh *(pour vertical haut)* qui est le pixel le plus en haut, et le pixel vb *(pour vertical bas)* qui est le pixel bas central. Notons que le pixel vertical haut n'est pas au niveau de la tête des entités. En effet ces dernières peuvent passer n'importe où.

Les joueurs possèdent beaucoup de méthodes. On y retrouve des getter pour obtenir les Pixel, mais également des méthodes de demande de déplacement et de déplacement. L'idée étant que lorsqu'un joueur veut se déplacer à gauche par exemple on appelle la méthode *deplacementGauche()*. Cette dernière n'a aucun effet si le déplacement n'est pas permis, ce qui est vérifié par la méthode *demandeDeplacementGauche()*.

Les méthodes de demande déplacent le pixel concerné d'un pixel dans la direction ou l'on souhaite se déplacer. On regarde sur quel bloc se situe le pixel. Si le bloc est *accessible (ou escaladable s'il s'agit d'une descente ou d'une montée)* alors on effectue le déplacement. Effectuer le déplacement consiste à déplacer les 4 pixels du joueur de 1 dans la direction demandée, de modifier la position *x* et *y* du sprite *(pixel en haut à gauche)* puis de le mettre à jour.

```
//Fonction vérifiant si un déplacement en haut est possible
public boolean demandeDeplacementHaut() {
    Pixel pix =  new Pixel( pix_vh.getX(), pix_vh.getY() - 1 ); //Création d'un pixel dont le pixel vertical haut est un pixel plus haut que le pixel passé en argumen
    Point p = pix.fromPixelToPoint(); //Convertion des coordonnées pixel au coordonnées de la grille
    if(map.getBloc(p).estEscaladable()) return true; //Si le bloc est escaladable en retourne true
    else return false; //Sinon on retourne false
}

//Fonction effectuant le déplacement en haut
public void deplacementHaut() {
    if(demandeDeplacementHaut()) {
        pix_hg.setY(pix_hg.getY() - 1); //On décrémente
        pix_hd.setY(pix_hd.getY() - 1); //Tous les pixels
        pix_vh.setY(pix_vh.getY() - 1); //De l'entitée
        pix_vb.setY(pix_vb.getY() - 1); //De 1

        y = getVb().getY() - 61; //On met à jour la position y de l'entitée
        sprite.setLocation(x, y); //On replace l'image de l'entitée
    }
}
```

Fonction demandeDeplacementHaut() et deplacementHaut()

Pour finir on trouve d'autre méthode assez explicites comme *resetPos()* retournant un joueur à sa position initial. Deux dernières fonctions importantes à aborder sont *detectionChute()* et *updateJoueur()*. detectionChute() permet de détecter si le pixel vertical bas se situe dans un bloc vide. Si oui il retourne true, sinon il retourne false.

updateJoueur() est la fonction mettant à jour le joueur. À l'aide de toucheAppuyee *(hashMap de la classe Fenetre)* on sait sur quelle touche le joueur a appuyé. On peut donc grâce à un switch effectuer le mouvement adéquat.

```java
//Fonction détectant si l'entitée chute
public boolean detectionChute() {
    Pixel pix =  new Pixel( pix_vb.getX(), pix_vb.getY() + 1 ); //Création d'un pixel de 1 pixel plus bas que le pixel vertical ba
    Point p = pix.fromPixelToPoint(); //Convertion des coordonnées pixel au coordonnées de la grille
    if(map.getBloc(p).estVide()) { //Si le bloc est vide
        return true; //On renvoie true
    }
    return false; //Sinon on renvoie faux
}

//Fonction mettant à jour les déplacement de l'entitée
public void updateJoueur() {
    if(fen.toucheAppuyee.get(KeyEvent.VK_LEFT)) { //Si on presse la flèche gauche
        deplacementGauche(); //On se déplace à gauche
    }
    if (fen.toucheAppuyee.get(KeyEvent.VK_RIGHT)) { //Si on presse la flèche droite
        deplacementDroit(); //On se déplace à droite
    }
    if (fen.toucheAppuyee.get(KeyEvent.VK_UP)) { //Si on presse la flèche haut
        deplacementHaut(); //On se déplace en haut
    }
    if (fen.toucheAppuyee.get(KeyEvent.VK_DOWN)) { //Si on presse la flèche bas
        deplacementBas(); //On se déplace en bas
    }
    if(detectionChute()) { //Si on détecte une chute
        deplacementBas(); //On se déplace en bas
    }
}
```

Fonctions detectionChute() et updateJoueur()

Concernant la classe Ennemi on retrouve des méthodes supplémentaires à Joueur (Ennemi hérite de Joueur), notamment tout ce qui est lié à l'IA. Des fonction comme *cheminLePlusCourt(Point Depart, Point Arrive)* permettent de calculer le chemin le plus court (algorithme de Dijkstra détaillé lorsque l'on parlera de la classe Dijkstra). La fonction *deplacement(Point p)* permet à l'ennemi de se déplacer sur un bloc adjacent (retourné par cheminLePlusCourt()), et la fonction updateEnnemi() *(appelée que si l'ennemi est une IA)* permet le déplacement continu de l'ennemi.

```java
public void updateEnnemi(){
    attente++; //Attribut permettant de ralentir le déplacement ennemi
    if(attente%2 == 0) {
        Point posCuisinier = cuisinier.getVb().fromPixelToPoint(); //On prend les pos du cuisinier
        Point posCuisinierR = new Point(posCuisinier.getY(), posCuisinier.getX()); //On inverse le x et le y

        Point posEnnemi = this.getVb().fromPixelToPoint(); //On prend les pos de l'ennemi
        Point posEnnemiR = new Point(posEnnemi.getY(), posEnnemi.getX()); //On inverse le x et le y

        Point deplacement = cheminLePlusCourt(posEnnemiR, posCuisinierR); //On obtient le point de deplacement
        deplacement(deplacement); //On se deplace
    }

    if(detectionChute()) deplacementBas(); //On détecte les chutes
}

public void addCuisinier(Cuisinier cuisinier) {
    this.cuisinier = cuisinier;
}

//Fonction permettant le déplacement sur un point adjacent
public void deplacement(Point p) {

    int dx = p.getY() - this.getVb().fromPixelToPoint().getX(); //dx le différenciel x
    int dy = p.getX() - this.getVb().fromPixelToPoint().getY(); //dy le différenciel y

    if(dy > 0) { //Si dy est positif
        this.deplacementBas(); //On se déplace vers le bas
    }
    else if(dy < 0) { //Sinon s'il est négatif
        this.deplacementHaut(); //On se déplace vers le haut
    }
    else if (dx > 0) { //Sinon si dx est positif
        this.deplacementDroit(); //On se déplace vers la droite
    }
    else if (dx < 0) { //Sinon s'il est négatif
        this.deplacementGauche(); //On se déplace vers la gauche
    }
}
```

