Projet Informatique, Quake

Stéphane Lescuyer Julien Mairal

juin 2004

FIG. 1 – Snapshot du jeu

Introduction

Plus de 10 ans après la sortie de Doom en 1993, un des premiers jeux en 3D faisant évoluer un personnage dans un labyrinthe peuplé de créatures hostiles, les doom-like et autres quake-like fleurissent toujours sur le marché du jeu vidéo et remportent toujours un franc succès auprès du public. Après un premier engouement pour ce type de jeu, le public a plébiscité leur adaptation au jeu en réseau. Ainsi Counter-Strike, une extension de Half-Life a tenu le haut du pavé pendant 5 années d'affilées, de 2000 jusqu'à aujourd'hui, faisant fi des progrès technologiques et algorithmiques. Aujourd'hui, ces derniers ont été tels qu'ils nous permettent de nous mouvoir dans des environnements spectaculaires. Il est alors intéressant de revenir sur les structures et algorithmes utilisés dans les premiers jeux. Une des techniques qui a le plus révolutionné les jeux 3D au début des années 90 fut celle premièrement utilisée par le créateur de Doom, John Carmack[1] : les arbres BSP (Binary Space Partition). En permettant une partition de l'espace de part et d'autre des murs du jeu, ces arbres sont très utiles et permettent entre autres la gestion de collisions et la détermination des surfaces visibles. Nous nous intéresserons ici uniquement aux arbres BSP 2D (leur équivalent 3D fonctionne sur le même principe à partir des normales des polygones 3D de l'espace de jeu, mais sont plus compliqués à mettre en œuvre et à visualiser). Notre projet consiste en deux programmes distincts XBsp et Xquake, le premier permettant la saisie à la souris d'un ensemble de murs 2D et la visualisation au fur et à mesure de sa création de l'arbre BSP correspondant. Le second permet d'ouvrir des mondes créés avec XBsp et d'évoluer dedans à la manière d'un doom-like. En plus du projet proprement dit, nous proposons quelques extensions : gestion des collisions, introduction d'autres personnages dans le monde sous forme de sprites, et le début d'une extension multijoueur pour jouer en réseau LAN ou WAN.

Table des matières

[1] qui avait commencé avec le célèbre Wolfenstein 3D, et dont l'entreprise ID Software éditera Quake par la suite

1 Les arbres BSP

1.1 La création

1.1.1 L'insertion d'un nouveau mur

Au cours de l'édition des murs dans XBsp, un arbre BSP est construit au fur et à mesure en introduisant le nouveau mur dans l'arbre déjà construit. Comme l'insertion d'un mur dans l'arbre est un procédé récursif, nous ne verrons que la première étape de celle-ci. On considère le mur qui est la racine de l'arbre courant, ce mur est muni d'une normale[2], et on regarde de quel côté de ce mur notre nouveau mur doit etre placé. Trois cas se présentent alors : le nouveau mur peut etre de part ou d'autre de la droite directrice du mur racine, ou bien il peut etre coupé en deux parties par celle-ci. Dans les deux premiers cas, on insère alors le mur récursivement dans le sous-arbre correspondant au côté où il se trouve. Pour déterminer le côté adéquat, on regarde le signe du produit scalaire entre le vecteur joignant les deux origines des murs et la normale au mur racine. Si celui-ci est négatif, alors le mur est

[2]qui par convention dans tout le projet, pointe vers l'intérieur du mur

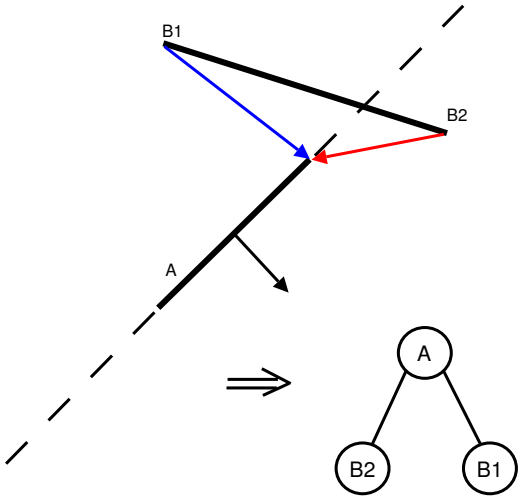

FIG. 2 – Exemple d'insertion du mur B par rapport à A

devant la normale, donc derrière le mur, et vice versa. Dans le deuxième cas, on découpe ce mur en deux et on insère chacun des nouveaux murs obtenus dans le sous-arbre correspondant[3].

1.1.2 Des murs aux segments

C'est ce découpage qui nous conduit à la distinction, dans l'implémentation, entre les murs (objets de la classe Mur) et les segments (objets de la classe XSegment). L'arbre BSP est en fait un arbre de segments tandis que les murs sont les objets rentrés effectivement par l'utilisateur et possèdent un indice qui leur est propre. Ainsi, chaque mur contient ses extrémités A et B, tandis que les segments contiennent l'indice du mur dont ils sont issus et deux flottants t_{start} et t_{end} compris entre 0 et 1 tels que les extrémités du segment soient $A + t_{start}\vec{AB}$ et $A + t_{end}\vec{AB}$. Ainsi, les informations comme les droites directrices ou les normales ne sont pas stockées de manière redondante dans chaque segment pour les segments qui proviennent d'un même mur.

