CNAM NFA032 : Programmation objet

Projet d'étude : Document d'analyse

Animation à l'écran et programmation orientée objet

Professeur ED : Tifanie BOUCHARA

Professeur titulaire : M.François BARTHELEMY

Etudiant : Alexandre CREMIEUX / SISCOL : 100110938

Année : 2014 / 2015, 2ème semestre

Table des matières

Type de logiciel développé en réponse au sujet

Le logiciel développé en Java dans le cadre de ce projet consiste en un jeu de Space Invaders simplifié. Le joueur manipule à l'aide du clavier un vaisseau positionné en bas de l'écran et tire vers le haut où des ennemis se déplacent de haut en bas suivant un ordre préétabli.

Documents fournis

- Cette note de synthèse au format PDF
- Un modèle UML décrivant les objets utilisés dans le cadre du projet
- Le code source du projet
- Une Javadoc du projet décrivant succinctement les méthodes développées
- Une archive .jar exécutable du programme en Java
- 10 images au format png (9 pour les ennemis, 1 pour le vaisseau)

Cahier des charges et expérience utilisateur

Le jeu, bien que simplifié par rapport à l'original, comporte les fonctionnalités suivantes :

- Les ennemis forment une matrice à l'écran et se déplacent tous en même temps (en tout cas l'illusion est donnée).
- Les ennemis peuvent tirer, vers le bas uniquement, et un ennemi ne peut tirer qu'une fois toutes les deux secondes. L'ordre de tir des ennemis est aléatoire. Seul le dernier ennemi d'une colonne peut tirer, c'est à dire celui se trouvant le plus bas à l'écran dans une colonne.
- Chaque colonne d'ennemis doit comporter au minimum trois ennemis au lancement d'une partie. Chaque ennemi est représenté au minimum par trois images qui s'inter-changent à chaque déplacement.
- Le vaisseau ne peut se déplacer que de gauche à droite.
- Le vaisseau ne peut tirer que vers le haut et au maximum deux fois pas seconde.
- Le vaisseau n'a qu'une seule image.
- Un tir de vaisseau peut détruire un ennemi lorsqu'il y'a collision. Un tir d'ennemi peut détruire le vaisseau de la même façon. Un missile allié et un missile ennemi peuvent se détruire mutuellement.
- Le jeu est terminé lorsqu'il n'y a plus d'ennemi ou lorsque le vaisseau est abattu.
- Un pop-up en fin de jeu propose à l'utilisateur de lancer une nouvelle partie.
- Si les images (sprites dans le code) ne sont pas chargées, le programme contourne le problème en proposant à la vue des carrés de couleur jaune (couleur choisie en fonction du fond d'écran noir).
- Lorsqu'une image disparaît de la fenêtre du jeu, il est retiré de tout ou partie des structures de données qui le collectionne après avoir disparu au fur et à mesure.

Modèle objet proposé (réponse technique) :

La page suivante présente le modèle objet du logiciel suivant une nomenclature UML simplifiée (NDR : venant de découvrir UML, la nomenclature présentée contient probablement certaines failles). Suivant l'énoncé du projet, les objets proposés dans ce modèle permettent essentiellement de manipuler des images qui leur sont propres à travers un classe dédiée afin d'animer ces images à l'écran suivant un comportement propre.

Le jeu est animé par la classe SpaceInvasion qui intègre le moteur du jeu et gèrent tous les objets présents à l'écran. Son but est de mettre en relation ces objets avec les structures de données développées spécifiquement autour d'eux et qui permettent de les animer à l'écran. L'objectif technique étant de manipuler ces objets aux travers de collections génériques. Les différents objets sont collectionnés dans des structures de type ArrayList. Le logiciel gère principalement quatre types d'ArrayList : <Enemy>, <Moves>, <Missile> (allié), <Missile> (ennemis).