Fonctions updateEnnemi() et deplacement(Point p)

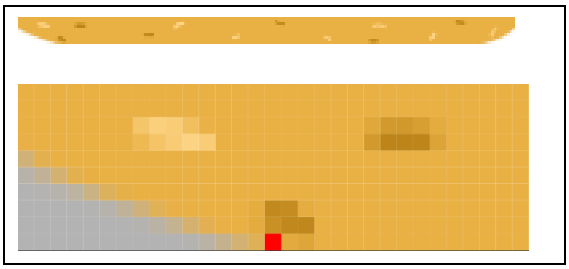
On ne détaille pas les classes Cuisinier, Oeuf, Piment et Saucisse car elles ne possèdent qu'un constructeur pour initialiser les joueurs et pour les afficher dans la fenêtre.

3) Aliments :

Les aliments, qui sont également des entités, sont gérés de manière bien différente des joueurs. Les classes héritant d'Aliment sont Pain_inferieur, Steak, Salade et Pain_superieur. Elles sont toutes similaires et ne seront, a l'instar de Cuisinier, Oeuf, Piment et Saucisse, pas détaillées car peu intéressantes.

Les aliments possèdent de nombreux attributs. Tout d'abord un JLabel[] *Aliment* qui stocke pour un même aliment 6 images (les aliments sont découpés en 6 parties). Un Pixel[] *tabPixel* qui stocke un pixel de chaque partie. Un Joueur *joueur* (qui devrait être un cuisinier mais cela n'influe pas sur le fonctionnement du jeu).

Si l'on se concentre sur ces premiers attributs on peut alors comprendre la structure d'un aliment :



Chaque aliment est décomposé en 6 parties, ici on prend l'exemple de la tranche de pain inférieur. Sur chaque partie on a un pixel central bas qui sert de « hit box ». On s'en servir lorsque l'aliment doit être aplatit. On retrouve également des tableau int[] *x* et *y* contenant chacun la position de la partie de l'aliment en attribut.

On a plusieurs attributs concernant l'aplatissement. Le premier est un int[] applatit *(Je m'excuse pour la faute faite à « aplatit », je ne m'en suis rendu compte bien après avoir utilisé partout dans le programme)*. Ce tableau de taille 6 ne stocke que des 0 et des 1.

On l'utilise dans la méthode *updateAliment().* Si le pixel vb du cuisinier est sur le pixel d'une des parties d'un aliment et si la case correspondante de *applatit* vaut 0, alors on la passe à 1 et on incrémente un autre attribut, *verif*. On déplace alors de 7 pixels vers le bas la partie de l'aliment. Un fois que *verif* vaut 6 on applique *chute()* pour faire tomber l'aliment. L'aliment chute tant que le pixel de sa première partie n'est pas dans le vide posé sur un mur, ou tant qu'il n'est pas à la hauteur limite. Si l'aliment atteint la hauteur limite on passe son attribut booléen *priventFall* de false à true qui empêche toute chute. *(Étant donné que la fonction chute() est un peu longue elle ne se situe pas en dans ce compte rendu, vous la trouverez cependant commentée dans la classe Aliment du package Entity)*

```
//Fonction mettant à jour l'aliment sur la fenêtre
public void updateAliment() {

    //Pour chaque label
    for(int i=0; i<6; i++) {
        //Si le joueur marche sur le pixel d'un label
        if( tabPixel[i].isEqual( joueur.getVb() ) ) {
            //Si le label n'est pas applatit
            if(applatit[i] == 0) {
                applatit[i] = 1; //On l'applatit
                verif++; //On incrémente vérif
                tabPixel[i].setY(tabPixel[i].getY() + 7); //On modifie la position du pixel
                y[i] = y[i] + 7; //On modifie la position du label
                Aliment[i].setLocation(x[i], y[i]); //On met le label à jour
            }
        }
        if(verif == 6) chute(); //Si verif vaut si on fait chuter l'aliment
    }
}
```

Fonction updateAliment()

Comment dit au début nous ne verrons pas le détail des classes Pain_inferieur, Steak, Salade et Pain_superieur. Elles ne contiennent que l'initialisation de leurs attributs et leur ajout à la fenêtre.

d) *Window :*

Le package ne contient qu'une seule classe : La classe Fenetre. Cette dernière a évidemment pour but de créer une fenêtre, mais pas seulement. En effet la classe, qui hérite de JFrame, implémente également KeyListener pour permettre à la fenêtre de capturer les entrées claviers de l'utilisateur.

