**Projet INFO4B**

Sujet 2

*STIZ Romain*

**SOMMAIRE :**

**I)** [**Présentation :**](#I)

**1 - But du jeu**

**2 – Règle**

**3 – Ennemies**

**II)Analyse Fonctionnelle :**

**1** [**– Règle de fonctionnement**](#II)

**2**[**- Schéma**](#II1)

**III)Structure de données :**

**1-**[**Tableau**](#III)

**2**[**-ArrayList**](#III1)

**3-**[**Hashtable**](#III2)

**IV)Méthode vitales :**

**1-**[**Constantes**](#IV)

**2-**[**HelpMethods**](#IV1)

**3-[LoadSave](#IV2)**

**V) Architecture logicielle :**

**1-[Schéma](#V)**

**2-[Serveur](#V1)**

**3-[Playing](#V2)**

**4-[Game](#V3)**

**5-[State](#V4)**

**6-[Client](#V5)**

**VI)Algorithmes Principaux :**

**1 – [« main »](#VI)**

**2 – [« GameStates »](#VI1)**

**3 – [« Entities »](#VI2)**

**4 – [« object »](#VI3)**

**5 – [« utilz »](#VI4)**

**VII)[Jeu de Test](#VII):**

**I)Présentation :**

BurgerTime est un jeu vidéo classique d'arcade, sorti pour la première fois en 1982.

Le jeu a été développé par Data East et est rapidement devenu populaire grâce à son gameplay unique et addictif.

Dans BurgerTime, vous incarnez le personnage de Peter Pepper, un chef qui doit assembler des hamburgers en évitant des ennemis tels que des hot dogs, des oeufs et des cornichons.

Le but est de marcher sur les différents ingrédients pour les faire tomber jusqu'à ce qu'ils atteignent le bas de l'écran et forment un hamburger complet. Le jeu comporte plusieurs niveaux, chacun étant plus difficile que le précédent. Les ennemis deviennent plus rapides et plus nombreux, et la construction des hamburgers devient plus complexe à mesure que vous progressez.

BurgerTime est un jeu qui a connu un grand succès à l'époque de sa sortie, et il est toujours apprécié par les fans de jeux vidéo rétro aujourd'hui. Il a également été adapté sur plusieurs plateformes, notamment la NES, la Game Boy et l'Atari 2600.



[Cette photo](https://www.arcadeclub.co.uk/games/burgertime/) par Auteur inconnu est soumise à la licence [CC BY-SA](https://creativecommons.org/licenses/by-sa/3.0/)

Photo de l’image du jeu BurgerTime

Dans la version que je propose, il y a certaines modifications que je tiens de noter.

La première est sur les ennemies, en effet le jeu de base est composé de 3 types d’ennemies, sur ma version cependant il n’y en a que 2 !  
  
Le 1er, la Saucisse, celle-ci se déplace de droite à gauche (dès qu’elle touche un rebord).

C’est l’ennemie le plus simple à battre. De plus dès que ce type d’ennemie voit un joueur (voir Algorithme ennemie), celui-ci se dirige vers le joueur en visée.

Le 2nd, l’œuf, est un ennemi plus coriace car il connaît notre position sur le plateau à n’importe quel moment.

L’algorithme ne fonctionne pas encore assez bien pour être décrit comme implémenté complétement.

Le second changement apporté au jeu est le spray à poivre, qui permet initialement d’immobiliser les ennemies, celui permet maintenant de faire des dégâts à l’ennemie !

Ceci a été fait pour lutter contre le fait que l’animation d’immobilisation était beaucoup trop courte est devenez presque inutile dû fait de sa rapidité.

Le troisième changement apporté au jeu est la vitesse d’attaque des ennemies.

En effet, on peut esquiver les attaques de ceux-ci car les ennemies n’attaquent et ne nous font des dégâts que quand l’animation arrive à un certain tick.

Par exemple, quand la saucisse tend ses bras, c’est à ce moment précis qu’elle fait des dégâts.

Une image contenant texte, appareil, jauge

Description générée automatiquement

Image de la saucisse au moment de l’attaque.

