
Projet IM : Jeu vidéo sur téléphone mobile

Juno, un jeu de rôle innovant

Lasorsa Yohan, Strievi Florent, Liodenot David, Perrucaud Romaric,
Tabet Kamel, Bumbolo Julien, Xue Dinghua

Sommaire

1. Introduction

Aborder la conception d'un jeu vidéo innovant sur plate-forme mobile n'est pas une chose aisée ; si l'idée initiale a bien évidemment un rôle important, il faut cependant savoir faire preuve de réalisme de pragmatisme afin de ne pas perdre de point de vue un aspect essentiel du projet : finaliser un prototype de jeu, fonctionnant sur téléphone mobile.
C'est donc dans ce but précis que nous avons choisi d'orienter nos choix de conception et de développement, faisant attention au poids de chaque fonctionnalité ajoutée, et en évaluant la faisabilité de chacune des idées avant de se lancer dans du code inutile.

Afin de pouvoir effectuer une répartition des tâches efficace, nous avons tout d'abord étudié les différentes parties du travail à réaliser, ainsi que les divers outils à utiliser pour leur conception. A partir de cette définition des besoins, nous avons ensuite pu répartir au mieux chaque tâche en fonction des compétences de chacun. Chaque membre du projet a pu ainsi mettre à profit ses connaissances dans ses domaines de prédilection, ce qui nous a permis de travailler de manière efficace et avec une plus grande motivation.

Cette approche nous a donc permis d'avancer relativement rapidement tout en gardant une bonne cohérence entre les différents éléments grâce à l'attribution d'un rôle de manageur, essentiel sur projet de ce type afin de garantir l'intégration et la cohésion des travaux.

La réalisation de ce projet s'est donc déroulée de manière organisée, et si de nombreux problèmes majeurs se sont posés durant le développement, nous avons pu à chaque fois y faire face et trouver une solution, pour finalement réussir à réaliser l'ensemble des fonctionnalités souhaitées, et avoir un prototype de jeu fonctionnant sur un véritable téléphone mobile et non pas seulement sur émulateur.

2. Présentation du jeu

a) Scénario

Durant plusieurs années l'homme a négligemment utilisé les ressources de sa planète dans son propre intérêt sans penser aux conséquences de ses actes. Aujourd'hui, tous les êtres humains ainsi que leur écosystème sont en danger et amenés dans un avenir proche à disparaître. Une dernière solution s'offre à eux afin de remédier à ce terrible dénouement : JUNO, une mission spatiale mis au point secrètement depuis deux ans par la World Space Agency.

Le but de cette mission est de tenter de trouver de nouvelles ressources et informations importantes sur une planète se trouvant dans une galaxie voisine. Cependant, comme tout projet de cette envergure, le secret ne fut pas gardé bien longtemps. Les enjeux socio-économiques sont importants et les russes seront prêts à tout pour faire échouer cette mission et tirer profit plus tard de cette planète quitte à détruire le vaisseau.

JUNO est un jeu de rôle passionnant et prenant, mettant en scène des saboteurs russes, des techniciens et un commandeur prêts à mettre leur vie en péril pour la réussite de la mission, ainsi qu'un extra-terrestre ayant pris l'apparence d'un membre de l'équipage semant le doute et la discorde au sein de l'équipe.

b) But du jeu

Comme nous avons pu l'expliquer dans l'introduction précédente, ce jeu se déroule dans l'espace et plus précisément dans un vaisseau spatial. JUNO met en scène 6 personnages représentés soit par l'utilisateur, soit par une intelligence artificielle évoluée.

L'équipage du vaisseau est ainsi constitué de :

- 1 capitaine *(commander)*
- 2 saboteurs *(wreckers)*
- 2 techniciens *(technicians)*
- 1 extra-terrestre *(alien)*

La répartition des rôles se fait de façon aléatoire au début de la partie, de même que la répartition de chacun dans les différentes salles du vaisseau (en mode multijoueur, la personne qui crée le serveur de jeu aura le rôle du capitaine). Nos 6 personnages évoluent à bord d'une navette composée de 10 pièces comme le montre le schéma ci-dessous :

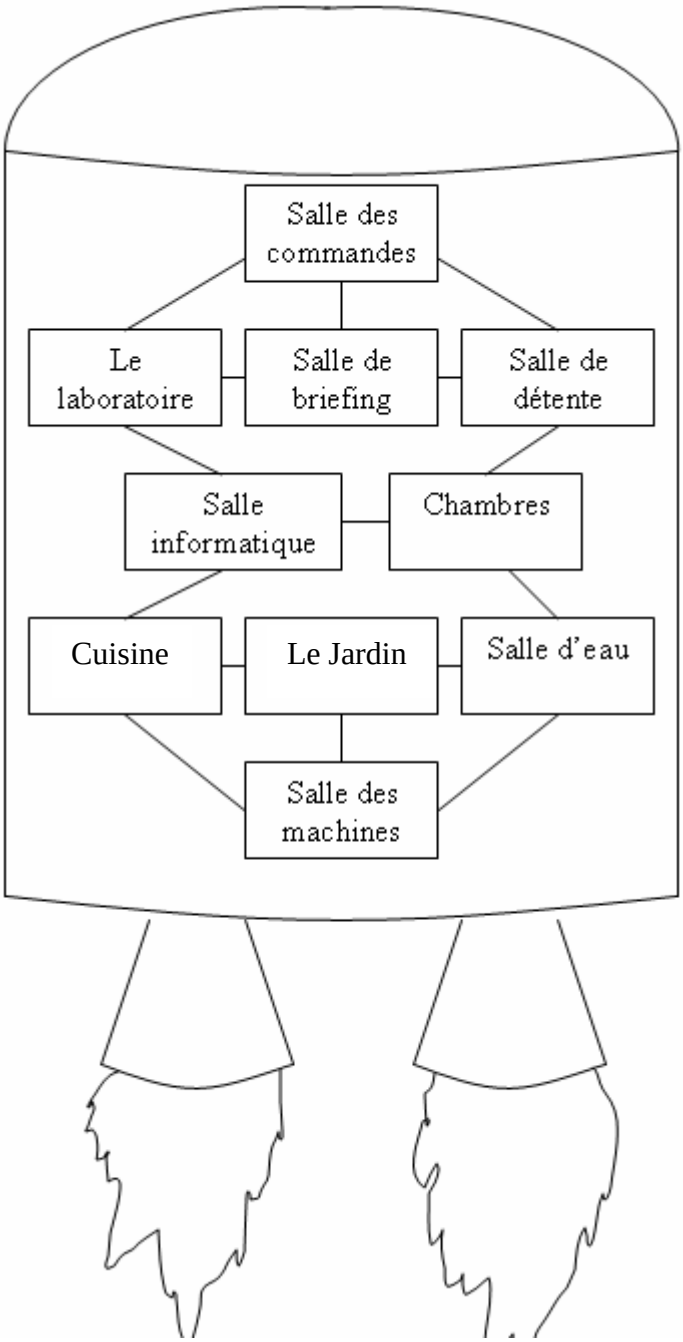


Fig. 1 : Plan de la navette spatiale

Chaque joueur ayant un rôle différent, il a donc un but différent afin de gagner la partie. :

- ➔ Les techniciens ainsi que le capitaine doivent faire arriver le vaisseau à destination, afin de pouvoir mener la mission à bien.

- ➔ Les saboteurs doivent saboter au moins 5 salles du vaisseau avant qu'il n'arrive à destination, pour faire exploser celui-ci.

- ➔ L'extra-terrestre doit semer le doute et faire en sorte d'être le dernier survivant à bord avec le capitaine, afin de dévorer celui-ci et prendre le contrôle du vaisseau.

c) Déroulement du jeu

Tout d'abord, au commencement d'une partie, le personnage incarné par le joueur est affiché, ainsi que son rôle dans la partie et son objectif.

Ensuite une vue globale du vaisseau est affichée (radar), sur laquelle il est possible de zoomer/dézoomer, montrant la position de chaque joueur dans la navette spatiale.
Le personnage incarné par le joueur est entouré d'un cercle rouge, et le capitaine du vaisseau d'un cercle vert avec une mention le précisant. Une fois bien ces positions observées et mémorisées, la partie peut commencer.

Fig. 2 : Ecran de présentation du personnage (à gauche) et affichage du radar (à droite)

Le déroulement principal du jeu se décompose en 5 phases que nous allons maintenant détailler plus précisément :

- **Phase 1** : Déplacement *(move)*
  Chaque joueur peut se déplacer ou non dans une pièce limitrophe à la sienne. Durant cette phase, un radar peut éventuellement être activé lorsqu'il est disponible, et permet ainsi de voir qui se situe dans les pièces voisines.

- **Phase 2** : Action *(action)*
  Les actions dépendent du rôle de chacun : un saboteur peut saboter des salles, un technicien peut réparer deux salles sur toute la durée de la partie et l'extra-terrestre peut réparer une unique salle et en saboter une. Durant cette phase, chacun peut constater l'état de la pièce (*en fonctionnement* ou *détruite*), affiché en bas de l'écran.

- **Phase 3** : Réaction *(reaction)*
  Les pièces sont détruites, réparées ou rien ne se passe. Ce jeu étant doté de son, chaque action (destruction ou réparation) effectuée par un joueur aura donc une conséquence sonore. Le résultat des actions sur la pièces courante ainsi que les pièces voisines sera représenté par un onde sonore ainsi qu'un bruit positionné en 3D. Le son pourra ainsi

être entendu par chaque joueur proche du lieu de l'action. Pour plus de vice, une réparation de salle fera le même bruit qu'un sabotage de salle.

Fig. 3 : Capture d'écran du jeu, phase de réaction

- **Phase 4** : Vote *(vote)*
  Chaque personnage choisit de vote pour quelqu'un ou alors vote blanc. Les joueurs attendent ensuite le verdict du capitaine qui prendra la décision d'éliminer quelqu'un ou non, en se basant (ou non, suivant son humeur) sur les résultats des votes.

Fig. 4 : Capture d'écran, phase de vote (choix du capitaine)

- **Phase 5** : Décision *(decision)*
  La décision prise, une personne est éliminée ou non du jeu et le jeu reprend de nouveau à la phase 1 pour les personnages restants.

3. Organisation du projet

a) Décomposition du travail

Comme expliqué en introduction, afin d'optimiser au mieux le développement et de pouvoir profiter des connaissances de chacun, nous avons tout d'abord, avant même d'attaquer la phase de conception du projet, défini des domaines de travail en se basant sur les différents éléments nécessaires à notre projet.

Chaque domaine ainsi défini est indépendant vis-à-vis des autres domaines, et correspond à une partie clé du projet. Ceci a donc permis à chacun des membres du groupe d'assumer une responsabilité concernant le projet et à la fois de paralléliser de manière efficace la réalisation des divers travaux.

Les différents domaines qui ont ainsi été dégagés sont les suivants :

- Design du jeu (règles, phases de jeu, principes…)
- Design et conception graphique (définition d'un thème, charte graphique, écrans du jeu, sprites…)
- Design, conception et réalisation des menus (écriture des SVG, définition de l'interface…)
- Design et conception de l'interface du jeu
- Ecriture d'un scénario et de la trame de fond
- Design et réalisation des séquences d'animation (écriture du storyboard, création graphique, animation, création et synchronisation de la bande son…)
- Design et réalisation de la partie sonore (voix, effets spéciaux, musiques d'ambiances…)
- Conception et réalisation du moteur de jeu
- Conception et réalisation d'un manageur de son (musique, FX et sons 3D)
- Conception et réalisation d'un système de lecture des séquences d'animation
- Conception et réalisation du système réseau bluetooth pour le multijoueur
- Conception et réalisation de l'intelligence artificielle

b) Répartition des tâches

Certains des domaines définis précédemment contenant beaucoup plus de travail que d'autres, certains membres ont été amenés à travailler sur plusieurs domaines et en collaboration.

Voici comment la répartition des différentes tâches de ce projet a été réalisée au sein du groupe :

- *Bumbolo Julien :* design et réalisation des séquences d'animation, écriture et réalisation de la trame de fond et du scénario, aide à la conception graphique.

- *Tabet Kamel :* design et conception graphique, aide pour l'écriture des storyboards et à la conception graphique pour les séquences d'animation, conception des menus.

- *Perrucaud Romaric :* design, conception et réalisation des menus, design de l'interface utilisateur, conception graphique des personnages, aide au design du jeu.

- *Lasorsa Yohan :* gestion du projet, design du jeu, conception et réalisation du moteur de jeu et de l'intelligence artificielle, conception graphique de l'interface du jeu.

- *Liodenot David :* design du jeu, conception et réalisation du système réseau bluetooth pour le multijoueur et intégration de celui-ci dans le moteur de jeu.

- *Strievi Florent :* design et réalisation de la partie sonore, conception et réalisation du manageur de son et du lecteur d'animation, aide à la réalisation du réseau bluetooth.

c) Organisation des sources et ressources du projet

Afin de pouvoir travailler de manière optimale, nous avons tout d'abord réfléchi à la manière d'organiser les sources du projet.

Afin de pouvoir s'affranchir du problème du développement simultané sur des fichiers sources, nous avons organisé la hiérarchie de classe afin que chaque partie puisse être développée indépendamment des autres :

- La classe *JunoMidlet.java* contient la base du programme (le midlet) et tout ce qui concerne la gestion des menus.

- La classe *JunoGameCanvas.java* contient le moteur d'affichage du jeu ainsi que le traitement des touches

- La classe *Mission.java* contient tout ce qui concerne la logique du jeu, ainsi que l'intelligence artificielle.