Dans la classe Fenetre on retrouve beaucoup d'attribut, que ce soit du simple JLabel sauvegardant un fond, des entiers pour connaitre la position du bouton de selection, ou même de tableaux Ennemi[] *ennemi* ou Aliment[] *aliment* répertoriant les différents ennemis ou aliments pour pouvoir les mettre à jour. Car en effet c'est dans Fenetre qu'on retrouve les fonctions *updateInGame()* et *updateInGameMulti()* vu dans la classe Main.

Avant de parler d'elles nous nous intéressons aux autres fonctions présentes. Il y a de nombreuses fonctions comme *fenTitleSetup()* ou *fenTitleRemove()* qui ont pour but d'ajouter ou d'enlever un fond.

```
//Fonction ajoutant le fond de l'écran titre
public void fenTitleSetup() {
    add(fondTitleScreen);
}

//Fonction retirant le fond du jeu
public void fenTitleRemove() {
    remove(fondTitleScreen);
}

//Fonction ajoutant le bouton de sélection à l'écran
public void fenSelectButtonSetup(int x, int y) {
    xSelectButton = x; //Mise a jour de sa coordonnée x
    ySelectButton = y; //Mise à jour de sa coordonnée y
    SelectButton.setBounds(xSelectButton, ySelectButton, 850, 15); //Positionnement du bouton
    add(SelectButton); //Ajout
}

//Fonction supprimant le boutton de sélection
public void fenSelectButtonRemove() {
    remove(SelectButton);
}
```

Fonction fenTitleSetup(), fenTitleRemove(), de même que pour le bouton de sélection

On retrouve aussi les fonctions override de keyEvent *KeyTyped(KeyEvent e), keyPressed(KeyEvent e), keyReleased(KeyEvent e)*. Ici nous n'utilisons que keyPressed et keyReleased. Comme décrit lors de la partie **I/Analyse**, les touches ont des effets différents en fonction de l'état du jeu. On a donc dans keyPressed une succession de conditions permettant d'obtenir le résultat souhaité.

```
//Detection lorsqu'une touche est pressée
@Override
public void keyPressed(KeyEvent e) {

    if(state.getEtat().equals("titleScreen")) updateTitleScreen(e); //Si l'état est titleScreen on appelle updateTitleScreen en lui passant la touche
    else if(state.getEtat().equals("titleScreenSoloMulti")) updateTitleScreenSoloMulti(e); //Si l'état est titleScreenSoloMulti on appelle updateTitleScreenSoloMulti en lui passant la touch
    else if (state.getEtat().equals("HebergerRejoindre")) updateHebergerRejoindre(e);// Si l'état est HebergerRejoindre on appelle updateHebergerRejoindre(e)
    else if(state.getEtat().equals("WaitingRoom")) updateWaitingRoom(e); //Si l'état est WaitingRoom on appelle updateWaitingRoom(e)
    else if(state.getEtat().equals("inGame") || state.getEtat().equals("inGameMulti")) toucheAppuyee.put(e.getKeyCode(), true); //Si l'état est inGame ou inGameMulti on passe à true
                                                                                                                                //la touche directionnelle pressée dans le hashMap
}

//Detection lorsqu'une touche est relachée
@Override
public void keyReleased(KeyEvent e) {
    if(state.getEtat().equals("inGame") || state.getEtat().equals("inGameMulti")) toucheAppuyee.put(e.getKeyCode(), false); //Si l'état est inGame ou inGameMulti on passe à false
                                                                                                                            //la touche directionnelle pressée dans le hashMap
}
```

Fonctions keyPressed() et keyReleased()

On a des cas similaires pour tous les états saufs pour *inGame* et *inGameMulti*. Dans ces 2 cas on se contente de passer à true le HashMap *toucheAppuyee<Integer,Boolean>* qui stocke les 4 touches directionnelles. Une fois la touche relâchée on le repasse à false. Cela permet le déplacement des joueurs en temps réel car sans cela il y a un léger temps d'attente entre le moment où l'on appuie *(qui produit un déplacement)* et le moment ou le joueur avance en continu.

En ce qui concerne les autres états les fonctions sont assez longues, je n'en ferais donc qu'une brève description mais elles sont commentées dans le projet. On récupère la touche pressée, on la passe dans un switch pour choisir l'action à effectuer. Cela permet d'utiliser les touches directionnelles haut et bas pour déplacer le bouton de sélection, puis les touches entrer pour valider la sélection et la touche échap pour revenir en arrière. Grâce à un autre 2 conditions et aux coordonnées du bouton de sélection *(qui sont des attributs)* on peut savoir lorsque le bouton sort des options à sélectionner et le remettre dedans.

Cependant les méthodes les plus importantes restes les méthodes *updateInGame()* et *updateInGameMulti()*. Ces méthodes sont appelées toutes les 3 millisecondes dans la boucle du Main. Elle mettent à jour les joueurs *(à l'aide de la méthode updateJoueur() dans la classe Joueur),* ainsi que les aliments *(à l'aide des méthodes updateAliment() et checkIfAlimentUp() qui applatit l'aliment si un autre aliment lui tombe dessus)* et les ennemis *(grâce aux méthodes updateEnnemi et checkIfContact() de la classe Ennemi).*

```
//Fonction update mettant à jour les événements
public void updateInGame() {

    //Évènements du joueur
    joueur.updateJoueur();

    //Évènement des aliments
    for(int i=0; i<taille_aliment; i++) {
        aliment[i].updateAliment();
    }

    //Check si un aliment tombe sur un autre
    for(int i=0; i<(taille_aliment - 1); i++) {
        aliment[i].checkIfAlimentUp(aliment[i+1].tabPixel[0]);
    }

    //Mise à jour des ennemis
    for(int i=0; i<taille_ennemi; i++) {
        ennemi[i].updateEnnemi();

        if(ennemi[i].checkIfContact()) { //Si un ennemi touche le joueur
            for(int j=0; j<3; j++) {
                ennemi[j].resetPos(); //On remet les ennemis à leurs
            }                         //Places initiales
            joueur.resetPos(); //On remet le cuisinier à sa place initiale
            state.retireVie(); //On retire une vie
        }
    }
}
```