Le vaisseau est libre et n'appartient à aucune collection. Ceci a été décidé par philosophie : le logiciel considère qu'un objet dont le contrôle appartient à l'utilisateur, supposé humain, ne doit pas être pris dans le contrôle effectué par le code sur les objets dont le comportement est programmé (typiquement les ennemis). Cependant, les missiles tirés par le vaisseau se trouvent dans une collection ad-hoc. Ceci s'explique pas le fait que le jeu émule un espace, les missiles alliés évoluant dans cet espace doivent subirent les caractéristiques de cet espace décrit par le code.

Chaque élément à l'écran est un objet dont la super classe est GameObject. Cette dernière est une classe abstraite qui ne comporte pas de comportement propre. Les classes qui héritent de GameObject sont les classes Enemy, Craft, Missile. La classe GameObject est rendue abstraite car chaque élément du jeu possède des comportements qui lui sont propres : déplacement, tir, chargement d'images, etc.

Chaque GameObject() utilise principalement deux classes pour représenter en mémoire un objet : la classe Rectangle qui détermine la position de l'objet à l'écran ainsi que son format, et la classe Image qui stocke la (les) image(s) (sprite) en mémoire et les régie en suivant un comportement standard à toutes les images.

GameObject implémente l'interface Moves qui spécifie les méthodes inhérentes au mouvement à l'écran. Cette interface fait d'ailleurs le lien entre les objets à l'écran et une classe nommée Vector (à ne pas confondre avec une classe du package java.util) qui permet de définir et de stocker en mémoire vive le déplacement de chaque GameObject(). Les différentes instances de Vector() sont stockées dans la mémoire par le biais d'une instance d'une classe en structure récursive nommée VectorQueue. Cette structure de données est une file doublement chainée redéfinie suivant le paradigme de la programmation orientée objet.

En plus de permettre le stockage des instances Vector(), la classe VectorQueue gère le temps de réponse de ces vecteurs, c'est à dire qu'elle contrôle le temps d'attente entre chaque appel de vecteur de translation par le moteur du jeu. Dans le contexte de la séparation entre objets contrôlés par le code et les objets contrôlés par l'utilisateur, la classe VectorQueue() reflète les caractéristiques de l'espace dans lequel évolue tous ces objets. Ceci peut être comparable à la vitesse d'un missile dans l'atmosphère comparativement au vide, les deux milieux engendrant des contraintes différentes (frottement de l'air, etc.). Le logiciel n'implémente pas ce niveau de complexité, mais ceci pourrait être envisagé dans un éventuel développement ultérieur.

Fonctionnement général du programme

1. Chargement des données et lancement du jeu

- Les objets sont manipulés par la classe SpaceInvasion, le main est donc situé dans cette classe. A l'appel de main(), un objet SpaceInvasion est instancié.

- Appel du constructeur de SpaceInvasion() :

    - Chargement des images des ennemis (elles doivent être situées dans le même dossier que l'application – archive jar exécutable)
    - Chargement de l'image du vaisseau (idem)
    - Entrée dans la boucle de contrôle :
        - Initialisations des collections et objets génériques du jeu (gmObjs, moves, craftVec, vectors)
        - Initialisation du vaisseau
        - Initialisation des ennemis : instanciation et création des références dans l'ArrayList *ennemies.*
        - Initialisation du fond et du tampon : remplissage de deux tableaux de int[] avec des pixels noirs
        - Ouverture de la fenêtre à l'aide d'une instance Afficheur() (si null)
        - Appel de fonction de boucle du jeu : renvoie un boolean définissant l'issue de la bataille
        - Ouverture d'un pop-up de contrôle permettant de savoir si l'utilisateur souhaite continuer
        - Destruction des références dans les collections (évite une collision en mémoire entre deux sessions du jeu)
        - Si le joueur souhaite quitter : sortie de la boucle de contrôle
        - En cas de sortie : fermeture de la fenêtre et destruction des références aux images en mémoire.