- La classe *SoundManager.java* contient tout ce qui concerne la gestion du son.

- La classe *Server.java* contient les méthodes relatives au serveur bluetooth.

- La classe *Client.java* contient les méthodes relatives à un client bluetooth.

Cette répartition de classe possède aussi un gros avantage, c'est que l'intégration des différentes sous-parties (comme le son, le bluetooth) s'est faite très rapidement et sans aucun soucis puisque chaque élément essentiel a sa propre classe : par exemple, tout ce qui concerne

la gestion du multijoueur via le réseau bluetooth s'est greffée par la suite dans la classe *Mission.java*, puisque c'est elle qui contient toute la logique du jeu. Durant cette intégration, il était donc encore possible de travailler sur la partie d'affichage et intégrer par exemple le son dans la classe *JunoGameCanvas.java* sans problème et sans causer d'effets de bord.

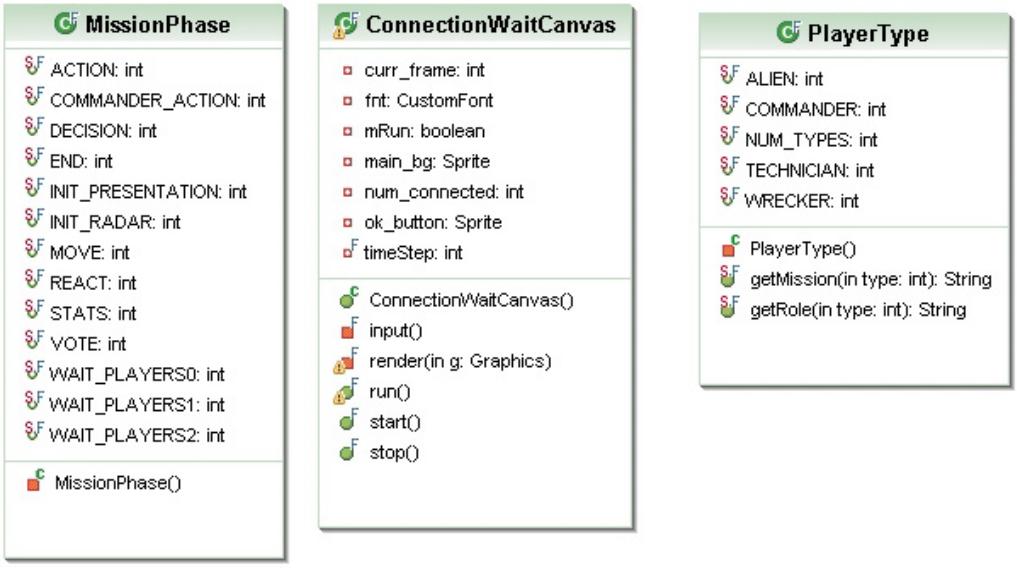
De même, afin de s'y retrouver de manière claire au sein des nombreuses ressources du projet (sons, musiques, documents SVG, images, vidéos…), chaque type de ressource dispose de son propre répertoire dans la racine du projet :

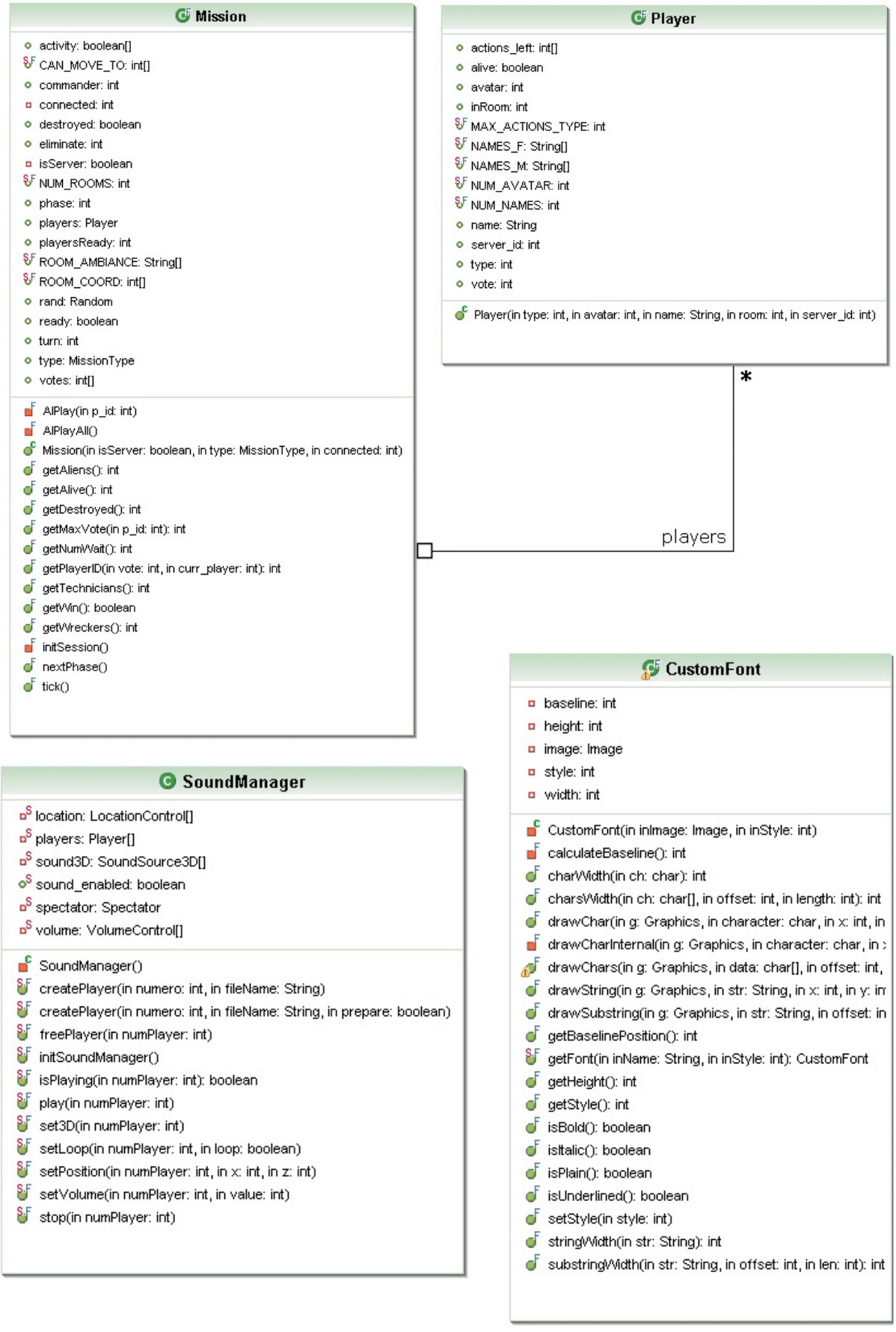
- */game :* les sources (.java) du jeu
- */icons :* les icônes du jeu
- */images :* les images du jeu (fonds et sprites animés)
- */sounds :* les sons et musiques du jeu
- */svg :* les documents SVG utilisés pour le menu
- */videos :* les séquences vidéos d'animation

2. Architecture du projet

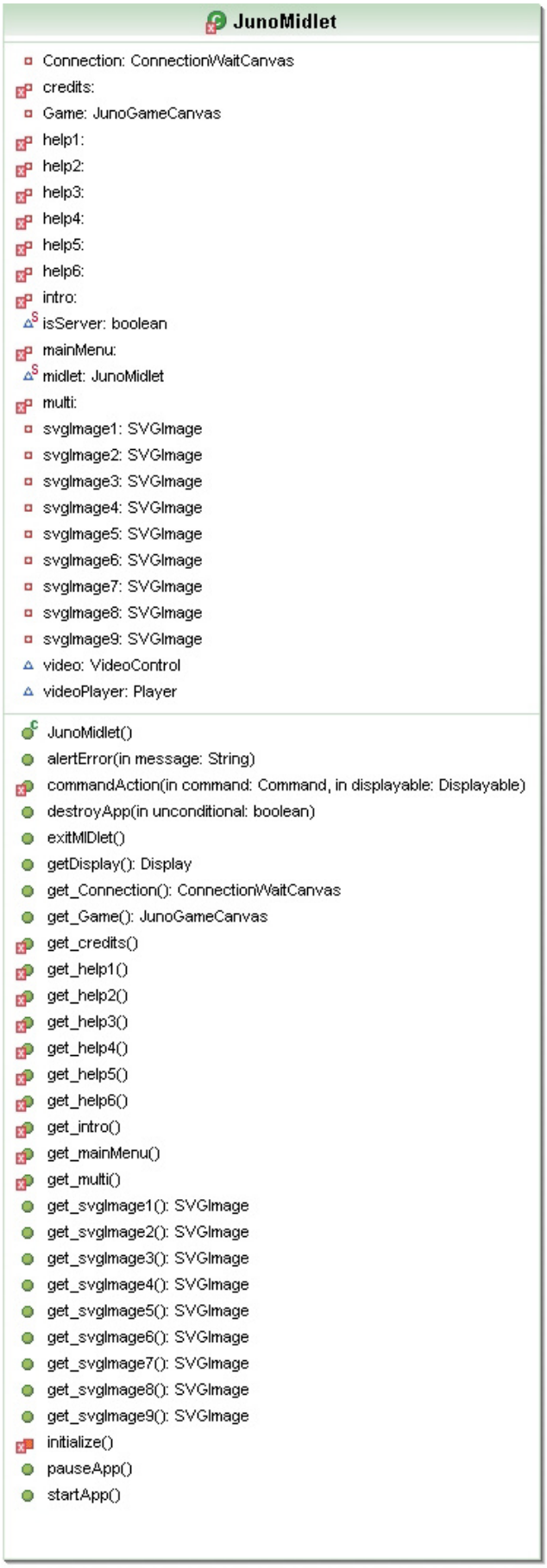
a) Les classes

▪ Diagramme des classes du projet :
(Afin d'alléger le diagramme, les relations entre classes ne sont pas représentées)




Mission
activity: boolean[]
CAN_MOVE_TO: int[]
commander: int
connected: int
destroyed: boolean
eliminate: int
isServer: boolean
NUM_ROOMS: int
phase: int
players: Player
playersReady: int
ROOM_AMBIANCE: String[]
ROOM_COORD: int[]
rand: Random
ready: boolean
turn: int
type: MissionType
votes: int[]
AIPlay(in p_id: int)
AIPlayAll()
Mission(in isServer: boolean, in type: MissionType, in connected: int)
getAliens(): int
getAlive(): int
getDestroyed(): int
getMaxVote(in p_id: int): int
getNumWait(): int
getPlayerID(in vote: int, in curr_player: int): int
getTechnicians(): int
getWin(): boolean
getWreckers(): int
initSession()
nextPhase()
tick()
Player
actions_left: int[]
alive: boolean
avatar: int
inRoom: int
MAX_ACTIONS_TYPE: int
NAMES_F: String[]
NAMES_M: String[]
NUM_AVATAR: int
NUM_NAMES: int
name: String
server_id: int
type: int
vote: int
Player(in type: int, in avatar: int, in name: String, in room: int, in server_id: int)
*
players
SoundManager
location: LocationControl[]
players: Player[]
sound3D: SoundSource3D[]
sound_enabled: boolean
spectator: Spectator
volume: VolumeControl[]
SoundManager()
createPlayer(in numero: int, in fileName: String)
createPlayer(in numero: int, in fileName: String, in prepare: boolean)
freePlayer(in numPlayer: int)
initSoundManager()
isPlaying(in numPlayer: int): boolean
play(in numPlayer: int)
set3D(in numPlayer: int)
setLoop(in numPlayer: int, in loop: boolean)
setPosition(in numPlayer: int, in x: int, in z: int)
setVolume(in numPlayer: int, in value: int)
stop(in numPlayer: int)
CustomFont
baseline: int
height: int
image: Image
style: int
width: int
CustomFont(in inImage: Image, in inStyle: int)
calculateBaseline(): int
charWidth(in ch: char): int
charsWidth(in ch: char[], in offset: int, in length: int): int
drawChar(in g: Graphics, in character: char, in x: int, in
drawCharInternal(in g: Graphics, in character: char, in
drawChars(in g: Graphics, in data: char[], in offset: int,
drawString(in g: Graphics, in str: String, in x: int, in y: in
drawSubstring(in g: Graphics, in str: String, in offset: in
getBaselinePosition(): int
getFont(in inName: String, in inStyle: int): CustomFont
getHeight(): int
getStyle(): int
isBold(): boolean
isItalic(): boolean
isPlain(): boolean
isUnderlined(): boolean
setStyle(in style: int)
stringWidth(in str: String): int
substringWidth(in str: String, in offset: int, in len: int): int