**II)Analyse Fonctionnelle :**

* **Règle de fonctionnement :**

Nous voulons qu’un client se connecte au serveur, le client envoie juste un pseudo et un port.

Grâce à cela, le serveur va récupérer le pseudo afin d’en faire un Joueur et lancera le jeu avec tous les autres clients.

Celui-ci renverra ensuite à tous les clients la position des autres (sauf le même client).

De plus chaque Joueur inscrit son score dans un fichier texte afin de garder le meilleur score à jour.

* **Une image contenant diagramme

  Description générée automatiquementArchitecture :**



**III) Structures de données :**

* **Tableau :**

Un tableau est une structure de données qui permet de stocker une collection de valeurs de même type, qui peuvent être accédées et manipulées à l'aide d'index. Les tableaux en Java sont utilisés pour stocker des données de manière organisée et efficace, et ils sont particulièrement utiles pour stocker de grandes quantités de données de même type.

En effet dans mon code, je l’ai utilisé plusieurs fois pour charger la partie graphique du programme.

*Exemple* :

*Une image contenant texte

Description générée automatiquement*Pour afficher le personnage, je veux tout d’abord importer l’image en tant que BufferedImage. Pour cela nous avons une méthode qui permet de faire la conversion.

Une image contenant texte, intérieur, capture d’écran, argent

Description générée automatiquementEnsuite je récupère et j’appelle une fonction qui permettra de charger l’image en sous-image afin de pouvoir avoir pleins de petits ‘sprites’ de personnage.

Une image contenant alcool

Description générée automatiquementDans cet exemple, le tableau nous sert donc de quoi mettre à jour l’animation de notre joueur.

* **ArrayList :**

En Java, ArrayList est une classe qui implémente l'interface List, et qui représente une collection d'objets ordonnée et dynamique. Elle permet de stocker un nombre variable d'éléments d'un type donné, et de manipuler ces éléments de manière flexible et efficace.

Contrairement à un tableau classique en Java, la taille d'une ArrayList peut être modifiée dynamiquement au cours de l'exécution du programme, en ajoutant ou en supprimant des éléments de la liste. Cela rend l'ArrayList très pratique pour les situations où la taille de la collection est inconnue à l'avance, ou lorsque la taille peut varier au cours du temps.

Lors de ma programmation, les ArrayList étaient très utiles afin de faire des vérifications d’éléments.

*Exemple*:

Afin de passer au niveau suivant, je dois regarder que tout les burgers soient tombés en bas.

Pour cela je parcours mon ArrayList de Burger afin de regarder la position de chacun d’entre eux.

Une image contenant texte

Description générée automatiquementEn effet, la méthode n’est pas optimisée car on aurait pu seulement avoir une seule boucle de l’ArrayList.

Hormis cela, on voit l’utilité de l’ArrayList qui me permet de parcourir rapidement les éléments de celle-ci !

* **HashTable :**

En Java, Hashtable est une classe qui implémente l'interface Map, et qui permet de stocker et de récupérer des éléments en utilisant des clés. Elle représente une table de hachage, c'est-à-dire une structure de données qui associe des clés à des valeurs en utilisant une fonction de hachage.

La Hashtable en Java permet de stocker un nombre variable de paires clé-valeur, où chaque clé est associée à une valeur spécifique. Les clés doivent être uniques, c'est-à-dire qu'il ne peut y avoir qu'une seule valeur associée à une clé donnée. Les valeurs peuvent être de n'importe quel type, y compris des objets Java.

Une image contenant texte

Description générée automatiquementJe n’ai pas décider de l’utiliser malgré sa simplicité. En effet lors de la création de mon fichier score, j’aurais pu relier le nom du Joueur à son score. A la place de cela j’ai préféré utiliser un FileWriter et un BufferedWriter afin d’écrire dans un fichier.

**IV) Méthode vitale :**

Plusieurs méthodes sont vitales dans ce projet, aussi bien pour charger les images que pour faire des calculs sur la position.