- Comportement de la boucle de jeu :

    - Remplacement du fond d'écran par son buffer
    - Calcul des collisions :
        - Si les coordonnées d'un objet (qui sont finalement des calculs d'adresses en mémoire) se superposent un test plus précis de collision est effectué pixel par pixel (si deux pixels visibles de deux objets se superpose, la collision est effective)
    - Gestion des collisions (nettoyage des structures de données, calcul de la visibilité)
    - Calcul des tirs ennemis (la méthode enmyShoot() contient un algorithme permettant un décalage aléatoire entre les tirs ennemis à partir du second tir).
    - Calcul du vecteur de translation des ennemis (vecteur commun)
    - Mise à jour du vecteur d'ennemis dans la structure de données VectorQueue()
    - Déplacement des objets sur le plan du jeu
    - Ajout des images au fond d'écran
    - Mise à jour de l'affichage
    - Test de sortie de la boucle: le vaisseau est-il toujours visible ?
    - Sleep() de 2 ms (pour diminuer la charge sur le processeur).

Notes sur la classe Afficheur

Pour permettre un contrôle du vaisseau au clavier, une instance Keylistener() appartenant à une instance de SpaceInvasion() a été rajoutée à Afficheur. Pour ne pas modifier le code donné, cet ajout est effectué en dehors du constructeur de Afficheur.

Dans la classe SpaceInvasion, l'appel à la fonction keyListener() permet d'instancier un objet Keylistener appartenant à l'instance afin de créer une relation (bijection ?) entre le contrôle de la fenêtre et le contrôle du jeu en lui même.

Cependant, un cas n'est pas géré : l'arrêt du programme par un clic sur la fenêtre (croix rouge). Ceci semble quitter le programme sans effectuer une destruction complète des références. Le cas est anodin en l'espèce mais pourrait être plus grave dans le cadre d'applications professionnelles.

Lancement du jeu

Les images du jeu doivent être placées dans le même dossier que l'archive jar exécutable. Le programme peut être lancé soit directement par l'interface graphique du système d'exploitation, soit par la console.

Dans ce dernier cas, il est nécessaire d'utiliser l'instruction suivante (système de type Unix) lorsque l'on se trouve dans l'archive se trouve dans le répertoire courant (-$ indique le prompt) :

```
-$ java –classpath SpaceInvdx.jar SpaceInvasion
```

Critiques personnelles sur le programme et difficultés rencontrées

Ce projet m'a permis de comprendre des aspects importants de la programmation orientée objet ainsi que la notion de référence. C'est à dire que certaines données ne doivent pas être passées directement par référence mais par le biais de copie mémoire en tampon. Par exemple, lorsque l'on souhaite modifier une image qui devra être réutilisée, il est préférable de créer une copie de la structure plutôt que de renvoyer directement sa référence. A ce titre, la dissociation entre le tas et la pile d'exécution par la JVM induit une utilisation importante de la notion de pointeur que l'on peut retrouver dans d'autres langages. Dans ces conditions, il m'a parfois semblé que le langage Java masquait la réalité ce qui m'a conduit en définitive à commettre des erreurs de conception. Je précise qu'il s'agit d'un avis personnel, en aucun cas d'une critique du langage.

Au cours du projet, je me suis aperçu que l'exécution du programme était parfois ralentie et que la consommation du temps processeur pouvait augmenter à l'exécution, surtout si d'autres programmes Java été lancés sur la machine exécutante. Il est donc évident que certaines des méthodes que je propose ici ne sont pas adaptées. Cependant, il reste encore difficile à mon niveau de proposer une optimisation.

Du point de vue du code, certaines méthodes échappent aux règles qui régissent l'organisation générale du programme. Par exemple, la méthode SpaceInvasion.enmyShoot() s'occupe à la fois de nettoyer l'ArrayList<Enemy> alors que le méthode SpaceInvasion.manageObjetcsCollision() remplie normalement cette fonction. Ce choix a été effectué dans l'approche d'une optimisation (ne pas parcourir trop souvent une structure lorsque ceci doit être fait ailleurs). Au cours du développement du jeu, ce choix ne s'est pas toujours révélé opportun.