JunoGameCanvas
actionIconsImage: Image
actionToDo: int
arrow: Sprite
buttonsImage: Image
buttonsImageRed: Image
curr_frame: char
curr_vote: int
death: Sprite
end_bg: Sprite
faceIconSet: Sprite
faceIconSetImage: Image
faceIconSetImageSmall: Image
faceSet: Sprite
fnt: CustomFont
fntMini: CustomFont
GAME_HEIGHT: int
GAME_WIDTH: int
INIT_PRESENTATION_bg: Sprite
INIT_PRESENTATION_pressKey: Sprite
INIT_RADAR_pressAction: Sprite
mRun: boolean
mainPressOK: Sprite
mainPressRadar: Sprite
main_bg: Sprite
mission: Mission
moveTo: int
newPress: boolean
PHASE_TIME: int
RADAR_HEIGHT: int
RADAR_SCROLL_SPEED: int
RADAR_WIDTH: int
radarActive: boolean
radarCanActivate: int
radarImage: Image
radarScan: Sprite
radar_bg: Sprite
radar_x: int
radar_y: int
radar_zoom: float
react_id: int
SND_MUSIC: int
SND_OK: int
SOFTKEY_LEFT: int
SOFTKEY_RIGHT: int
scrollGo: boolean
scrollStart: int
selector: Sprite
selectorCommImage: Image
selectorCommImageMini: Image
selectorImage: Image
selectorImageMini: Image
ship: Sprite
shipImage: Image
shipImageSmall: Image
shipResizedImage: Image
soundWave: Sprite
timeStep: int
timer: float
VIEW_HEIGHT: int
VIEW_WIDTH: int
wait: boolean
wait2: boolean
wave_x: int
wave_y: int
wstart_frame: int
ACTION_paint(in g: Graphics)
COMMANDER_ACTION_paint(in g: Graphics)
DECISION_paint(in g: Graphics)
END_paint(in g: Graphics)
INIT_PRESENTATION_paint(in g: Graphics)
INIT_RADAR_paint(in g: Graphics)
JunoGameCanvas()
MOVE_paint(in g: Graphics)
REACT_paint(in g: Graphics)
STATS_paint(in g: Graphics)
VOTE_paint(in g: Graphics)
WAIT_PLAYERS012_paint(in g: Graphics)
clip(in img: Image, in clipX: int, in clipY: int, in width: int, in height: int): Image
drawScrollingString(in g: Graphics, in str: String, in x: int, in y: int)
initPhase()
input()
keyPressed(in keycode: int)
paintProximityRadar(in g: Graphics, in x_offset: int, in y_offset: int)
paintRoom(in g: Graphics, in room: int, in x_offset: int, in y_offset: int)
render(in gfx: Graphics)
resizeImage(in src: Image, in screenWidth: int, in screenHeight: int): Image
run()
start()
stop()
tick()
toBars(in num: int): String


JunoMidlet
Connection: ConnectionWaitCanvas
credits:
Game: JunoGameCanvas
help1:
help2:
help3:
help4:
help5:
help6:
intro:
isServer: boolean
mainMenu:
midlet: JunoMidlet
multi:
svgImage1: SVGImage
svgImage2: SVGImage
svgImage3: SVGImage
svgImage4: SVGImage
svgImage5: SVGImage
svgImage6: SVGImage
svgImage7: SVGImage
svgImage8: SVGImage
svgImage9: SVGImage
video: VideoControl
videoPlayer: Player
JunoMidlet()
alertError(in message: String)
commandAction(in command: Command, in displayable: Displayable)
destroyApp(in unconditional: boolean)
exitMIDlet()
getDisplay(): Display
get_Connection(): ConnectionWaitCanvas
get_Game(): JunoGameCanvas
get_credits()
get_help1()
get_help2()
get_help3()
get_help4()
get_help5()
get_help6()
get_intro()
get_mainMenu()
get_multi()
get_svgImage1(): SVGImage
get_svgImage2(): SVGImage
get_svgImage3(): SVGImage
get_svgImage4(): SVGImage
get_svgImage5(): SVGImage
get_svgImage6(): SVGImage
get_svgImage7(): SVGImage
get_svgImage8(): SVGImage
get_svgImage9(): SVGImage
initialize()
pauseApp()
startApp()

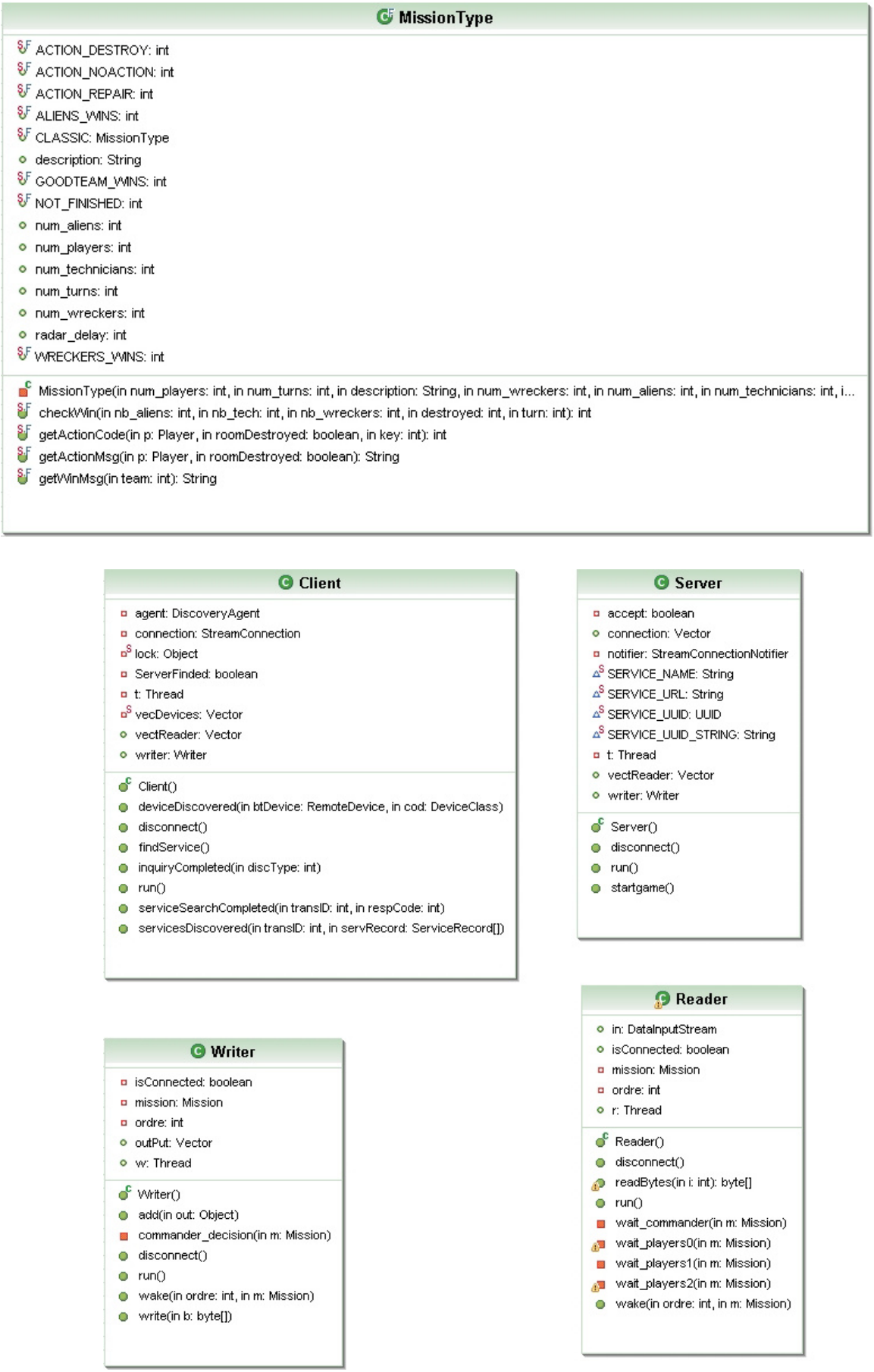


Fig. 5 : Diagramme des classes du projet

▪ Rôle de chaque classe :

- **JunoMidlet :**
  C'est la classe principale de notre midlet Java. Elle implémente les fonctions requises pour un midlet, et contient les méthodes nécessaires au fonctionnement du menu.

- **JunoGameCanvas :**
  C'est le canvas qui contient le jeu en lui-même. Cette classe gère tout ce qui se rapporte aux entrées/sorties du jeu (affichage, gestion des touches).

- **Mission :**
  Cette classe représente une partie (instance) de jeu. Elle contient tout ce qui concerne la logique du jeu, ainsi que la gestion de l'intelligence artificielle.

- **Player :** Cette classe représente un joueur, et contient toutes les informations associées.

- **PlayerType :**
  Classe représentant le type énuméré *PlayerType*, qui décrit le type de personnage d'un joueur (capitaine, alien, saboteur…).

- **MissionPhase :**
  Cette classe représente le type énuméré *MissionPhase*, qui décrit toutes les phases de jeu pour la machine à état.

- **MissionType :**
  Cette classe contient les différents types de missions (une seule à ce jour), ainsi que les paramètres (règles du jeu, textes…) associés.

- **ConnectionWaitCanvas :**
  C'est le canvas utilisé avant le début d'une partie multijoueur, pour permettre l'affichage d'information pendant la connexion entre le serveur et les clients. C'est également ici que les connexions réseaux sont initialisées.

- **SoundManager :**
  Cette classe statique constitue le manager de sons pour notre midlet, et intègre les fonctions nécessaires à la lecture des sons simples, 3D et de la musique du jeu.

- **CustomFont :**
  Cette classe contient les méthodes permettant d'initialiser et d'utiliser une police de caractère bitmap dans notre jeu, et est utilisée pour l'affichage de tous les textes.

- **Server :**
  Cette classe contient les méthodes pour créer et faire fonctionner un serveur bluetooth.

- **Client :**
  Cette classe contient les méthodes pour créer et faire fonctionner un client bluetooth.

- **Reader :** Contient les méthodes pour lire des objets envoyés.

- **Writer :** Contient les méthodes pour transformer des objects en bytes et les envoyer.

b) Description de la logique du jeu

▪ Fonctionnement global du jeu :

Le jeu organisé selon le modèle d'une machine à état (automate) :

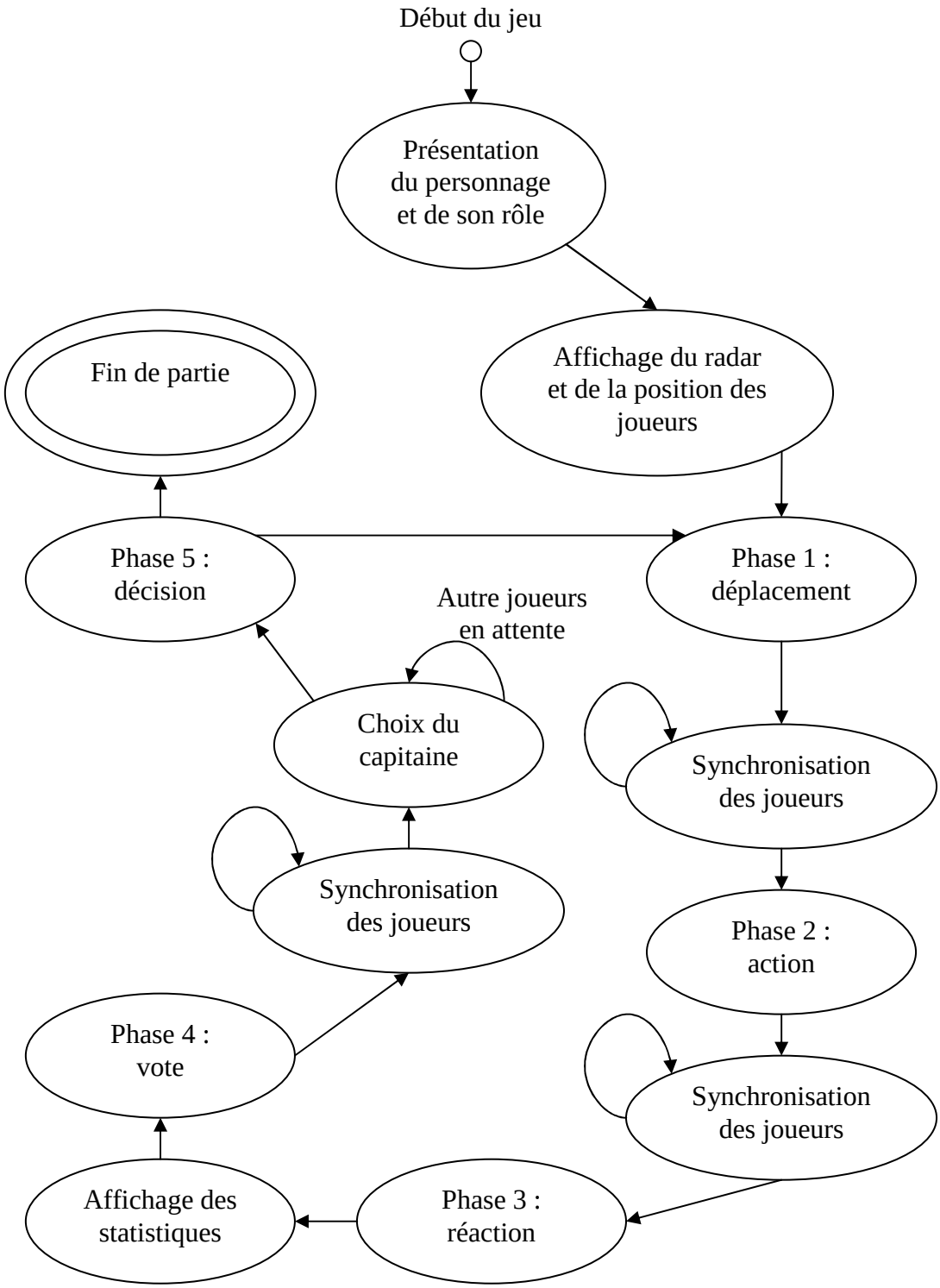


Fig. 6 : Automate du jeu

Les phases de synchronisation servent à échanger les données entre les clients et le serveur, afin que les mises à jour et information soient effectuées sur tous les mobiles. C'est également durant cette phase qu'intervient l'intelligence artificielle sur le serveur, prenant ainsi la place du joueur manquant.