Ces méthodes sont le plus souvent dans mon packages ‘utilz’.

* **Constants :**

La classe Constant de mon package utilz, contient comme son nom l’indique beaucoup de constantes ‘vitales’ au programme tels que la taille des blocs, la gravité,…

Concernant les méthodes cette classe ne contient seulement des méthodes permettant de retourner le nombre de sprites par animations.

Une image contenant texte

Description générée automatiquementOn voit que par exemple, si mon personnage est entrain de courir ‘RUNNING’ alors je retourne 6 car comme dans [l’image des sprites de mon personnage](#spriteperso), il y a 6 dessins à afficher sur cette ligne.

Cela permet donc d’afficher le bon nombre de sprite au bon moment.

* **HelpMethods :**

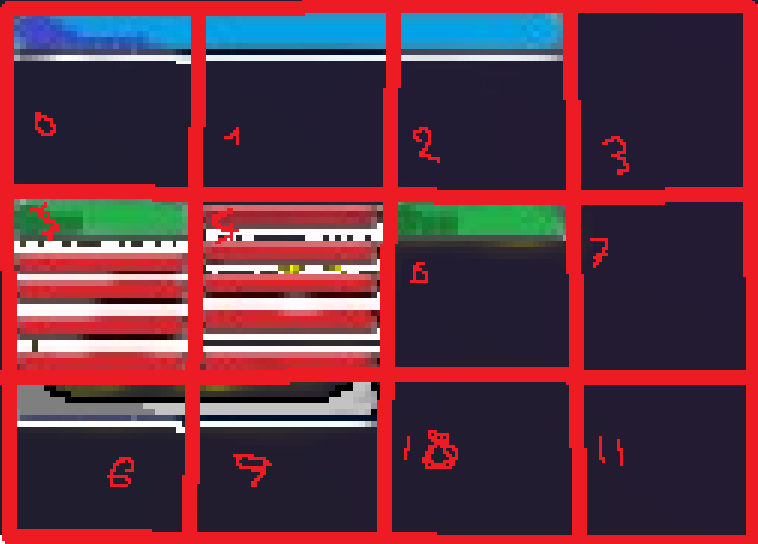
La classe HelpMethods contient cette-fois ci toutes les méthodes statics et importantes de ce projet. Elles permettent, pour la plupart de se repérer dans le jeu.

En effet, afin de savoir si, par exemple, la case est solide pour le joueur nous devons créer une méthode regardant chaque pixel et savoir si elle est ‘dite’ solide ou non.

Une image contenant texte

Description générée automatiquement

On peut voir que dans cette méthode, on regarde si les tiles 4, 5 et 7 seront considérées comme des échelles et permettront ainsi aux entités de pouvoir monter ou bien descendre.

On peut voir d’ailleurs que certains carrées sont vides, et ne sont là que pour avoir des fonctionnalités.

Le 3, on peut le traverser, c’est l’ ‘aire’ de notre jeu.

Le 7 nous sert d’échelle, car sinon, quand on monte, le joueur resterait bloqué à la tile 3.

Le 11, sert de bloque invisible afin que le joueur ne sorte pas des limites du terrain.

* **LoadSave :**

La dernière classe vitale pour ce projet est celle-ci, en effet elle permet au jeu d’avoir les différents niveaux en les chargeants dans des BufferedImage.

Pour voir un exemple de ce que l’on retrouve dans cette classe ([lien](#getSpritesAtlas)).

**V) Architecture logicielle :**

* **Schéma simplifié de l’architecture :**

**Une image contenant table

Description générée automatiquement**En se basant sur l'architecture d'un système d'exploitation, où chaque couche gère plus de tâches que les précédentes, la structure est établie couche par couche, de la plus haute à la plus basse. Le serveur, agissant comme le noyau d'un système d'exploitation, est chargé de la gestion de toutes les tâches.

* **<< Serveur >> :**

Le Serveur gérera le plus gros en termes de gestions des tâches, en effet celui-ci permet la liaison entre les clients et le jeu.

Selon un port donné, toutes les données seront transitées.