Méthode updateInGame()

La méthode *updateInGameMulti()* est légèrement différente. On utilise l'id du joueur stocké dans setup et donné par le serveur lors d'une partie multijoueur. En fonction de ce dernier on appelle *updateJoueur()* sur le cuisinier, l'œuf, le piment ou la saucisse.

```
public void updateInGameMulti() {

    int id = setup.clientID; //On récupère l'id du joueur

    switch(id) { //On le prend
        case(0) :
            joueur.updateJoueur(); //Si il vaut 0 on appelle updateJoueur sur joueur
            break;
        default :
            ennemi[id-1].updateJoueur(); //Sinon on l'appelle sur ennemi[id-1]
            break;
    }


    //Évènement des aliments
    for(int i=0; i<taille_aliment; i++) {
        aliment[i].updateAliment();
    }

    //Check si un aliment tombe sur un autre
    for(int i=0; i<(taille_aliment - 1); i++) {
        aliment[i].checkIfAlimentUp(aliment[i+1].tabPixel[0]);
    }

    //Mise à jour des ennemis
    for(int i=0; i<taille_ennemi; i++) {

        if(ennemi[i].checkIfContact()) { //Si un ennemi touche le joueur
            for(int j=0; j<3; j++) {
                ennemi[j].resetPos();//On remet les ennemis à leurs
            }                        //Places initiales
            joueur.resetPos(); //On remet le cuisinier à sa place initiale
            state.retireVie(); //On retire une vie
        }
    }
}
```

Fonction updateInGame()

e) Position :

Le package *Position* est une petit package. Il contient les classes Point, Pixel, Dijkstra et Pile. Ces 4 classes ne sont utilisées que pour manipuler des positions.

1) Point et Pixel :

Dans le projet il est important de différencier les coordonnées dites « Point » des coordonnées dites « Pixel ». La grille est composée de 61 blocs de long par 33 blocs de large. Ainsi on peut donc donner des coordonnées à chacun de ces blocs : Ce sont les coordonnées « Point ».

Les coordonnées Pixel quant à elles sont les coordonnées des pixels. Ainsi à un bloc correspond à 31x31 pixel (car un bloc fait 31 pixels par 31), mais à 1 pixel correspond un seul bloc.

Les classes Point et Pixel sont assez similaires, on y retrouve 2 attributs privé int *x* et *y*, des getter et setter, une fonction *isEquals(Point/Pixel p)* pour savoir sir 2 points ou 2 pixels sont égaux, et (seulement dans la classe Pixel) une fonction *fromPixelToPoint()* qui à partir des coordonnées d'un pixel en déduis les coordonnées Point.

```java
public boolean isEqual(Pixel pix) {
    if( (this.getX() == pix.getX() ) && (this.getY() == pix.getY()) ) return true;
    return false;
}

//Fonction convertissant les coordonnées Pixel en coordonnées de la grille
public Point fromPixelToPoint() {
    Point p = new Point( (int) Math.floor( this.getX()/31 ), (int) Math.floor( this.getY()/31 ) ); //Division des coordonnées x et y du pixel par 31, arrondie vers le bas et cast en in
    return p;
}
```

Méthodes isEqual() et fromPixelToPoint()

Ces 2 classes sont donc des classes « coordonnées ».

2) Dijkstra et Pile :

La classe Dijkstra est une classe avec plusieurs attributs. Un Point *sommet*, un int *valeur*, un Dijkstra *predecesseur*, un Dijkstra[] *sommetAdj* et un int *taille_sommetAdj*. Lors de l'appel du constructeur *sommet* prend la valeur d'un Point passé un argument, *valeur* prend 422 (valeur max possible dans notre cas), *sommetAdj* prend une taille de 4 et *taille_sommetAdj* prend 0.

La classe Pile est une pile de Dijsktra. Elle possède un tableau de Dijkstra dont la taille est passée dans le constructeur ainsi qu'un entier *sommet_pile* qui indique ou se situe la dernière valeur dans le tableau. Elle possède 2 méthodes, *push(Dijkstra d)* qui ajoute un élément au tableau et *pop()* qui en retire un.

```java
public class Dijkstra {

    public Point sommet;
    public int valeur;
    public Dijkstra predecesseur;
    public Dijkstra[] sommetAdj;
    public int taille_sommetAdj;

    public Dijkstra(Point sommet) {
        this.sommet = sommet;
        valeur = 422;
        sommetAdj = new Dijkstra[4];
        taille_sommetAdj = 0;
    }

    //Fonction modifiant le prédécesseur de l'instance
    public void predecesseur(Dijkstra predecesseur) {
        this.predecesseur = predecesseur;
    }

    //Fonction ajoutant un sommet au tableau sommetAdj
    public void addSommetAdj(Dijkstra d) {
        sommetAdj[taille_sommetAdj] = d;
        taille_sommetAdj++;
    }
```