Les 13 phases de jeu définies dans la classe *MissionPhase* sont donc les suivantes :

```
    public static final int INIT_PRESENTATION = 0;
    public static final int INIT_RADAR = 1;
    public static final int MOVE = 2;
    public static final int WAIT_PLAYERS0 = 3;
    public static final int ACTION = 4;
    public static final int WAIT_PLAYERS1 = 5;
    public static final int REACT = 6;
    public static final int STATS = 7;
    public static final int VOTE = 8;
    public static final int WAIT_PLAYERS2 = 9;
    public static final int COMMANDER_ACTION = 10;
    public static final int DECISION = 11;
    public static final int END = 12;
```

Au début du jeu (constructeur de la classe *JunoGameCanvas*), une nouvelle partie est initialisée en créant une nouvelle instance de la classe *Mission*. C'est cette classe qui contient toute la logique du jeu, ainsi que toutes les informations relatives à la partie courante.

Durant la partie, les actions (temps, appui sur une touche) déclenchées dans la classe *JunoGameCanvas* résultant au passage à la phase suivante, font ainsi appel à la fonction *nextPhase()* de la classe *Mission*.

C'est dans cette méthode que sont vérifiées (aux phases correspondantes) les conditions de victoire, que la fonction d'intelligence artificielle est exécutée, et que les différents paramètres relatifs à la phase courante sont initialisés.

Concernant l'affichage du jeu (classe *JunoGameCanvas*), celui est géré dans l'unique méthode *render()* et fait appel à une sous-méthode différente suivant l'état du jeu :

```
switch(mission.phase) {
    case MissionPhase.INIT_PRESENTATION: INIT_PRESENTATION_paint(g); break;
    case MissionPhase.INIT_RADAR: INIT_RADAR_paint(g); break;
    case MissionPhase.MOVE: MOVE_paint(g); break;
    case MissionPhase.WAIT_PLAYERS0: WAIT_PLAYERS012_paint(g); break;
    case MissionPhase.ACTION: ACTION_paint(g); break;
    case MissionPhase.WAIT_PLAYERS1: WAIT_PLAYERS012_paint(g); break;
    case MissionPhase.REACT: REACT_paint(g); break;
    case MissionPhase.STATS: STATS_paint(g); break;
    case MissionPhase.VOTE: VOTE_paint(g); break;
    case MissionPhase.WAIT_PLAYERS2: WAIT_PLAYERS012_paint(g); break;
    case MissionPhase.COMMANDER_ACTION: COMMANDER_ACTION_paint(g); break;
    case MissionPhase.DECISION: DECISION_paint(g); break;
    case MissionPhase.END: END_paint(g); break;
}
```

Ainsi, chaque phase dispose de sa propre fonction d'affichage.
Ce système permet à la fois d'avoir une meilleure organisation (une fonction d'affichage par état) et une plus grande rapidité d'exécution, car les clauses *case* étant rapprochées et ne

contenant qu'un appel de fonction, elles seront converties par le compilateur java en un tableau de pointeurs.

b) Description de l'intelligence artificielle

Comme expliqué précédemment, l'intelligence artificielle intervient dans la méthode *nextPhase()* ; la méthode appelée qui se charge de faire « réfléchir » les joueurs ordinateurs est la méthode *AIPlayAll()* :

```
    /** play phase for all AI-controlled alive players */
    private final void AIPlayAll() {
        for(int i=0; i < (type.num_players - connected - 1); i++)
            if (players[type.num_players-i-1].alive)
                AIPlay(type.num_players-i-1);
    }
```

Ainsi, pour tous les joueurs ordinateurs, la méthode *AIPlay()* est appelée, en fournissant en paramètre l'ID du joueur à gérer.

L'intelligence artificielle est ici simulée à l'aide d'un comportement probabiliste : nous n'avons pas chercher à simuler à l'aide d'automates la façon de jouer et de réfléchir que pourrait avoir un être humain, mais basé le comportement sur un système de probabilité, que nous avons équilibré afin de permettre à un joueur en mode solo d'avoir ses chances quel que soit son rôle dans la partie.

Les probabilités utilisées ici pour le comportement représentent ainsi le comportement observé et supposé d'un humain au cours d'une partie : par exemple, si l'on arrive vers la fin de la partie, les saboteurs vont tout faire pour détruire les pièces du vaisseau, quitte à se faire repérer, ou encore le commandant va accorder une plus grande importance à son propre vote et va suivre la majorité des voix si celle-ci est significative.

Par exemple, le système de décision pour la phase d'action d'un saboteur est le suivant :

```
    int actionToDo = MissionType.ACTION_NOACTION;
    int base;
    if (players[0].type == PlayerType.WRECKER && connected == 0)
        base = 60;
    else
        base = 50;
    if (rand.nextInt(100) < (base + (100-base) * ((float)turn /
                                                  (float)(type.num_turns-1))))
        actionToDo = MissionType.getActionCode(p, destroyed[p.inRoom], 1);
```

Si le joueur humain joue en mode solo et est lui-même un saboteur, son collègue géré par l'ordinateur aura une probabilité de base de détruire une salle plus forte, afin de l'aider à gagner. Dans tous les cas, plus l'on avancera dans la partie (nombre de tours passés), plus la probabilité de détruire la salle dans laquelle il se trouve sera grande. Ceci caractérise donc bien le comportement d'un humain : plus la fin de partie approche, moins il reste de marge pour détruire les salles, et donc plus la probabilité d'en détruire augmente afin de gagner.

Tout en étant relativement réaliste et en offrant une grande variété de situations possible (le comportement étant donc aléatoire en suivant des règles de probabilité établies), cette solution a aussi l'avantage d'être simple à mettre en œuvre et d'être très efficace en terme de performance.

Nous aurions voulu compléter ce système en rajoutant un système d'accumulation d'expérience au cours des parties, ce qui aurait conduit à affiner l'IA en lui rajoutant un facteur de comportement stratégique : par une simple analyse des actions effectuées au cours d'une partie et de leur résultat – victoire ou défaire – il suffit alors d'appliquer un bonus ou un malus à la probabilité lors du choix de l'action à un moment et une configuration donnée de la partie. Si la partie implémentation de ce système n'est pas compliquée à mettre en œuvre, la constitution de la base de données d'expérience aurait requis de pratiquer un grand nombre de parties (> 100), et ceci était impossible au niveau planning car cela nécessite beaucoup de temps et n'aurait pu se faire qu'une fois le jeu terminé, donc peu de temps avant la fin du projet.

Ce système pourrait cependant être facilement ajouté par la suite (par exemple pour les IMG Awards), étant donné que le prototype est maintenant fonctionnel.

d) Conception du système multijoueur bluetooth

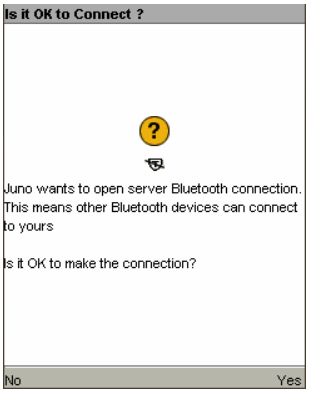
Le jeu Juno offre la possibilité de jouer jusqu'à 6 joueurs humains simulatnément pour donner plus d'intérêt et plus de convivialité. Pour les échanges entre les joueurs, nous avons choisi une connexion Bluetooth en utilisant l'API JSR82 qui offre des fonctions simplifiant les échanges bluetooth.

Lors d'une partie en multijoueur, un téléphone sert de serveur et les autres sont des clients connectés à celui-ci. Le jeu Juno se déroulant au tour par tour, différentes phases de synchronisation entre les joueurs sont mises en place et les échanges de données se feront essentiellement dans ces phases.

Lors de chaque phase de synchronisation, le serveur est en attente de recevoir les données envoyées par l'ensemble des clients et envoie ensuite les données voulues aux clients. Les clients, après avoir envoyé leurs données au serveur, sont donc en attente d'une réponse du serveur pour mettre à jour leurs données personnelles concernant la partie. Une fois ces échanges effectuées, la partie peu donc continuér.

Création d'une partie multijoueur :

Cette phase consiste à lancer le bluetooth du téléphone en mode serveur, c'est-à-dire que plusieurs appareils pourront se connecter à celui-ci, et de proposer un service Juno aux clients.
Une fois le mode serveur lancé, le téléphone est en attente de clients qui veulent se connecter et affiche à l'écran le nombre de clients qui sont prêts à vouloir jouer dans cette partie. La personne décide ensuite de lancer la partie : L'initialisation de la partie est faite côté serveur et les données d'initialisation sont envoyés aux clients.



Lancement du serveur

Ecran d'attente de clients

Un objet de classe *Server* est créé lors de la création de la partie multijoueur. Un thread sera dédié à accepter les clients qui veulent se connecter à ce serveur. Le thread enregistre un service correspondant au jeu Juno à l'adresse local de la manière suivante :

```
static String SERVICE_UUID_STRING = "C25D49733B7845EA982354ACD65FC2A4";
//service proposé
static UUID SERVICE_UUID = new UUID( SERVICE_UUID_STRING, false );
//UUID du service
static String SERVICE_NAME = "Juno";
//nom du service
static String SERVICE_URL = "btspp://localhost:" + SERVICE_UUID + ";name="
+ SERVICE_NAME;  //url où enregistrer le service
```

et dans la méthode run() :

```
notifier = (StreamConnectionNotifier)Connector.open( SERVICE_URL );
```

Ensuite, le thread attend qu'un client souhaite se connecter et accepte la connexion :

```
StreamConnection conn=notifier.acceptAndOpen();
```

A chaque connexion ouverte avec un client, un *outputStream* et un *inputStream* sont ouverts pour l'échange des données :

```
DataOutputStream out=conn.openDataOutputStream();
DataInputStream in=conn.openDataInputStream();
```

Pour l'envoi des données, un objet *writer* gère tous les *DataOutputStream*, afin que le thread s'occupe d'envoyer les données en multicast sur tous les *DataOutputStream*.

Pour la réception des données, un objet r*eader* est créé pour chaque client gérant chacun un *DataInputStream*. Il y aura dont un thread par client pour s'occuper de récolter les données reçues par ce client.

Dès que le serveur décide de lancer la partie, un tableau de 100 bytes est envoyé à chaque client pour que ceux-ci initialise la partie à l'identique de celle du serveur. Nous avons jugé que l'envoi de 100 bytes pour une initialisation était négligeable, d'autant plus que le jeu

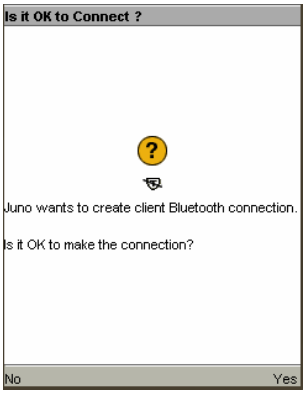
se déroule au tour par tour et donc avec beaucoup de phases où l'on attend que tous les joueurs aient fini de jouer.

Contenu du tableau envoyé :
- 1 byte pour l'identifiant du client
- 6 bytes pour les rôles des 6 joueurs
- 6 bytes pour les avatars des 6 joueurs
- 6 bytes pour les pièces où se trouvent les 6 joueurs
- 6 bytes pour l'identifiant côté serveur des 6 joueurs
- Les autres bytes pour les noms des 6 joueurs

Rejoindre une partie multijoueur :

Un objet de classe *Client* est créé. Un thread s'occupe de la recherche de tous les appareils visibles en bluetooth et recherche sur chacun d'eux si le service du jeu bluetooth est présent. Ainsi, le client trouve le serveur qui est lancé et se connecte à celui-ci.
Une fois la connexion établie, un objet *Reader* est créé pour s'occuper de récolter les données envoyer par le serveur sur un *DataInputStream*. Un objet *Reader* est créé pour s'occuper d'envoyer les données au serveur sur le *DataOutputStream* ouvert.

Attente de trouver le serveur



Ouverture de la connexion

Attente des données du serveur

Une fois la connexion établie, le client se met en attente de recevoir le tableau de 100 bytes envoyé par le serveur pour initialiser la partie.

Phases de synchronisation :

A chaque fin de phase du jeu, tous les joueurs sont en attente que les autres joueurs aient fini leur phase. Ainsi, du côté client, lorsqu'il fini une phase, il envoie ses données modifiées au serveur et attend que le client lui renvoi l'ensemble des données modifiées de tous les clients.
Du côté serveur, un thread par client s'occupe de récolter les données qu'il a modifié. Dès qu'il a recu les données de tous les clients, un thread envoie les données à tous les clients.

Les différentes phases de synchronisation :
- Attente que tous les joueurs se soient déplacés dans le vaisseau :
    2 bytes envoyés par chaque client (identifiant et numéro de la pièce où il se trouve.
    6 bytes envoyés par le serveur (pièce où se trouve chaque personne).

- Attente que tous les joueurs aient fait leur action dans leur pièce du vaisseau :
    2 bytes envoyés par chaque client (action faite dans la pièce et état de la pièce)
- Attente que tous les joueurs aient voté côté serveur :
    1 byte envoyé par le client pour désigner pour qui il vote.
- Attente de la décision du commandant côté client.
    1 byte envoyé par le serveur pour désigner qui il élimine.

On remarque que les données échangées ne sont pas excessives et ne sont pas un poids pour la fluidité du jeu Juno par rapport au mode un seul joueur.

La partie Bluetooth fonctionne correctement sur l'émulateur Sun mais nous n'avons pas eu l'occasion de pouvoir tester Juno en multijoueur sur plusieurs téléphones.