Une image contenant texte

Description générée automatiquement

On crée donc en premier lieu, un serverSocket afin d’avoir cette liaison TCP/IP.

Ensuite à chaque nouvelle connexion reçue, on crée un ClientHandler afin de gérer les communications avec celui-ci.

Une image contenant texte

Description générée automatiquement

La classe PlayerClientHandler permet de récupérer le pseudo du client afin de créer un Joueur avec celui-ci.

Il l’ajoute ensuite dans son ArrayList de Joueur et crée le jeu avec cette ArrayList.

Le Serveur reçoit ensuite les coordonnées de chaque Client et les diffuse à tous les autres Clients.

* **<<Playing>> :**

Playing est la classe qui permet de mettre en place tout le système du Jeu.

Il permet, à l’aide de ses méthodes et des liens avec les autres classes de créer cette univers, c’est à l’aide de sa méthode « initClasses() » que tout prend forme.

Une image contenant texte

Description générée automatiquement

Il appelle tout les constructeurs des autres classes afin d’initialiser le jeu.

LevelManager permet la création des niveaux. EnemyManager permet la création des Ennemies. ObjectManager permet la création de la nourriture. Ensuite on crée un joueur, on lui charge les données du niveau et son spawn.

* **<<State>> :**

State est une classe qui permet de savoir où l’on est dans le jeu.

Par exemple si je suis dans un MENU, ou bien entrain de jouer. C’est avec l’aide d’autres méthodes du package gamestate que nous afficherons le menu du jeu, la musique du jeu, la musique de l’écran principale,…

* **<<Game>> :**

Game sert au chargement de l’interface graphique et audio du jeu.

En effet, il appelle les constructeurs GameWindow, GamePanel, AudioOptions,… qui permettent de donner de la vie à ce jeu.

Mais, il est tout de même important car dans sa méthode run(), (car il implémente Runnable), il permet au jeu de définir sa limite de Frames Per Seconds mais aussi sa limite de Update Per Seconds.

Une image contenant table

Description générée automatiquementPour comprendre la logique de l’utilisation des FPS/UPS, il faut comprendre que si le jeu n’était défini qu’avec la puissance du PC, alors si le PC ne tourne le jeu qu’à 5 FPS, le joueur serait extrêmement ralenti contrairement à s’il tournait à 200 FPS.

Pour contrer ceci, j’utilise les UPS, qui ne sont pas en fonction de la puissance du PC, mais bien du code.

Donc peu importe le PC, le jeu aura 120 actualisations ‘logiques’. Ce qui permet d’avoir un jeu plus stable en termes de logiques malgré les différents PC.

* **<<Client>> :**

Le client est vu comme notre Joueur, en effet, il se connecte avec un pseudo au Serveur afin de pouvoir jouer au jeu.

Il envoie ensuite ses coordonnées au Serveur afin de les renvoyer aux autres.

**VI) Algorithmes Principaux :**

Je vais passer en revue les algorithmes les plus important de chaque package afin d’avoir une idée générale de comment sont faits les ennemies, les niveaux,…

* **<<main>> :**

**Game :**

Nous allons parler du seul algorithme important dans ce package qui est la méthode run() afin de créer la boucle de notre jeu.

Une image contenant texte

Description générée automatiquement

Pour comprendre facilement cet algorithme, je vais juste expliquer comment les FPS sont construits car ils sont semblables au UPS.

Nous avons besoin d’une variable de savoir combien de temps une frame dure en nanosecondes. C’est à ça que sert la variable timePerFrame qui divise par 1 nanoseconde / 120, qui est notre limite de FPS.  
Ensuite dans la boucle while, on regarde si le temps actuel en nanoseconde – la dernière nanoseconde / par timePerFrame est supérieur à 1, ce qui signifie qu’il faut réafficher avec repaint() afin d’atteindre les FPS désirées.

On reset ensuite cette variable et on incrémente notre variable Frame afin de savoir combien de FPS on a.

Chaque seconde on va afficher cette valeur en la réinitialisant à la fin de la seconde.