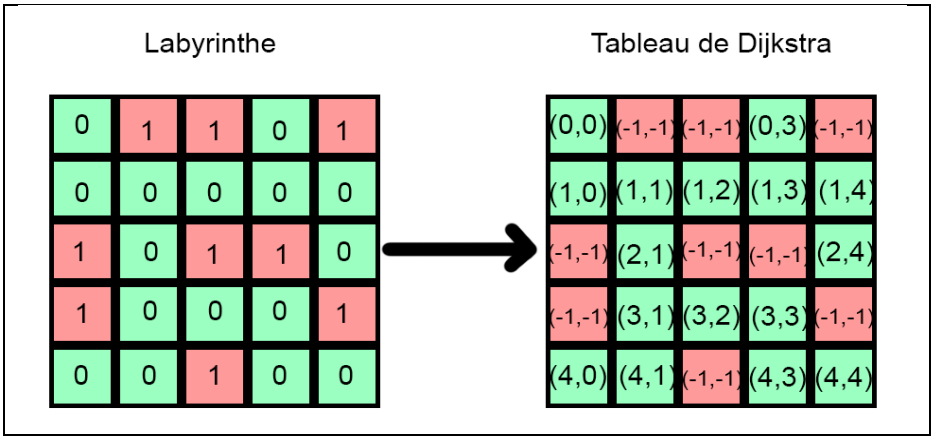
```java
public class Pile {

    public Dijkstra[] pile;
    public int sommet_pile;

    public Pile(int taille) {
        pile = new Dijkstra[taille];
        sommet_pile = 0;
    }

    public void push(Dijkstra d) {
        pile[sommet_pile] = d;
        sommet_pile++;
    }

    public Dijkstra pop() {
        sommet_pile--;
        return pile[sommet_pile];
    }

}
```

Classes Dijkstra et Pile

Ces deux classes n'ont qu'une utilité : Trouver le chemin le plus court entre 2 points de la grille. Pour ce faire on utilise plusieurs fonctions. Une fonction que je n'ai volontairement pas présenté est la fonction *tab1().* Elle se situe dans Map. Elle a pour but de transformer la grille en un tableau de 0 et de 1, 0 correspondant à une case sur laquelle on ne peut pas marcher, 1 correspondant à l'inverse.

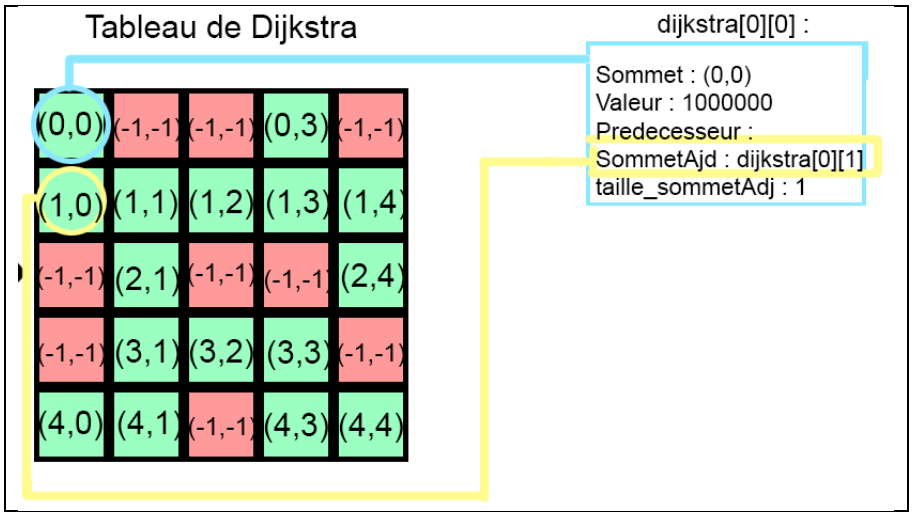
On utilise cette fonction avec la fonction *createTableau()* située dans Ennemi. *CreateTableau()* fait, à partir du tableau de 0 et de 1, un tableau de Dijkstra. Si la valeur du tableau est 0, on crée un élément de Dijkstra de sommet (-1 ,-1), sinon on en crée un de sommet (i,j).

Projet Info4B : BURGER TIME



(Les 0 et les 1 ont été inversé mais cela ne change rien à l'idée)

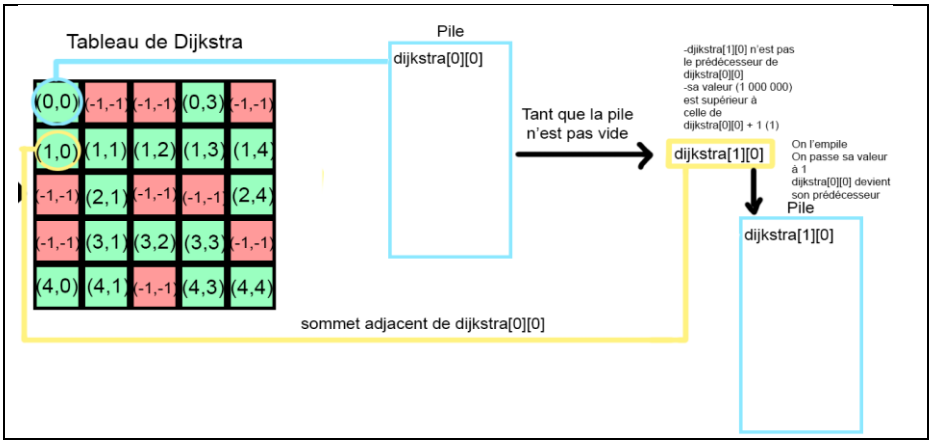
Un fois ce tableau de Dijkstra crée on va reparcourir le tableau de 0 et de 1. Pour chaque case valant 1, on va regarder les 4 cases autour d'elle. Pour chaque case autour ayant une valeur de 1 on l'ajoute au tableau sommetAdj de la cellule correspondant du tableau de Dijkstra. Cela permet ainsi de finir le remplissage du tableau de Dijkstra, chaque case possède une instance, chaque instance possède un tableau avec les instances voisines.