5. Design, conception et implémentation du jeu

a) Scénario initial et design du jeu

Afin de définir plus précisément un univers particulier pour notre jeu, et ainsi de pouvoir favoriser l'immersion du joueur du dans la partie, nous avons décidé d'écrire un scénario de base *(voir partie 2.a Scénario).*

Donc de manière assez rapide durant la première semaine, il a fallu créer une histoire au jeu. Après que chacun ait proposé plusieurs idées, nous avons défini un but à cette mission spatiale et les rôles des différents personnages du jeu, ainsi que leur motivation. Autour de cette base, nous avons imaginé un contexte et plusieurs anecdotes autour de celui-ci, afin de renforcer le scénario. Le design du jeu s'est donc déroulé durant la même période, car c'est l'affinage progressif des règles et du gameplay qui a inspiré les évènements décrits dans l'histoire.

Ceci nous a donc donné un réel aperçu des problèmes que peut soulever la création d'un jeu dans son ensemble, non pas seulement au niveau technique mais aussi d'un point de vue scénaristique, car le joueur doit pouvoir comprendre l'environnement dans lequel il évolue et s'y immerger. Scénario et game design sont ainsi difficilement dissociables, le premier s'appuyant sur le second, c'est pourquoi la définition stricte du gameplay et des acteurs associés doit être finalisée avant tout autre chose, de même que le scénario car c'est de celui-ci que va découler toute l'inspiration pour la partie graphique du jeu.

b) Conception des graphismes

Le graphisme de notre jeu allie un design futuriste permettant de mettre l'utilisateur dans une ambiance spatiale ainsi qu'une certaine simplicité et homogénéité lui permettant de naviguer en toute liberté. Nous avons donc utilisé pour cela des couleurs assez sombres et sobres tels que bleu et noir ainsi que des effets chromés. La plupart des dessins ont été réalisés avec le logiciel Adobe Illustrator.

Pour le jeu en lui-même, nous étions partis sur l'idée de départ de le faire entièrement SVG. Le plan de la navette a été donc conçu au départ sur Illustrator mais des problèmes

techniques (le manque de mémoire, du code assez « sale » généré par Illustrator, …) nous ont fait changer d'avis et nous avons décidé que seuls les menus utiliseraient le SVG. La navette qui a donc été conçu sur Illustrator s'est vue donc rasterisée et peaufinée sur Photoshop.

Pour les personnages du jeu, nous avons nous même dessiné les personnages à la main. Puis après les avoir scannés, nous les avons retravaillés et colorisés sur Photoshop.

Fig. 7 : Personnages du jeu

Pour les autres graphismes nous avons utilisé PaintShop Pro. Celui-ci était en effet très pratique pour la réalisation de sprites tels celui nécessaire pour simuler le clignotement des différents boutons, ou encore l'animation du bouton « ok ».

c) Design et réalisation des séquences d'animation

Dans un premier temps, nous avions pensé profiter des nouvelles fonctionnalités SVG afin de pouvoir créer nos séquences d'animation. Nous avons donc commencé à créer des animations en SVG à l'aide Adobe Illustrator (pour les dessins) et Ikivo Animator (pour l'animation) durant la première semaine, mais les premiers tests d'intégration ont montré qu'il allait être difficile d'utiliser Ikivo. En effet, ce logiciel d'animation ne permet de créer que des animations au format SVG Tiny en version 1.2, version comportant des fonctions qui ne sont malheureusement pas encore supportées avec les API disponibles sur téléphone portable que nous utilisions. Cela nécessitait donc reprendre tout le code SVG à la main, ce qui nous a amené à devoir réfléchir à une autre solution.

Nous avons donc trouvé un autre moyen de créer ces animations en utilisant le logiciel Adobe Flash, ce qui nous permettait de conserver l'aspect vectoriel des dessins, mais surtout d'exporter les animations Flash en AVI, format facilement convertible en format vidéo pour téléphone portable, par exemple 3GPP. Les premiers tests d'intégration étant concluants, nous nous sommes lancés dans la conception de l'animation d'introduction.

Un storyboard a tout d'abord été fait après avoir défini l'histoire (scénario) du jeu :

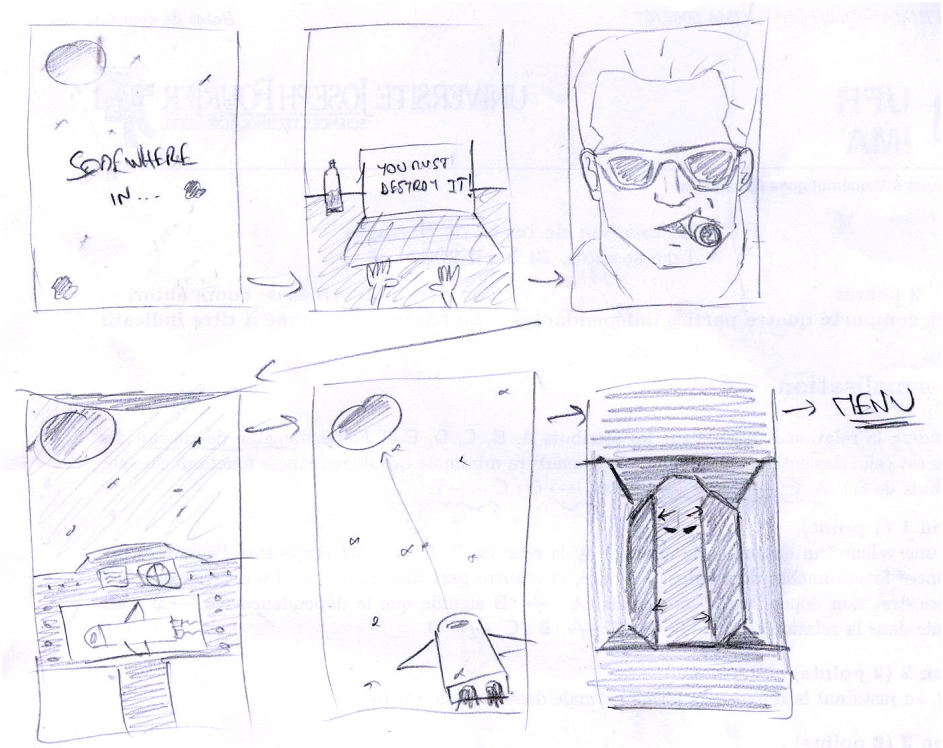


Fig. 8 : Storyboard de la séquence d'introduction

Une fois le storyboard défini, il a fallu passer à la réalisation. Cette partie du projet a été très intéressante puisqu'il nous a fallu inventer toute une ambiance visuelle au jeu, en insérant des petits détails permettant de s'immerger dans les différentes scènes, tout en devant rester simple dans les graphismes. La création d'une animation de ce genre nous a permis de mettre en œuvre et de mélanger des notions d'esthétisme et des notions cinématographiques.

Une fois la séquence d'animation d'introduction terminée, nous avons ensuite utilisé le logiciel audio Samplitude pour créer la bande son. Aucun problème particulier ne fut à noter. Les sons utilisés ont été extraits d'une banque de sons libre. Nous avons aussi du créer nous-même quelques sons, comme des voix, car il était difficile de trouver des sons déjà fait correspondant bien à la scène à sonoriser. La partie technique de ce son a donc principalement été portée sur la synchronisation du son avec la vidéo.

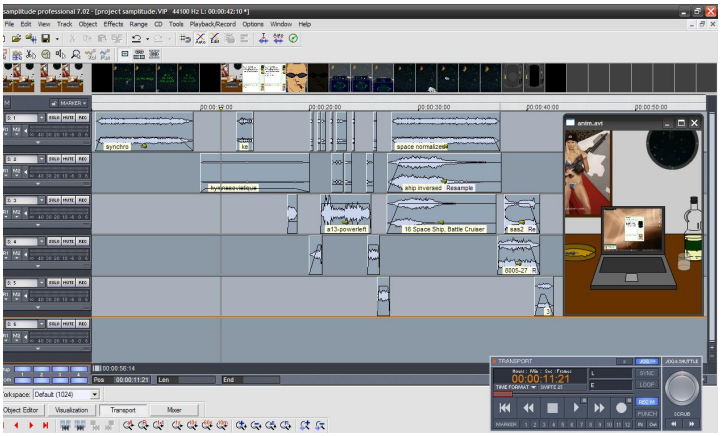


Fig. 9 : Synchronisation de la bande son avec la vidéo à l'aide du logiciel Samplitude

L'interface de Samplitude nous a permis de synchroniser facilement les sons et la vidéo, et d'exporter le tout en format vidéo AVI. De plus, toutes les bandes son ont été par la suite masterisées sous un autre logiciel audio (Wavelab) afin d'égaliser les niveaux sonore.

L'animation d'introduction fut totalement terminée une semaine avant la fin du projet, nous avons donc par la suite décidé de créer plusieurs petites séquences d'animation supplémentaire pour la fin de la partie. De même que pour l'introduction, ces animations ont été faites en Flash et les bandes son mixées et synchronisées avec Samplitude. Ce jeu comporte ainsi 4 fins différentes suivant le dénouement de la partie et le rôle du personnage joué.

d) Conception des menus SVG

Nous avons décidé de réaliser nos menus en SVG Tiny 1.1. Pour cela, nous avons réalisé le design des menus sur Illustrator qui permet de faire très simplement des graphismes vectoriels, puis de les enregistrer sous format SVG Tiny 1.1.

Une fois le design de chaque menu finalisé, nous avons créé les animations à la main dans le code SVG. Nous sommes basés sur des exemples fournis par le Mobility Pack de Netbeans.

Ainsi, nous avons fait deux animations pour chacun des *items* de nos menus. Une pour l'item qui est entouré de blanc lorsque qu'il est sélectionné et noir sinon, puis une autre animation pour le texte d'un item. Pour cette dernière animation, le texte se met à grossir lorsque l'on passe sur l'item.

En sus de ces deux animations, nous en avons réalisé d'autres pour nos menus, tels que des translations de texte pour le menu *credits* et une translation d'un panneau contenant tous les items pour l'écran *mainMenu*.

Code pour l'animation d' un item (changement de la couleur du contour) :

```
<path id="menuItem_x5F_5_x5F_quit" fill="#262763" stroke="#000000" stroke-width="2"
d="M166.75,252.008 c0,5.883-8.348,10.653-18.646,10.653H61.093c-10.297,0-18.645-4.771-
18.645-10.653v-1.775c0-5.879,8.348-10.648,18.645-10.6
h87.011c10.301,0,18.646,4.77,18.646,10.648V252.008L166.75,252.008z">
        <animate  fill="freeze" values="rgb(0,0,0);rgb(250,250,250)" dur="0.5s"
begin="menuItem_x5F_5_x5F_quit.focusin" attributeName="stroke" calcMode="linear"
accumulate="none" additive="replace" restart="always"
end="menuItem_x5F_5_x5F_quit.focusout">
        </animate>
        <animate  fill="freeze" values="rgb(250,250,250);rgb(0,0,0)" dur="0.5s"
begin="menuItem_x5F_5_x5F_quit.focusout" attributeName="stroke" calcMode="linear"
accumulate="none" additive="replace" restart="always"
end="menuItem_x5F_5_x5F_quit.focusin">
        </animate>
</path>
```

Nous avons ici pour l'animation de l'item deux balises *animate*. Pour la première, les valeurs des contours (*attributeName="stroke"*) vont passer du blanc au noir correspondant (en vert plus dans le code). Cet évènement commence lorsque l'on se place sur l'item *Quit*, c'est-à-dire lors de l'évènement *focusin*. Puis l'évènement s'arrête ensuite lorsqu'on que l'on sélectionne un autre item, c'est-à-dire lors de l'évènement *focusout*.
Pour le deuxième *animate*, il s'agit simplement de l'évènement inverse. Donc lors de l'évènement *focusin* le contour de l'item passe du blanc au noir et cela s'arrête lors de l'évènement *focusout*.

Code pour l'animation du texte d'un item (effet de grossissement) :

```
<text transform="matrix(1.0476 0 0 1 88.2183 255.9268)" fill="#FAFAFA" font-
family="'Arial-Black'" font-size="18">Quit
        <animate  fill="freeze" values="18;22" dur="0.5s"
begin="menuItem_x5F_5_x5F_quit.focusin" attributeName="font-size" calcMode="linear"
accumulate="none" additive="replace" restart="always"
end="menuItem_x5F_5_x5F_quit.focusout">
        </animate>
        <animate  type="font-size" fill="freeze" values="22;18" dur="0.5s"
begin="menuItem_x5F_5_x5F_quit.focusout" attributeName="font-size" calcMode="linear"
accumulate="none" additive="replace" restart="always"
end="menuItem_x5F_5_x5F_quit.focusin">
        </animate>
</text>
```

Pour l'animation du texte de l'item, il a aussi besoin de deux balises *animate*. L'attribut qui sera modifié ici est la taille de police (*attributeName="font-size"*). Donc, lors de l'évènement *focusin* la police passe d'une taille de 18 à 22 et lors d'un évènement *focusout* la police passe d'une taille de 22 à 18.

Voici le résultat que l'on obtient grâce aux codes décrits plus haut. A gauche, l'item n'est pas sélectionné donc le contour est noir et le texte est de taille 18. Puis à droite on voit que l'item est sélectionné par conséquent le contour devient blanc et la taille de police du texte Quit passe à 22.