Concernant l'utilisation des ArrayList, il s'est avéré que les interruptions matérielles déclenchées par le clavier et les actions qui en découlent sur les données du jeu provoquent des accès concurrents en mémoire. Il aurait donc été préférable d'utiliser la classe Vector du package java.util qui semble prendre en charge ce phénomène. Cependant cette structure n'était pas prévue dans l'énoncé du projet. Pour contourner le problème, le programme tente de retenir les actions de l'utilisateur en bloquant l'accès à la structure gmObjs() lorsque des parties de codes critiques sont exécutées. Ceci est fait par l'instruction « synchronized(*referenceStructure*){ } ». Ainsi, chaque méthode concurrente attend la disponibilité de la structure avant d'exécuter son propre code (notion de la concurrence et collision vue en NFA003 et NF004).

L'interface Moves semble ici superflue. Elle a été conçue en premier lieu afin de faire évoluer rapidement le code du programme en me permettant éventuellement de créer des objets non mobiles à l'écran comme des boucliers ou encore de proposer un code concentrant les méthodes susceptibles de varier dans une classe ad-hoc (par exemple rassembler dans un classe statique des variantes de la méthode move() pour faciliter la maintenance du code). Je n'ai pas cependant pas eu le temps de les développer, détails qui aurait dû être pris en compte dans le projet.

A la fermeture du projet, je remarque que même pour un projet de ce type, il existe des difficultés qui sont en partie dues aux bonnes et aux mauvaises organisations du code, à la référence des objets traités et à leur niveau d'indirection. Par exemple, est-il nécessaire de créer une nouvelle référence dans un objet, ce qui engendrera un nouveau niveau d'indirection et donc un ralentissement ? Etc. Dans une autre mesure, je me suis interrogé sur la complexité du langage Java et de sa machine virtuelle. Par exemple, est-il finalement opportun de laisser croire au programmeur qu'il ne gère pas de pointeur alors que ce n'est pas vraiment le cas : on pourrait effectivement se demander quels seraient les effets de bord dans un plus gros programme conçus sans assez de recul comme le programme proposé ici. Il apparaît aussi que le poids du programme en mémoire centrale augmente dans le temps.

En fin de compte, il me semble maintenant difficile de concevoir un programme correct sans avoir une certaine connaissance des mécanismes internes d'un ordinateur et, pour le langage Java, de la JVM, ceci afin de devenir capable de programmer des applications de plus grande envergure qui ne souffriront pas directement d'une mauvaise gestion de la mémoire et des cycles processeurs.

Dans le cadre d'une évolution possible du jeu et de la maintenance du code, il pourrait être possible de rassembler toutes les méthodes identifiées comme *standard*s dans une classe spécifique qui se rapprocherait plus d'un réel moteur. Par exemple, il pourrait être définie une méthode de type *Engine.createEnemy(enemyType)* qui permettrait de développer un autre petit jeu avec les mêmes classes ou d'instancier des ennemis se déplaçant sans relation avec la troupe d'ennemis standard en diminuant la gestion nécessaire de la part de l'utilisateur pour ces objets (à la manière des bibliothèques Java). L'écran pourrait aussi incorporer des objets (avec images) non mobiles qui pourraient servir de bouclier, ou la destruction complète des ennemis pourrait engendrer l'arrivée d'un ennemi maitre du jeu, ou faire clignoter l'ennemi avant qu'il disparaisse, etc.

Vector

x : int; y : int;

Vector(int x, inty) : void;

VectorQueue

next : VectorQueue();
prev : VectorQueue();
ref : int;
timeStamp : long;
time : long;
released : boolean;

VectorQueue(Vector(), int ref) : void;
VectorQueue(Vector(), int ref, long time) : void;
//Accesseur
getVector(inr ref) : void;
//Méthode d'instance
add(Vector toAdd, int ref) : void;
add(Vector toAdd, int ref, long time): void;
update(Vector(), int ref) : void
updateNow(Vector(), int ref) : void
empty() : void
getVector(int);
remove(int);
update(Vector, int) : void
updateNow(Vector : int) : void