(ici on a mis valeur = 1 000 000 au lieu de 422)

Ce n'est qu'une fois ce tableau crée que l'on peut utiliser la méthode, dont nous avons déjà parlé, *cheminLePlusCourt(Point Depart, Point Arrive).* A partir de ces 2 point fourni on trouve les instances de Dijkstra correspondantes. On prend celle de départ, on initialise sa valeur à 0, son prédécesseur est elle-même et on la met dans la pile.

Tant que la pile n'est pas vide on dépile un élément appelé sommet courant. On regarde les sommets adjacents de cet élément. Si le sommet adjacent n'est ni le prédécesseur du sommet courant ni le sommet de fin, alors on regarde la valeur du sommet adjacent. Si elle est supérieure à la valeur du sommet courant + 1 alors on met le sommet adjacent dans la pile, on définit son prédécesseur comme étant le sommet courant et on définit sa valeur à celle du sommet courant + 1.



Pour finir on remonte les prédécesseurs à partir du sommet de fin jusqu'à retourner juste avant le sommet du début. Le point de cette instance est le Point retourné. Il ne faut également pas oublier de remettre toutes les valeurs du tableau de Dijkstra à 422.

f) Reseau :

Le package Reseau est composée de 3 classes : La classe Serveur, ConnexionClient et la classe Client. Ces deux classes ont pour but d'assurer une communication entre les participants d'une partie multijoueur.

1) Serveur et ConnexionClient :

La classe serveur st un Thread et a 2 rôles. Le premier est bien évidemment de créer le serveur. Le second est de créer et de maintenir la connexion avec les participants tant que la partie n'est pas terminée. Pour cela on doit donc créer un serveur multi-clients.

Le serveur multi-clients est créé à l'aide d'une boucle while. Tant que le nombre de connexion maximale n'est pas atteint on accepte les connexions entrantes, que l'on stocke dans un tableau ConnexionClient[] static. Si on atteint le nombre maximal de connexion on entre dans une autre boucle infinie dont on ne sort que si setup.end *(un booléen)* passe à true, ce qu'il fait en fin de partie.

Nous avons dans la classe serveur 3 éléments static : int *numclient* qui représente le numéro du client qui se connecte. Il est incrémenté à chaque connexion, int max_*connexion* qui représente le nombre de connexion maximale et ConnexionClien[] *cc* qui est le tableau stockant les connexions. Le tableau cc est déclaré static car n'importe que socket coté serveur peut forcer l'envoie de message des autres sockets au clients. Quant à *numclient* et max_*connexion* ils permettent de savoir quand sortir d'une boucle infinie. Une fois la connexion acceptée, l'objet ConnexionClient créé puis démarré et l'entier *numclient* incrémenté on reboucle.

La classe ConnexionClient est dans le même fichier que la classe Serveur. C'est un Thread également. Il a pour attribut un id (*numclient* passé en argument), un socket (passé en argument également), un BufferedReader et un printWriter.

La méthode run de ConnexionClient est la suivante : On commence par envoyer l'id au client. On entre dans une boucle infinie dont on ne sort que si le String str vaut « END ». On lit le String reçue du client. S'il vaut « updateSpritesWaitingRoom » alors on envoie à tous les clients « updateSpritesWaitingRoom »

ainsi que notre id. Cette commande a pour but de mettre à jour l'interface graphique dans la salle d'attente multijoueur. Un autre moyen qu'on a de sortir de cette boucle infinie est d'atteindre le nombre de connexion maximale. Dans ce dernier cas on envoie « END » au client 3. Ce dernier répondra en nous envoyant « END » en retour au socket coté serveur numéro 3.

Il faut également savoir que dans cette salle d'attente l'hébergeur peut à tout moment appuyer sur entrer pour envoyer à l'aide du client 0 un « END » et ainsi sortir de la boucle.

```
pw.println(id); //On envoie notre id au client

String str = "a"; //On initialise str à a

while(true) { //Boucle infinie

    if(str.equals("END")) break; //Si str vaut END on sort de la boucle
    str = br.readLine(); //str lit la valeur reçue

    switch(str) { //On prend str
        case("updateSpritesWaitingRoom") : //Si il vaut updateSpritesWaitingRoom
            for(int i=0; i<Serveur.numclient; i++) { //Pour chaque client
                Serveur.cc[i].pw.println("updateSpritesWaitingRoom"); //On envoie updateSpritesWaitingRoom
                Serveur.cc[i].pw.println(id); //On envoie notre id
            }
            break;
    }

    if(Serveur.numclient == Serveur.max_connexion) {
        if(id == 3) Client.pw.println("END");
        //break; //Si le nombre max de connexion est atteint on sort de la boucle
    }
}
```

Première boucle infinie du serveur

Un fois sortie de cette boucle le socket coté serveur d'*id* 0 rentre dans une boucle for. À l'aide du tableau static *cc* cette boucle envoie à tous les client le message « END », sortant ainsi les clients de leur boucle infinie *(code coté client plus bas)*.

On rentre ensuite dans la deuxième boucle infinie de ConnexionClient. Cette boucle a pour but d'actualiser les positions de chaque joueur sur leur fenêtre en simultané. On commence donc par lire les entiers *x* et *y* reçu des clients. On passe ensuite dans une boucle for qui envoie à chaque client, excepté celui qui nous a envoyé les coordonnées, l'id du socket coté serveur (qui permet de savoir quelle entité déplacer) et les coordonnées *x* et *y*. On lit ensuite le String reçu du serveur qui, s'il vaut « END », nous fait sortir de la boucle infinie.

```
while(true) { //Boucle infinie

    int x = Integer.parseInt(br.readLine()); //On lit la coordonnée x
    int y = Integer.parseInt(br.readLine()); //On lit la coordonnée y
    //System.out.println("[SERVEUR " + id + "] : Pos de " + id + " reçues : (" + x + "," + y + ")");

    for(int i=0; i<Serveur.numclient; i++) { //Pour chaque client
        if(i != id) {
            //System.out.println("[SERVEUR " + id + "] : Envoie des pos de " + id + " à " + i + " :
            Serveur.cc[i].pw.println(id); //On evoie notre id
            Serveur.cc[i].pw.println(x); //On envoie la coordonnée x
            Serveur.cc[i].pw.println(y); //On envoie la coordonnée x
        }
    }

    str = br.readLine(); //On lit le string reçu
    if(str.equals("END")) break; //Si il vaut ENd on sort de la boucle
}
```

Seconde boucle while coté serveur

Le Thread connexion client se termine en fermant le BufferedReader, le PrintWriter et le Socket.