* **<<State>> :**

**Playing :**

Dans la classe Playing, on retrouve la méthode update() qui permet de mettre à jour les informations selon un contexte donnée.

Une image contenant texte

Description générée automatiquement

On voit que si le jeu est en pause, alors on affiche le menu Pause ce qui causera au jeu de s’arrêter en fond.

Si le niveau est complété, on affiche le menu pour le niveau suivant.

Si le joueur meurt, alors on met à jour le joueur qui changera d’état est deviendra mort.

Sinon, c’est que le jeu est en cours, alors on met à jour chaque donnée, les objets, le joueur et les ennemies.

**ScoreFile :**

Dans le package state, on retrouve de même la classe ScoreFile, qui permet d’écrire dans un fichier texte le score du Joueur. Je vais expliquer la méthode qui permet d’écrire dans ce fichier.

Une image contenant texte

Description générée automatiquement

On ouvre tout d’abord dans un FileWriter le fichier source en écrivant true afin de ne pas écraser les données déjà présentes.

Avec un BufferedWritert, on écrit donc, le score avec le nom du Joueur.

Bien sûr cette méthode est synchronized afin que plusieurs processus ne se rentrent en collisions afin d’écrire dans le fichier.

Cette méthode permet de comprendre comment fonctionne l’algorithme suivant qui sert à retourner le top 5 des Joueurs.

Une image contenant texte

Description générée automatiquement

Je précise avant que la même méthode existe pour retourner le nom des Joueurs, mais par soucis de comparaisons avec des String, les scores retournés n’étaient pas forcément les bons, car par exemple « 1999 » < « 999 » même malgré la conversion en Integer.parseInt().

Cette méthode lis donc ligne par ligne le fichier, sépare en 2 parties le nom de Joueur et le score tout en enlevant les possibles espaces restants grâce à trim().

Et on ajoute le score dans une ArrayList.

Grâce à la méthode sort() de Collection, je peux donc avoir par ordre croissant les scores, je m’assure donc de l’inverser et je retourne une nouvelle ArrayList contenant score (qui est trié) tout en ne prenant que les 5 premiers s’ils existent ou bien la taille de score. (si elle est inférieur à 5).

* **<<Entities>> :**

Le package entites contient toutes les classes qui permettent au projet d’avoir des ennemis, des joueurs,…

**Enemy :**

Dans la classe Enemy, on retrouve les 2 algorithmes importants pour faire déplacer les ennemis.

Une image contenant texte

Description générée automatiquement

La méthode move(), qui est utilisé par les Saucisses, permet à l’ennemi de courir de droite à gauche tant que le sol est solide pour lui.

Pour ce faire, on définit une variable xSpeed qui permet de savoir si l’ennemi va à gauche ou à droite.

Ensuite, on regarde si l’ennemi peut aller à la tile à côté de lui et si cette tile est solide.

Si tel est le cas, alors on incrémente xSpeed à la valeur x de l’ennemi.

Et on lui fait changer de direction à la fin.

La seconde méthode, qui ne fonctionne pas encore très bien car beaucoup plus complexe, est la méthode moveintoPlayer(), qui permet à l’œuf de nous chercher dans le jeu afin de nous tuer.

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Cette longue méthode, regarde tout d’abord si l’ordonné du Joueur est différente de celle de l’ennemi.

Si tel est le cas, l’ennemi vas marcher grâce à move(). Si le y (ordonné) du joueur est inférieur à celui de l’ennemi (càd que le joueur est plus haut) alors on regarde si l’ennemi est sur une échelle afin de savoir s’il doit monter ou descendre. (Dans ce cas il doit monter).

Tant qu’il n’est pas sur le haut de l’échelle, soit la [tile invisible n°7](#tiles), alors il monte, sinon on regarde s’il est à droite ou à gauche du joueur afin de le faire sortir de l’échelle.

On fait l’inverse s’il doit descendre.

Et si l’ennemi est au même niveau que le joueur, il va regarder s’il peut le voir, s’il peut alors il va se tourner vers lui et avancer.