Fig. 10 : Aperçu de l'animation SVG d'un item du menu

Une fois nos menus SVG terminés, il ne nous restait plus qu'à lier les menus entre eux. Pour le faire, nous avons utiliser NetBeans. En effet, NetBeans propose un outil cité plus haut, à savoir le Mobility Pack qui permet très simplement de manière graphique de lier tous nos menus en générant le code automatiquement. Voici un aperçu de l'environnement de travail :

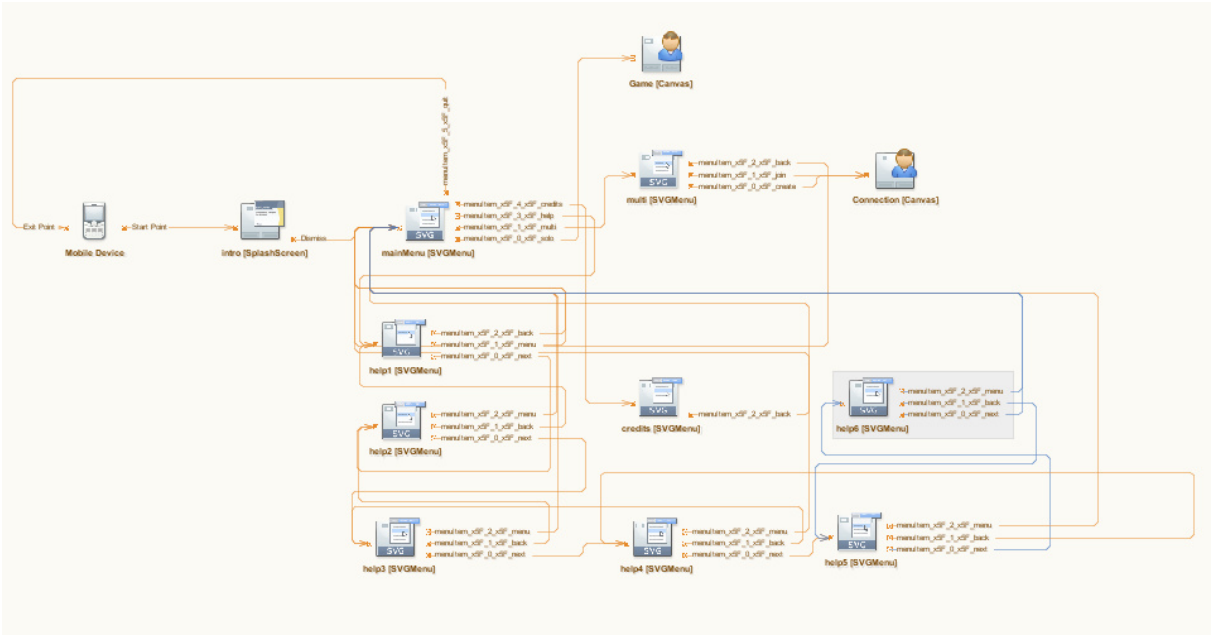


Fig. 11 : Flow design des menus sous NetBeans

Les limites des menus SVG sur téléphone portable.

Comme décrit précédemment, le NetBeans Mobility Pack est un outil utile et plutôt intuitif dans la réalisation de menus SVG sur téléphone portable. L'émulateur un peu lent mais assez complet nous a permis, lors de ce projet d'émuler des menus SVG sans trop de problèmes. Cependant, l'émulateur ne tiens pas compte de la quantité de RAM d'un téléphone ce qui a été ennuyeux lors de l'intégration de notre jeu sur le S60 de Nokia. En effet, des exception "out of memory", non détectée lors de l'implémentation et de l'émulation ont engendré problèmes lors des tests sur téléphone réel. Afin de mieux cerner ce phénomène, nous avons donc utiliser une des options de l'émulateur, le moniteur de mémoire.

Le moniteur de mémoire est un outil pratique avec lequel il est possible de visualiser un graphique représentant l'utilisation de la mémoire. Cet outil nous a ainsi permis de comprendre cet excès de mémoire que nous n'avons néanmoins pu résoudre qu'à moitié en faisant une version « light » de nos SVG pour le menu.

e) Design de l'interface utilisateur du jeu

Afin d'adapter au mieux l'interface du jeu à la plate-forme et au gameplay défini, nous avons effectué une étude préliminaire en nous basant sur les méthodes apprises cette année.

- Contexte d'usage et modèle de qualité

Profil utilisateur

- Données générales :
Homme ou femme, sans limite d'age, comprenant la langue anglaise, pouvant tenir un téléphone portable.

- Compétences métier (niveau d'expertise dans son métier):
Aucune connaissance métier particulière requise.

- Compétences informatiques :
Notions dans l'utilisation d'un téléphone portable.

Modèle de plate-forme

- IHM centralisées :
Téléphones portables récents supportant SVG, vidéo, et son 3D.

- IHM distribuées :
Téléphone portable → Téléphone portable (partie multijoueur)

Modèle d'environnement

- Milieu physique et milieu social :
- Domicile du joueur (peu de bruit, peu de personnes)
- Voiture (assez bruyant : radio, conducteur du taxi)
- Rue (très bruyant, beaucoup de monde)
- Lieux publics (bruyant, présence de personnes)

Qualité

On fixe des priorités dans les critères d'ergonomie :

- Homogénéité / Cohérence :
Dans notre jeu, nous utilisons toujours la même manière de faire lorsque l'on veut exécuter une tache. L'interface est similaire quelque soit la tache à effectuer.

- Guidage :
. *Incitation :*
L'utilisateur est guidé tout au long de la partie.
. *Groupement / distinction entre items :*
Le menu est fait de telle sorte que les actions différentes sont bien regroupées.
. *Retour d'information immédiat :*
L'utilisateur est informé dès qu'il effectue une action. L'utilisateur est également informé à tout moment ou il se trouve dans le déroulement de la partie.
. *Lisibilité :*
L'écran d'un téléphone portable étant assez petit, on veut une lecture aisée.

- Charge de travail
. *Brièveté, concision dans les actions :*
L'utilisateur choisi rapidement ce qu'il veut faire, il ne lui faut pas beaucoup d'action pour réaliser sa tâche. De plus, le jeu pouvant se jouer en multi-joueurs, chaque joueur à une limite de 30s pour effectuer une action.
. *Signification des codes :*
L'utilisateur retrouve des icônes en adéquation avec des instructions défilantes pour effectuer différentes tâches.

. *Contrôle des erreurs :*

L'utilisateur étant totalement guidé tout le long de la partie, il ne peut donc pas se perdre ou effectuer des actions non conformes à la politique du jeu.

- Modèle des tâches

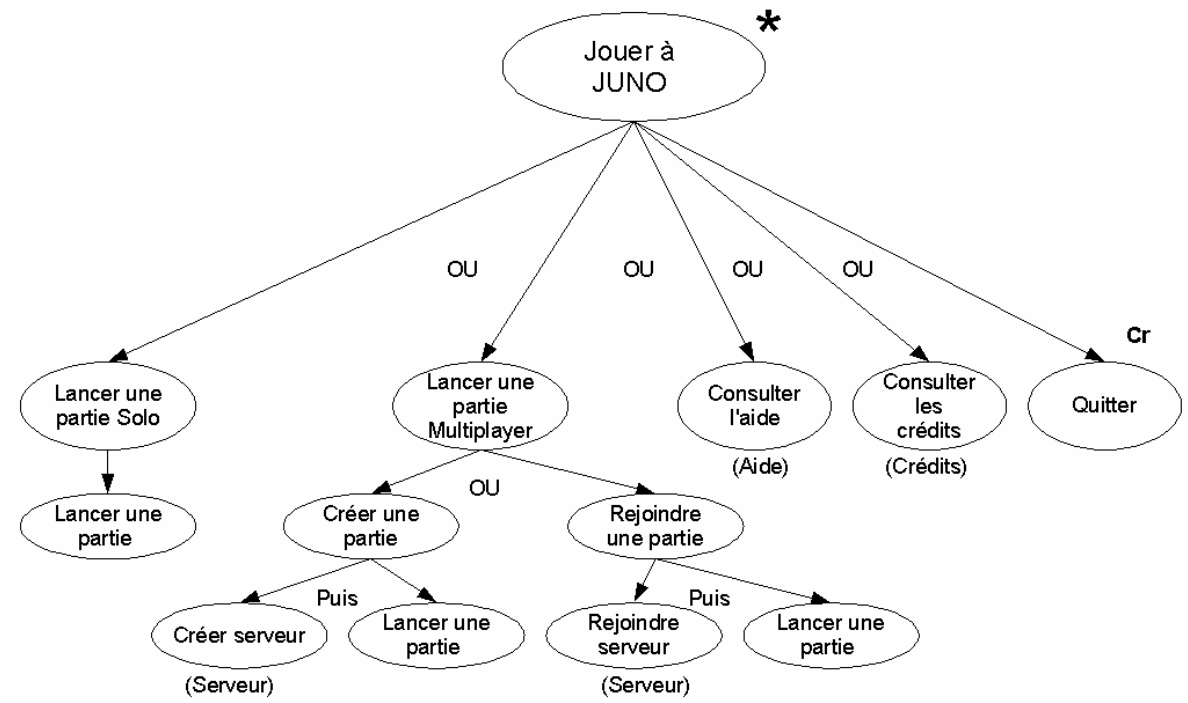


Fig. 12 : Modèle des tâches « jouer à Juno »

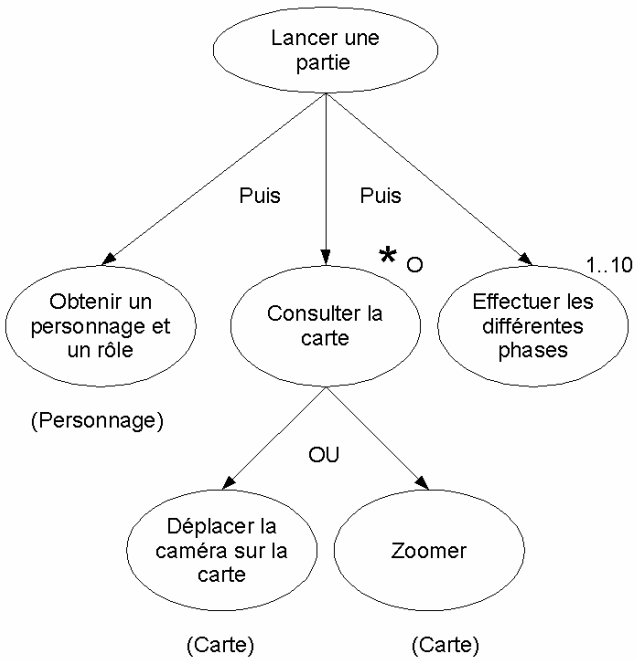


Fig. 13 : Modèle des tâches « lancer une partie »