2) Client :

La classe Client est complémentaire à la classe ConnexionClient. Elle est un Thread et sa méthode run exécute les commandes suivantes :

On commence par se connecter au serveur, on crée notre BufferedReader ainsi que notre PrintWriter puis on récupère l'id que nous envoie le serveur dans l'attribut *id*. On envoie au serveur « updateSpritesWaitingRoom » car à chaque nouvelle connexion il faut mettre à jour l'interface graphique des joueurs, puis on entre dans la première boucle while.

On ne sort de cette boucle infinie que si on reçoit « END » du serveur. On lit le String reçu du serveur, s'il vaut « updateSpritesWaitingRoom » on appelle la fonction du même nom en lui passant l'id du serveur en paramètre. Cette fonction appellera la fonction setSpriteEnnemi dans Setup qui mettra à jour les sprites sur la fenêtre.

```
pw.println("updateSpritesWaitingRoom"); //On envoie updateSpritesWaitingRoom au serveur
String str = "a"; //On initialise str à a

while(true) { //Boucle infinie

    if(str.equals("END")) break; //Si on reçoit END on sort de la boucle
    str = br.readLine(); //On lit le String reçu
    switch (str) {
        case("updateSpritesWaitingRoom") :  //Si on reçoit updateSpritesWaitingRoom
            updateSpritesWaitingRoom(Integer.parseInt(br.readLine())); //On appelle updateSpritesWaitingRoom avec
            break;                                                     //En lui passant l'entier reçu
    }
}
```

Première boucle while coté client

Un fois sorti de la boucle on appelle la fonction de Setup enterInGameMulti() qui charge sur la fenêtre les images adéquates pour jouer une partie. On rentre ensuite directement dans la deuxième boucle infinie.

On envoie nos coorodonnées *x* et *y* à qu'on obtient à l'aide les méthodes *getMyPosX(id)* et *getMyPosY(id)* de Setup. On lit la réponse du serveur. On stocke l'id reçu dans *idRecu*, et les coordonnées x et y dans *x* et *y*, puis on appelle la méthode *updateMyEntities(idRecu, x, y)* de Setup. On sort de la boucle si setup.end vaut true, sinon on envoie « enCours » au serveur.

```
setup.enterInGameMulti(); //On appelle enterInGameMulti de setup

while(true) { //Boucle infinie

    pw.println(setup.getMyPosX(id)); //On envoie la position x de notre entitée
    pw.println(setup.getMyPosY(id)); //On envoie la position y de notre entitée

    //System.out.println("[CLIENT " + id + "] : Envoie des pos (" + setup.getMyPosX(id) + "," + setup.getMyPosY(id)

    str = br.readLine(); //On lit le string reçu
    int idRecu = Integer.parseInt(str); //On transforme le string en int id reçu
    str = br.readLine(); //On lit le string reçu
    int x = Integer.parseInt(str); //On transforme le string en int x
    str = br.readLine(); //On lit le string reçu
    int y = Integer.parseInt(str); //On transforme le string en int x
    //System.out.println("[CLIENT " + id + "] : Nouvelles pos de " + idRecu + " recu : (" + x + "," + y + ")");
    setup.updateMyEntities(idRecu, x, y); //On appelle la méthode updateMyEntities de setup avec l'id reçu
                                          //et le coordonnées
    //System.out.println("[CLIENT " + id + "] : Actualisation des pos de " + idRecu + " : (" + x + "," + y + ")");

    if(setup.end) break; //Si le booléen end de setup vaut true on sort de la boucle
    pw.println("enCours"); //Sinon on envoie enCours au serveur
}
```

Seconde boucle while coté client

Le Thread client se termine en envoyant « END » au serveur, en fermant son BufferedReader, son PrintWriter et sn Socket.

III/Test et utilisation du programme :

La partie test et utilisation du programme est disponible en vidéo visionnable en cliquant sur les liens suivants :

Partie 1 : https://youtu.be/8-RQF9m8rNc

Partie 2 : https://youtu.be/52td1UihCOU

Projet Info4B : BURGER TIME

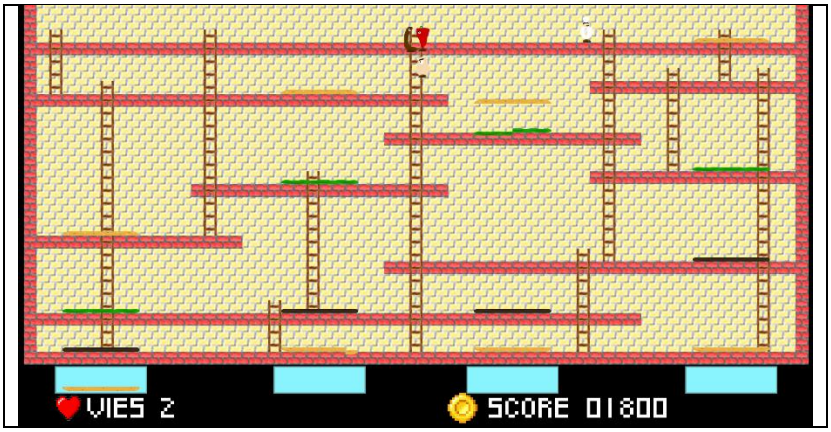
Le programme démarre en nous amenant sur l'écran titre :

On y retrouve 3 options : Joueur, Score et Quitter, ainsi que le bouton de sélection modélisé par les 2 points blancs.