Sinon il va appeler move() afin de marcher.

* **<<Object>> :**

**ObjectManager :**

Dans la classe ObjectManager, nous avons plusieurs méthodes qui permettent à la nourriture d’interagir avec son environnement.

Une image contenant texte

Description générée automatiquement

Cette méthode regarde si une hitbox (Rectangle2D.Float) est en collision avec une des nourritures, si tel est le cas, alors la nourriture va s’aplatir afin de signifier qu’elle est en mouvement et qu’elle va chuter.

Une image contenant diagramme, texte

Description générée automatiquementOn peut voir que à gauche la nourriture est normale mais à droite celle-ci est écrasée.

(Il n’y a pas de salade car qui prend de la salade avec son hamburger ?)

Une image contenant texte

Description générée automatiquement

Cette méthode permet de faire tomber la nourriture, en effet si elle est décrite comme active, alors elle va tomber.

On va ensuite parcourir la liste de nourritures afin de voir si la hitbox de la nourriture interfère avec la seconde.

Si c’est le cas, alors la première nourriture va s’arrêter, se changer afin de redevenir comme avant et la seconde va alors tomber et se changer pour s’aplatir.

Si la nourriture atteint l’ordonnée 1040 alors elle s’arrête car elle touche l’assiette en bas.

Une image contenant texte

Description générée automatiquement

La dernière méthode de cette classe regarde quand la nourriture tombe, si elle touche un ennemi.

Si c’est le cas, alors l’ennemi meurt et le joueur gagne 50 points.

* **<<Utilz>> :**

**HelpMethods :**

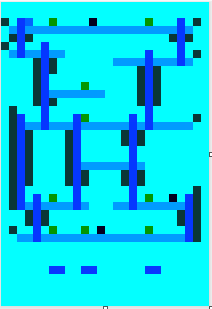
Cette classe contient une méthode unique afin de charger les niveaux.

Une image contenant texte

Description générée automatiquement

Pour comprendre assez facilement cette méthode, il faut voire comment est construit le niveau.

Prenons l’exemple du niveau 1.

Voici le niveau sous forme de dessin Paint.

La méthode va parcourir l’image et regarder la composante rouge de chaque pixel.

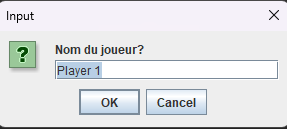
Cette composante correspond à une tile de notre [image](#tiles).

Par exemple, si la composante du premier pixel est égale à 1, alors on retournera l’image 1. Et ainsi de suite.

Mes autres méthodes, pour placer les ennemis ou la nourriture se base aussi sur cela mais elles regardent les autres composantes. (Verte et bleue)

**VII)Jeu de Test :**

Pour commencer, le jeu va ouvrir une fenêtre afin de demander le pseudo du Joueur.



Ensuite on arrive dans un Menu avec différentes options.

Une image contenant texte, horloge, capture d’écran

Description générée automatiquement

La 1ère option « Jouer » permet de lancer le Jeu.

La 2nd « options » nous amène dans un autre menu.

La 3ème « quitter » permet de fermer le jeu.

* **« Jouer » :**

Une fois cette option cliquer, on apparaît directement dans le jeu.

Une image contenant graphique

Description générée automatiquementOn peut aller de droite à gauche, ou si on est sur une échelle descendre ou monter.

La barre de vie descend quand on se prend un coup de la part d’un ennemi. On peut en prendre 3 maximum puis on meurt ensuite.

Exemple de la barre de vie après 2 coups.



Si on meurt voici le menu avec la musique « [Souliko](https://youtu.be/KSSZPxsj2bA) » en fond.

Une image contenant texte, tableau

Description générée automatiquement

Mais si on parvient à réussir le niveau, on passe à un écran de victoire avec le choix de revenir au menu ou de continuer au niveau suivant.

Une image contenant texte, signe

Description générée automatiquement

* **« Options » :**

Options permet d’ouvrir cette fenêtre qui permet de couper le son ou bien les effets du personnage.

On peut aussi revenir au Menu.