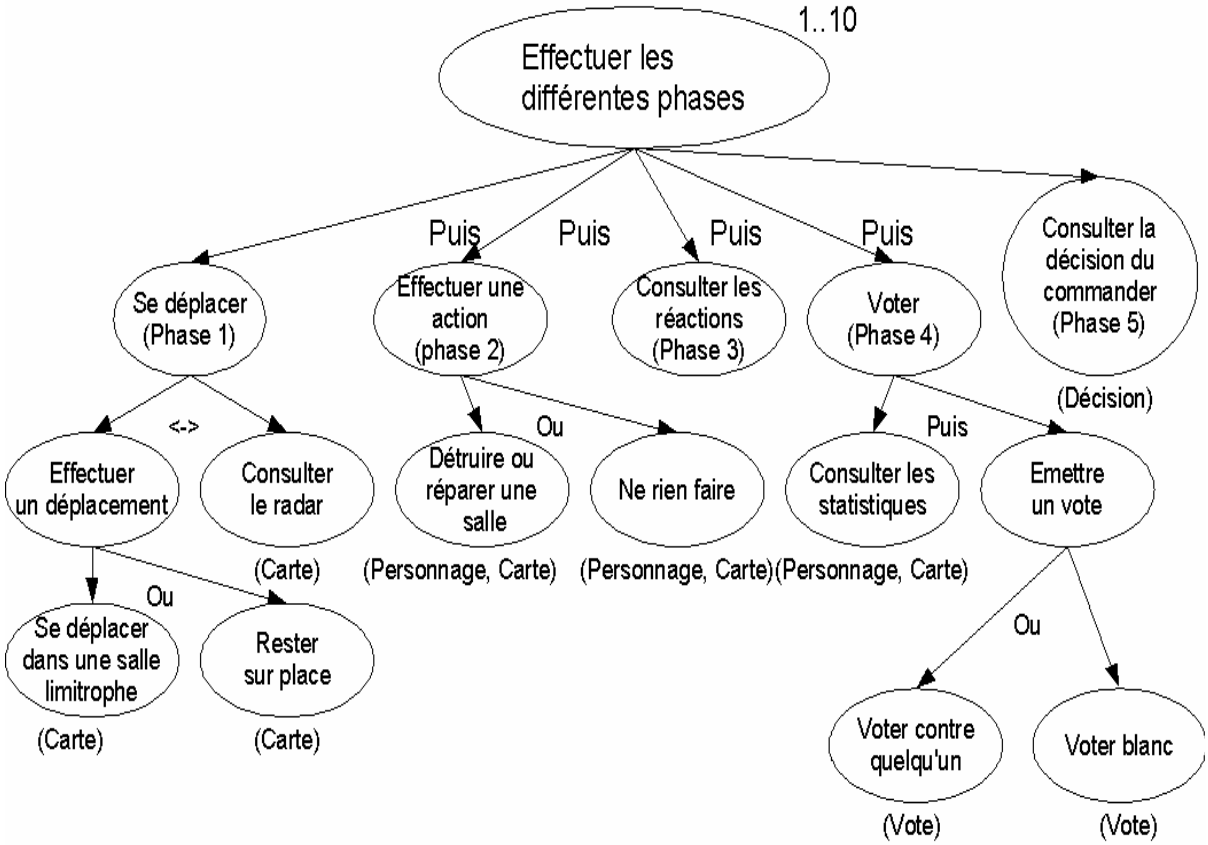
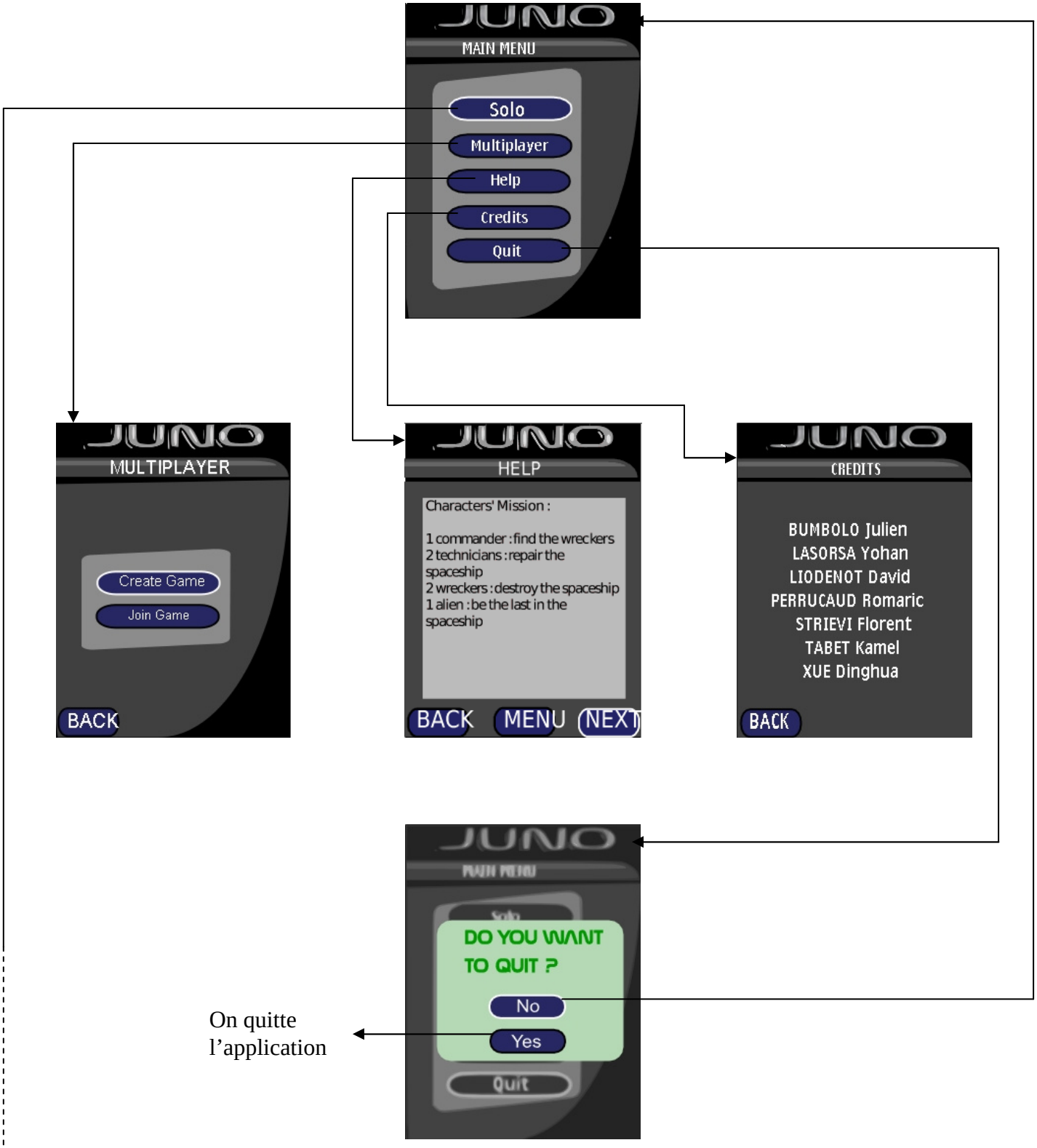


Fig. 14 : Modèle des tâches « effectuer les différentes phases »

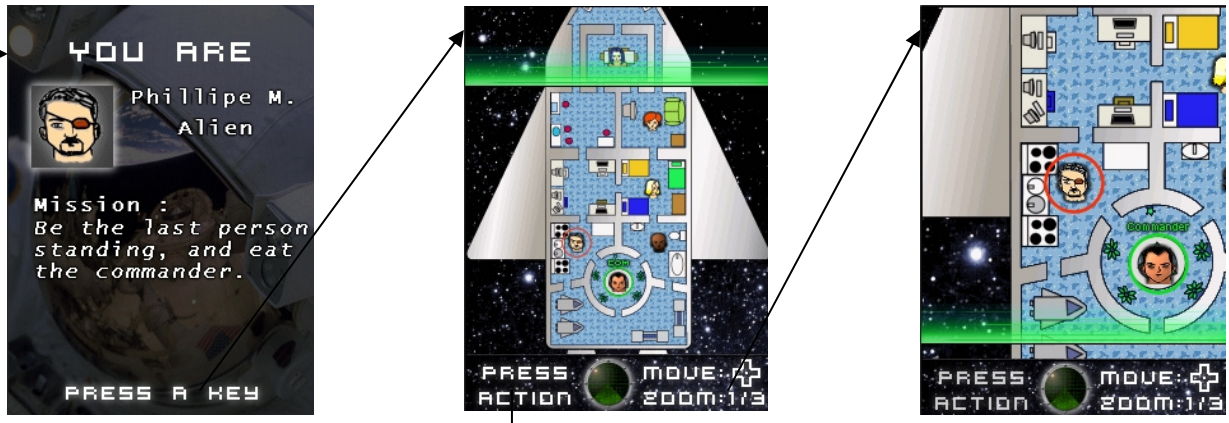
▪ Maquette

Jouer à JUNO :

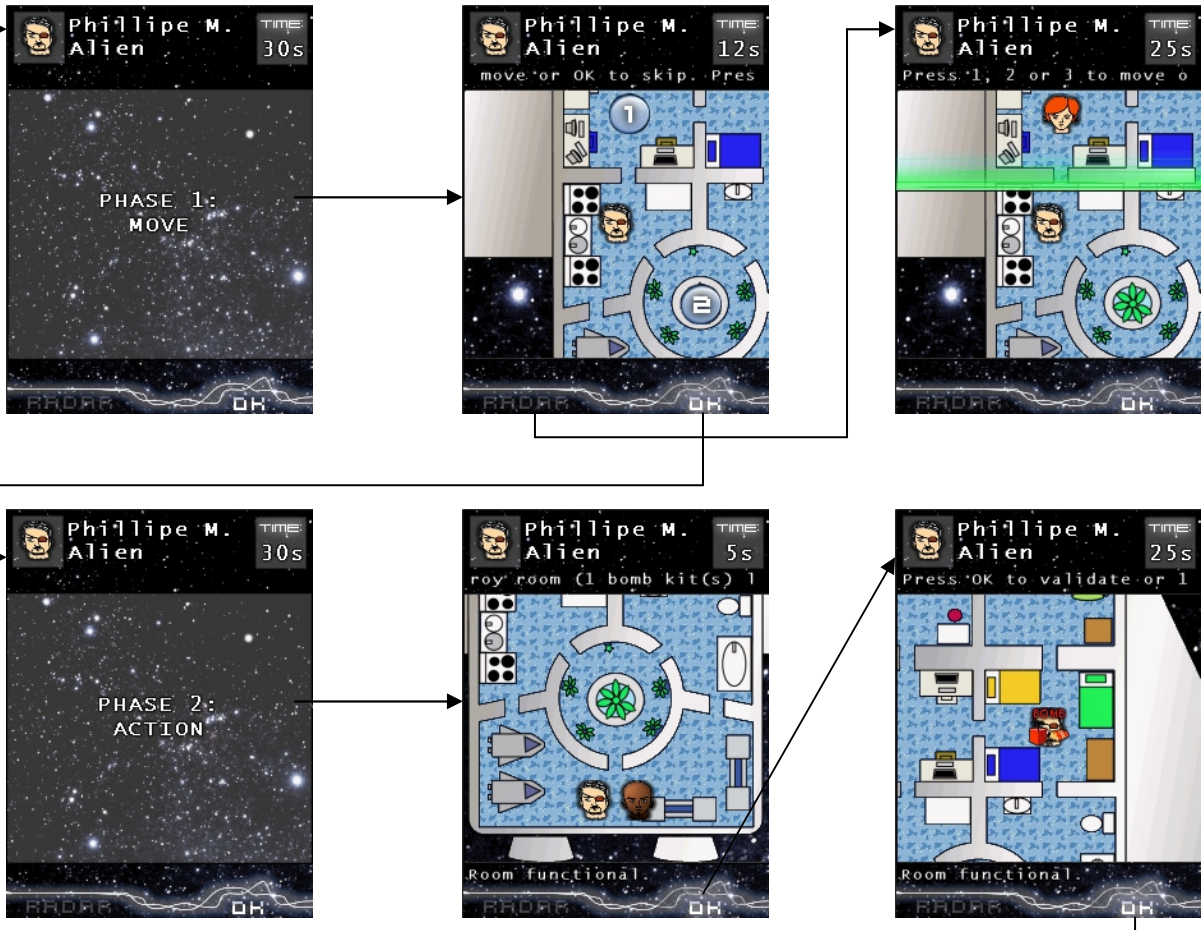


On quitte l'application

Lancer une partie :



Effectuer les différentes phases :




Phillipe M.
Alien
TIME
--
PHASE 3:
REACTION
RADAR
OK
Tony P.
Alien
TIME
--
Look and listen carefully
RADAR
OK
Phillipe M.
Alien
TIME
30s
PHASE 4:
VOTE
RADAR
OK
Phillipe M.
Alien
TIME
30s
GATHERING
STATISTICS
RADAR
OK
Phillipe M.
Alien
TIME
29s
Press OK to continue.
Statistics :
Turn : 4/10
People on ship : 5/6
Rooms functional : 7/10
Rooms destroyed : 3/10
RADAR
OK
Phillipe M.
Alien
TIME
26s
yer to vote against, then
Blank vote
Kamel S.
Maryse B.
Clara M.
Jean Mike T.
RADAR
OK
Phillipe M.
Alien
TIME
15s
PHASE 5:
DECISION
RADAR
OK
Phillipe M.
Alien
TIME
13s
Press OK to continue.
Decision :
Nobody was choosen.
You are safe...
RADAR
OK
THE END
You were ejected
from the ship...
OK
Fin de la
partie !

f) Design des sons d'ambiance et FX audio

▪ Description de l'espace sonore :

Juno est un jeu où le son tient une place importante. Afin d'offrir une certaine profondeur au jeu, nous avons décidé d'intégrer à la fois un environnement sonore 3D (phase 3) mais aussi 2D pour tout le reste du jeu.

Ainsi, le jeu dispose d'un système sonore riche, mêlant aussi bien musiques d'ambiances (propres à chaque salle du vaisseau), sons spatialisés en 3D, musiques de menus etc…

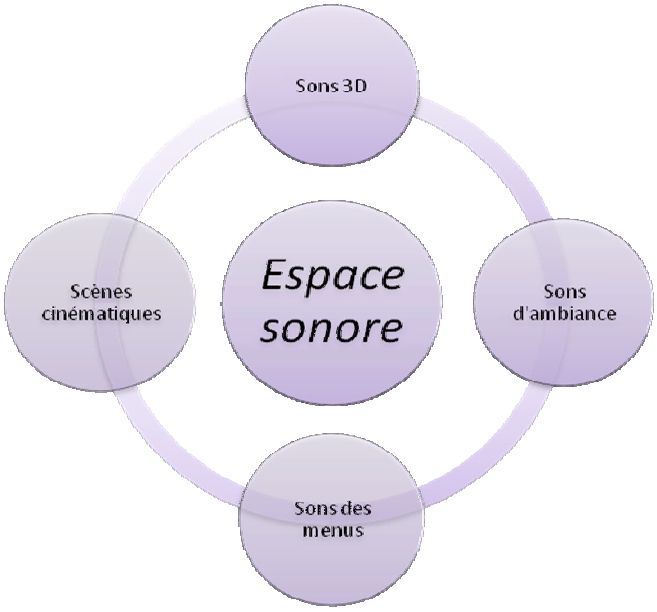


Fig. 15 : Espace sonore du jeu

En nous basant sur le système vu dans les TPs, nous avons implémenté une classe *SoundManager*. Cette classe offre différentes primitives de manipulation de sons, telle la création d'un player, la lecture d'un son …
Nous verrons plus tard que cette classe possède quelques particularités, conséquences d'une optimisation du code, compte tenu des capacités limités de la plateforme accueillant notre jeu.

▪ Design des FX sonores et spatialisation 3D:

Comme nous avons pu le voir précédemment, la partie sonore du jeu est gérée au sein de la classe *SoundManager*. Celle-ci possède 4 tableaux en attributs, tableaux contenants les différents players ainsi que leur contrôles :

```
static private Player players[];
static private SoundSource3D sound3D[];
static private Spectator spectator;
static private VolumeControl volume[];
static private LocationControl location[];
```

players : contient les différents players permettant de lire un son.

sound3d : contient les environnements sonores 3D pour spatialiser les sons nécessaires.
location : contient les différents contrôles de positionnement des sources sonores dans leurs environnements 3D.
spectator : le spectateur est initialisé à la position (0,0,0), c'est-à-dire au centre de l'espace sonore. Les sons seront ensuite placé "autour" de lui.
volume : comme son nom le laisse deviner, ce tableau contient les différentes contrôles de volume sonore liés à chaque player.