Image

Private tab : int[]
hig : int
wid : int
size : int

Image(int wid, int hig, int[] image)
//Mutateurs
setPx(int, int, int) : void
setPx(int, int) : void
setSprite(Image) : void
//Accesseurs
getPx(int, int) : int
getPx(int) : int
getSprite() : int[]
getSize() : int
//Methodes d'instance
add(int, int, Image) : void
//Methodes private
cptPos(int, int) : private int

SpaceInvdx

play : boolean;
vectors : VectorQueue();
aff : Afficheur();
gmObjs : ArrayList<GameObject>();
screen ; Image(); buff : Image();
WID : int; HIG : int;
board : Rectangle();
moves : ArrayList<Moves>();
craft : Craft;
craftVec : Vector();
craftRec : Rectangle();
CRAFTIME : long; CRAFTMISSY: int; MSCRAFTME : long;
enmySprites : int[][][];
enmyRec : Rectangle();
enmyVec : Vector()
ennemies : ArrayList<Arraylist<Enemy>>();
enMsls : ArrayList<Missile>();
msls : ArrayList<Missile>();
ENMYMISSY : int, ENMYMISTIME : long;
ENMYSHOOTME : int, ENMYTIME : int;
gmKbLst : KeyListener();

initBackGround() : void
loadCraftSprite() : void
initCraft() : void
loadEnemiesSprites() : void
initEnnemies() : private void
addToVectors(Vector, int id, long time) : void
addToVectors(Vector, int id) : void
moveGameObjects() : private void
computeEnmyVector() : private Vector()
drawGameObjects() : void
craftShoot() : void
enmyShoot() : void
colputeShipVec(boolean left) : void
zeroShipVec() : void
computeCollision() : void
gameLoop() : boolean
keyListener(this) : KeyListener()
main() : void
freeCollections() : void
manageGameObjectCollision() : void

Rectangle

wid : int, len ; int, x : int, y : int;

Rectangle(int x, int y, int wid, int len) : void;

//Methode static
collision(Rectangle, Rectangle) : boolean

<<Abstract>>

GameObject

img : Image();
isMoving : boolean;
isColsn : boolean;
isVisible : boolean;
shooter: boolean;
rec : Rectangle();
gameId : static int;
nbGo : static int;

GameObject(int x, int y, int len, boolean is
--> Mutateur
setVisible(boolean) : void;
setCollision(boolean) : boolean;
setSize(int, int) : void;
--> Accesseur
getRectangle() : Rectangle();
isVisible() : boolean;
getGameId() : int;
--> Methodes d'instance
colision(GameObject()) : boolean;
isCovering(GameObject) : boolean;
move(Vector) : void;
shoot() : Missile();
overBounded(Rectangle()) : boolean;
//Methodes static
getGameId() : int;
nbOfGameObject() : int;

<<Interface>>

Moves

move(Vector) : void
getGameId() : int
isVisible() : boolean
overBounded(Rectangle()) : boolean

ImageUtil

Afficheur

Stockage dans des structures de données

Enemy

shootTime : private long
lastShTm : private long
gameId : private int
gId : private static int

Enemy(int x, int y, int wid, int hig, boolean isShoot
Enemy(int, int, int, int, boolean, int[][])
//Mutateurs
setShootTime(long) : void
//Accessseur
getGameId() : int
//Methodes d'instance
shoot() : Missile
//Methodes static
classId() : int

Craft

gameId : private int
instanceId : private int
shootTimle : private long
lastShTm : private long

Craft(int x, int y, int wid, int hig, boolean isShoot
Craft(int, int, int, int, boolean, int)
//Mutateurs
setShootTime(long)
//Accesseurs
getGameId() : int
//Method d'instance
move() : void
shoot() : new Missile()

Missile

gameId : private static int

Missile(int x, int y, int wid, int hig)
//Accesseur
getClassId() : int
getGameId() : int
//Méthodes d'instance
move() : void