N'ayant pas eu le temps de réaliser le score cette option nous fait quitter le programme

En cliquant sur jouer on a le choix entre jouer en solo ou en multijoueur.

Le bouton échap quant à lui nous fait revenir à l'écran précédent



Dans une partie solo classique on a le nombre de vies en bas (initialisé à 3) avec le score.

Comme on peut le voir il y a 3 ennemis qui nous poursuivent.

On peut aplatir en parti un burger, et une fois ce dernier dans le carré bleu il est assemblé.

En choisissant Multijoueur on peut ensuite choisir soit d'héberger la partie, soit d'en rejoindre une.

Attention il n'y a pas de sécurité, si un serveur est déjà créé rien n'empêche un autre joueur d'en créer un, de même que rien n'empêche un joueur de rejoindre un serveur si aucun n'existe.

Projet Info4B : BURGER TIME

L'écran d'attente est présenté de cette manière. On a l'hébergeur à gauche et les clients à droite.

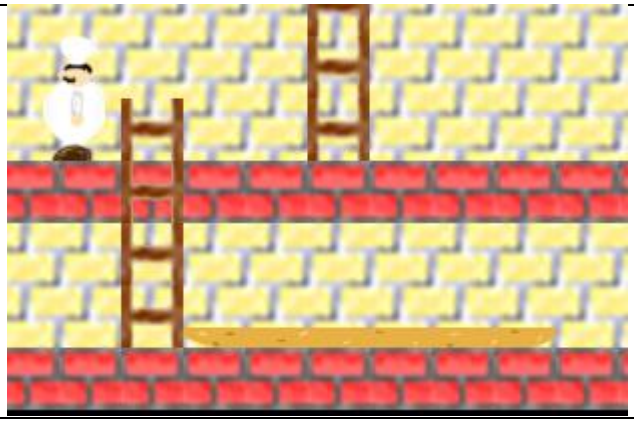
Au fur et à mesure qu'ils se connectent les clients apparaissent 1 à 1 sur la fenêtre de chaque joueur.

On commence par l'œuf, puis le piment et enfin la saucisse. Les joueurs n'ont pas le choix du personnage qu'ils jouent.

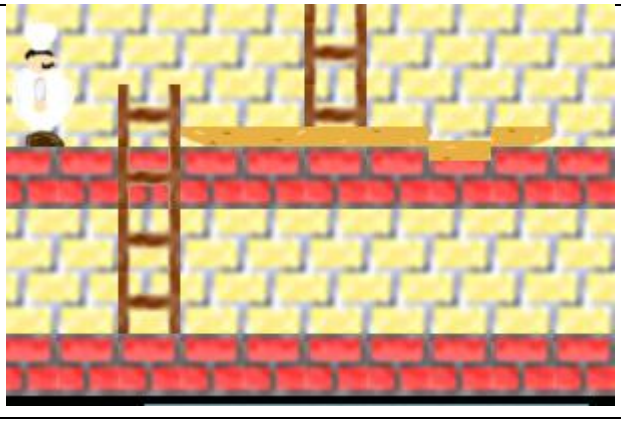
L'écran final de la salle d'attente est celui-ci. Il n'apparait qu'in court instant car la partie se lance directement.

Le mode multijoueur à 4 présente un bug. Il est donc impossible de joueur à 4 en même temps avec cette version du programme.

On peut cependant lancer une partie entre 1 à 3 joueurs. Cependant là aussi des bugs se manifeste. Le déplacement des entités dont on reçoit les positions sont très saccadé, elles sautent certains pixels. Cela implique que, par exemple, le joueur cuisinier qui écran une partie de burger entière sur sa fenêtre ne le fera pas forcément sur celle des autres joueurs :



Point de vue cuisinier



Point de vue œuf

Conclusion :

Le programme Burger Time que j'ai réalisé comporte 3 grandes structures :

- Un Noyau qui comporte les éléments, les met à jour et les affiches aux utilisateurs,
- Une Entrée / Sortie qui permet à l'utilisateur d'interagir avec le programme
- Un Réseau qui permet la communication entre plusieurs joueurs pour joueur ensemble

Ces trois parties sont réalisées à l'aide de 6 packages : *Launching*, *Plateau*, *Entity*, *Window*, *Position* et *Reseau* :

- *Launching* sert au démarrage et à la mise à jour du programme
- *Plateau* sert à la structure statique du jeu
- *Entity* sert à la partie dynamique du jeu
- *Window* à l'affichage utilisateur
- *Position* à réaliser des calculs
- *Reseau* à établir la communication entre les joueurs

On note cependant l'apparition de comportement indésirés dans le programme en mode multijoueur lorsque l'on souhaite jouer à 4 personnes. De même la mise à jour des meilleurs scores n'a pas été traité.

On pourrait apporter des améliorations à ce programme, notamment au niveau des entités. Dans le jeu original les entités présente sur un aliment en chute étaient soient écrasés soit aplatit et faisait gagner des points. De même le cuisinier pouvait lancer du poivre sur les ennemis les rendant immobiles et inoffensif un certain temps.

Bibliographie :

***Livre :* Programmer en JAVA, C.Delannoy**

Vidéos : *https://www.youtube.com/playlist?list=PLDKRpuLSe0f_4YqSROSWf_3wsdXwqefw9*
*https://www.youtube.com/playlist?list=PLBNheBxhHLQw30CKVBqfD1ateFz43qcU_*