Nous avons décidé d'implémenter nos différents sons sous la forme d'un tableau de players pour plusieurs raisons :

- Notre volonté d'offrir une sensation d'immersion aux joueurs nous a poussé à créer un environnement sonore riche. De ce fait, de nombreux sons peuvent être joués en même temps.
- Afin d'optimiser les performances de notre jeu, l'utilisation de manière statique de la classe *SoundManager* nous permet de limiter l'occupation en ressources de la partie sonore.

Sachant que le maximum de sons devant être prêts à être joué à un moment donné ne dépasse jamais 8, nous instancions donc nos tableau pour une taille de 8.

```
players = new Player[8];
volume = new VolumeControl[8]; […]
```

Primitives de son :

La gestion des sons se fait facilement à l'aide des quelques fonctions explicites suivantes :

```
    /** initialize the sound manager */
    public final static void initSoundManager()
    /** create or replace one of the players */
    public final static void createPlayer(int numero, String fileName)
    /** set the player 3D */
    public final static void set3D(int numPlayer)
    /** set the 3D position of a player */
    public final static void setPosition(int numPlayer, int x, int z)
    /** start a player */
    public final static void play(int numPlayer)
```

D'autres methodes permettant de consulter le statut d'un player sont aussi implémentées.

Voici un exemple de création de 2 players, l'un spatialisé et l'autre représentant une musique de fond :

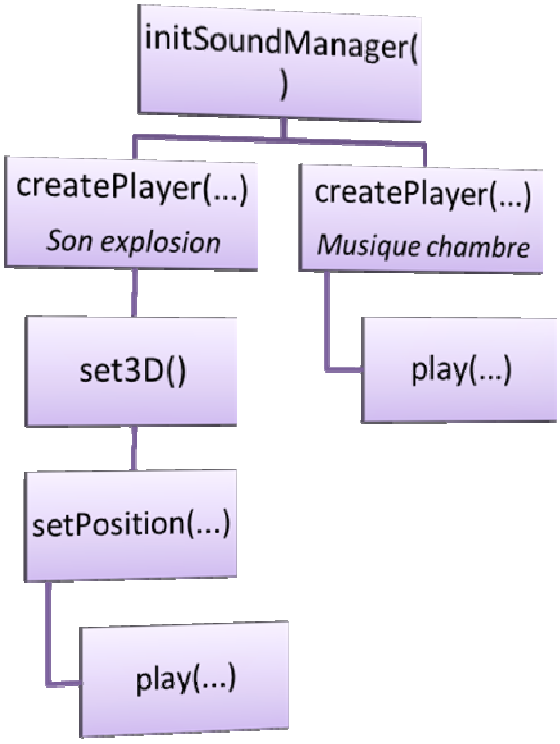


Fig. 16 :Exemple d'instanciation d'un son 3D et d'une musique

Comme on peut le voir sur cette figure, après la création d'un player, il possible de lui associer ou non un environnement 3D.
Dans le cas où l'on décide d'associer un environnement 3D à un son, on doit ensuite lui donner une position dans cet environnement avant de pouvoir lire le son.
Ce système de positionnement 3D est uniquement utilisé dans la phase 3 (Phase 3 : Réaction), lorsque le joueur entend les différentes actions réalisées dans sa salle ou les salles proches.

Le déclenchement des différents sons se fait au sein de la classe *JunoGameCanvas* qui gère en grande partie la logique du jeu.
L'utilisation de ces fonctions étant triviale, nous vous invitons à vous reporter au code contenu en annexes pour obtenir de plus amples informations sur la mise en place du système sonore du jeu.

▪ Optimisations

Compte tenu du type de plateforme sur lequel tourne notre jeu, nous nous sommes retrouvés confronté à de nombreuses contraintes. En particulier des contraintes liées aux performances de la machine.
La où n'aurions pas rencontré de problème sur une machine plus puissante telle un ordinateur ou encore une console, nous avons dû porter une attention toute particulière à l'optimisation de notre code.

Au niveau de la partie sonore, l'optimisation s'est faite sur 4 points principaux :

- Modification de la classe *SoundManager*
- Suppression/création dynamique des players au cours de la partie
- Spatialisation sélective

- Mastering des sons

La première optimisation s'est faite en modifiant profondément la structure de la classe *SoundManager*. Toutes les méthodes sont alors déclarées en tant que `final static.`
De même les players ne sont plus créés lors de l'instanciation de *SoundManager* mais seront créés suivant les besoins du jeu à un moment donné.
En se faisant, nous n'instancions alors plus d'objet *SoundManager*. Le constructeur de celui-ci est déclaré *private* afin d'en empêcher l'instancisation. La classe est donc utilisable uniquement de manière statique.
L'initialisation du système est délégué à la fonction :

```
public final static void initSoundManager()
```

D'un point de vue sémantique, on se rapproche ici plus d'un langage comme le C que d'un langage typé objet.

La seconde optimisation a consisté à donner la possibilité à la logique de jeu de supprimer les players lorsque ceux-ci ne sont pas utilisés. En effet, en testant sur le N95, nous nous sommes aperçus que l'utilisation d'un trop grand nombre de players en même temps entraînait un redémarrage du portable.

Nous avons alors ajouté la méthode :

```
public final static void freePlayer(int numPlayer)
```

De cette manière, nous avons pu diminuer l'occupation mémoire de notre jeu afin de rester dans une utilisation raisonnable compte tenue de l'occupation mémoire des autres modules de notre jeu.

La troisième optimisation a déjà été évoquée dans la partie précédente. Elle a consisté à ne spatialiser que les players nécessaires, au moment où ceux-ci sont utilisés. En effet la machine limite le nombre de players pouvant être spatialisés en même temps.
Au delà d'un simple problème de performance, ce choix a été contraint vis-à-vis des limites de la machine supportant notre jeu.

On remarque bien ici l'importance de la connaissance de la machine sur laquelle doit tourner un projet. Alors que nous avions déjà commencé le développement du jeu, nous n'avions pas décelé cette limitation vis-à-vis du son 3D. Dans le cadre de ce projet, l'erreur a rapidement pu être corrigée mais dans le cadre d'un projet de plus grande envergure, cette restructuration du code aurait pu entrainer un surcout non négligeable.

Enfin la dernière optimisation a été réalisée non pas au niveau du code de notre application mais au niveau du mastering des ressources (ici les sons, mais aussi des images, vidéos etc…).
Le N95 ne gérant pas le streaming de l'audio depuis la carte mémoire, la totalité de nos sons sont chargés dans le téléphone et ce, dès le lancement de la midlet. Afin de ne pas saturer la mémoire dévolue aux sons, nous avons alors décidé d'encoder ceux-ci dans différents formats, compte tenu de la nature du son (musique, bruit d'ambiance …) et de la durée de ceux-ci.

Les différents formats retenus ont étés les suivants :

- MP3 32kbits/s mono pour les musiques
- AMR 12kbits/s pour les fx sonores
- AAC pour les vidéos d'introduction et de fin (3GP pour le conteneur)

Nous obtenons ainsi une taille totale pour les fichiers audio de moins de 1.5Mo (sans tenir compte des séquences vidéos).

Enfin, il est à noté que la musique du menu ainsi que différents sons des vidéos ont été réalisés par nos soins. De même pour la plupart des sons d'ambiance.
Les autres sons (musiques mises à part) sont libres de droits et ont été récupérées sur internet.

g) Optimisation

La plateforme cible (téléphone portable) disposant d'une quantité de mémoire et d'une capacité de calcul réduite, il a fallu tenir compte de ce facteur lors du développement des différentes parties de ce projet.

Dans plusieurs domaines, les contraintes dues à cette plateforme nous ont conduit à optimiser aux mieux les différentes parties du code et ressources.

Au niveau de la vitesse d'exécution, notre jeu se jouant au tour par tour, et ne faisant pas appel à d'intenses phases d'actions, la charge processeur est plutôt réduite et ne nécessitait pas d'optimisation particulière, au vu des résultats observés sur le *profiler*.

Nous avons cependant pris la peine de déclarer *final* toutes nos méthodes et d'utiliser des méthodes statiques lorsque c'était possible, afin de gagner en mémoire et en vitesse.

La partie d'optimisation s'est donc principalement portée sur la mémoire : en effet notre jeu comporte de nombreux éléments audio et visuels (sprites, sons, musiques…) et ceux-ci occupent une quantité de mémoire importante. Pour les sprites nous, avons réduit la profondeur de couleur à 256 couleurs (8 bits) au lieu de 24 bits, ce qui a réduit de manière significative l'occupation mémoire, sans perdre beaucoup de qualité. Pour les sons, les FX ont été compressés au format AMR et les musiques en MP3, comme expliqué dans une partie précédentes.

Au niveau du code, la mémoire est gérée au plus strict, les allocations d'objets on été réduites au maximum, et des *garbage collection* sont effectuées fréquemment pour libérer la mémoire lorsque cela est nécessaire.

Ainsi, nous avons pu garder une occupation mémoire réduite et stable lors du déroulement d'une partie, comme l'on peut l'observer sur ce graphe affiché par le *memory monitor* :

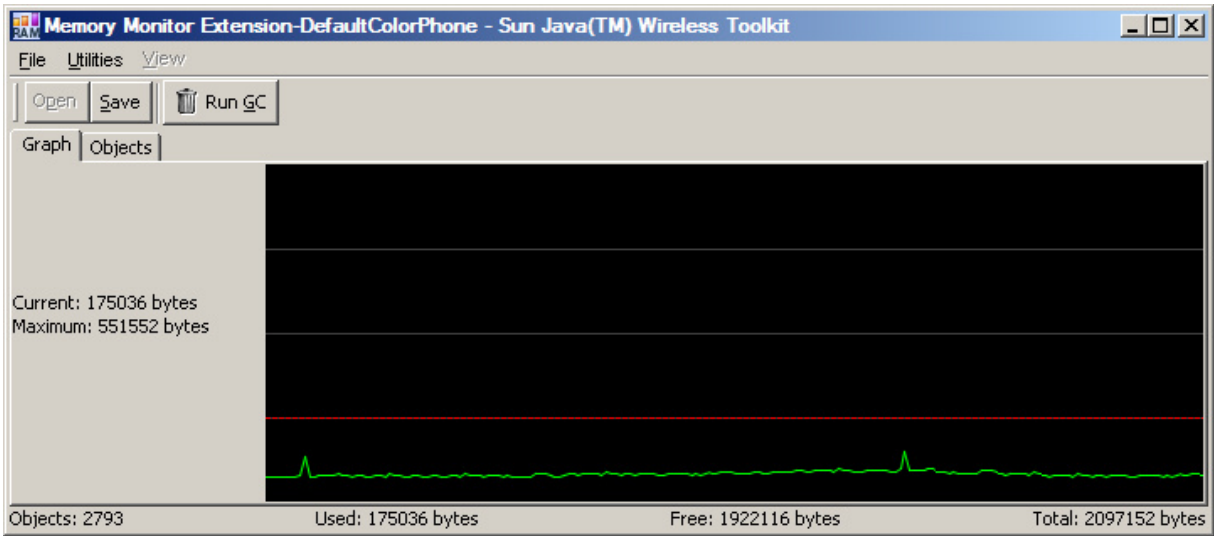


Fig. 17 : Occupation mémoire au cours d'une partie

Comme l'on peut le voir sur le graphe précédent, l'occupation mémoire se maintient en dessous de 200Ko de RAM, ce qui reste très raisonnable.

6. Conclusion

Le jeu Juno est arrivé à une phase de prototypage bien avancée. En effet, il est possible à ce jour de jouer une partie solo sur le téléphone N95 sans aucun problème. Cet état avancé du jeu est dû à une réflexion collective sur les objectifs à atteindre, un groupe hétérogène où chacun a pu apporter des compétences différentes et dialoguer pour un résultat final cohérent.

Le principal problème rencontré est la difficulté de travailler sur des émulateurs qui ne gèrent pas l'ensemble des technologies que nous souhaitions utiliser. En effet, l'émulateur S60 ne peut pas lire les vidéos ou jouer plusieurs sons en même temps. De plus, il est très long à lancé et donc désagréable à utiliser. L'émulateur Sun ne gère ni les vidéos ni les sons MP3. De plus le son est saccadé et le jeu est très ralenti. Cependant il est le seul sur lequel nous avon pu tester la partie bluetooth. Quant à l'émulateur Sony Ericsson, il ne prend pas en compte ni le SVG, ni le son 3D.

Malgré ces difficultés, nous sommes cependant arrivés à un résultat final où l'ensemble de nos technologies fonctionne sur le téléphone N95. Nous n'avons cependant pas pu tester la partie multijoueur gérée par le bluetooth car nous ne disposions pas de plusieurs téléphones.

Pour finir, nous sommes satisfaits d'avoir atteint notre objectif initial, à savoir la conception et la réalisation d'un jeu complet utilisant des technologies avancées telles que le SVG, le son 3D, le bluetooth ainsi qu'une intelligence artificielle avancée, tout en ayant réussi à s'adapter aux problèmes liés à ces nouvelles technologies qui sont encore mal gérées par les émulateurs et les mobiles.