Les noeuds des arbres BSP sont donc des segments, et l'insertion d'un mur est en fait l'insertion d'un segment où t_{start} et t_{end} valent resp. 0 et 1. Le découpage éventuel d'un segment par un autre se fait en cherchant pour quelle valeur de t les droites directrices se coupent, et si t est dans l'intervalle

[3] dans le programme, les murs sont numérotés par des entiers, et lorsqu'ils sont découpés, on ajoute à leur nom respectivement b et f pour les parties derrière et devant

$[t_{start}, t_{end}]$, en créant les segments $[t_{start}, t]$ et $[t, t_{end}]$ et en les insérant de part et d'autre du segment racine.

1.1.3 Problèmes d'implémentation

La théorie s'accompagne de quelques problèmes d'implémentation dus au caractère discret des données double et float en particulier. Ainsi, si le début d'un segment est trop proche du segment racine par rapport auquel on veut le placer, le produit scalaire destiné à déterminer de quel côté il se situe peut devenir nul (alors qu'il ne l'est pas en réalité) et ainsi ce segment se retrouve placé d'un côté arbitraire de l'arbre (et donc du mauvais côté une fois sur deux ! !). Pour pallier ce problème, on utilise un seuil[4] en-dessous duquel on utilise l'autre extrémité du segment pour effectuer le produit scalaire. Si l'autre extrémité également est trop proche, alors une exception ArbreException est levée et signale que le segment est quasimment sur l'autre, donc on l'enlève purement et simplement de l'arbre. Cela nous évite des problèmes et ne crée aucune différence d'affichage.

1.2 L'affichage

La partie gauche du programme (Canvas dérivé en DessinArbre) permet de visualiser l'arbre en cours de construction. On peut se déplacer quand l'arbre devient trop grand en faissant des glisser-déplacer avec la souris. L'algorithme utilisé pour tracer l'arbre est l'algorithme de Reingold-Tilford, qui consiste à calculer récursivement les encombrements horizontaux des sous-arbres gauches et droits, tracer ces sous-arbres de manière à laisser deux unités entre eux (pour éviter l'overlap), et placer la racine au milieu. Quand il n'y qu'un fils, on place la racine à une unité du sous-arbre, pour garder la symétrie.

Pour ce faire, chaque noeud de l'arbre contient, en plus d'un XSegment, les valeurs lWidth et rWidth qui représentent la largeur des sous-arbres gauche et droit de ce noeud. Ces valeurs sont calculées récursivement et gardées dans ces variables pour ne pas etre recalculées ensuite. La fonction récursive Bsp.drawTree() accède à ces valeurs ce qui lui permet de tracer les sous-arbres droit et gauche correctement, puis dessine la racine, et renvoie l'abscisse à laquelle celle-ci a été dessinée.

Enfin, le programme permet également de visualiser la position de l'observateur (petit rond noir dans le Canvas de gauche) dans l'arbre BSP. On peut le déplacer à la souris et voir alors dans quelle feuille de l'arbre il vient se placer.

[4] ce seuil est contenu dans XSegment.THRESHOLD

1.3 Les optimisations

Le programme XBsp permet par l'intermédiaire du menu Edition d'optimier l'arbre réalisé au fur et à mesure de la saisie des murs. Deux optimisations sont proposées : une de taille, et l'autre de profondeur. Ces optimisations sont effectuées par les fonctions de la classe BspCompiler qui reprennent la liste de murs et cherchent à chaque étape à sélectionner le *meilleur* mur possible. Ce concept de meilleur dépend de l'optimisation recherchée : si on s'intéresse à la taille, on va choisir le segment qui découpe le moins les autres, si en revanche on s'intéresse à la profondeur, on va choisir le segment qui répartit au mieux les autres murs de part et d'autre. Ce problème d'optimisation d'arbre nécessiterait en théorie de les tester tous ($n!$ où n est le nombre de murs ! !), mais cela est beaucoup trop lourd, on utilise donc cette méthode optimiste de meilleur choix à chaque étape, en espérant obtenir ainsi le meilleur arbre (ce n'est pas le cas en général, mais c'est mieux que rien).

2 Les problématiques d'affichage

Nous avons choisi d'utiliser la librairie AWT de java pour effectuer toutes les opérations d'affichage. Même si les performances sont beaucoup moins bonnes que si nous avions utilisé java 3D, qui implémente l'OPEN GL dans java, il aurait été frustrant de faire afficher un monde en 3D en reprogrammant des fonctions qui existent déjà dans cette API.

2.1 Utilisation des arbres BSP

2.1.1 Algorithme du peintre

L'algorithme du peintre est en soi extrêmement simple. Il consiste simplement à afficher les murs à l'écran dans le bon ordre afin que les arbres situés au loin soient dessinés en premier, et que les arbres situés à l'avant-plan soient dessinés en dernier. Le principe que nous utilisons est le suivant : dans la convention choisie, le fils gauche d'un noeud contient les murs qui se trouvent devant celui-ci. Inversement, le fils droit contient les murs situés derrière ce noeud. L'algorithme utilisé consiste alors à effectuer un parcours infixe de l'arbre BSP :

- si l'arbres est nul, on s'arrête.
- On teste la position du joueur par rapport à l'arbre racine. (il suffit de tester le signe d'un produit vectoriel)

– Si le joueur se trouve devant l'arbre racine, on trace récursivement les murs du fils gauche, puis le mur racine, puis les murs du fils droit. On fait l'inverse si le joueur se trouve derrière l'arbre racine.

Nous remarquons que le nombre d'appel à cette fonction est égal au nombre n de noeuds de l'arbre.

2.1.2 Test de collision

Lorsque l'on a réussi à implémenter un programme qui permet de se mouvoir dans le labyrinthe, une deuxième problématique intervient : comment faire pour que le personnage ne puisse pas traverser les murs ? Pour cela, nous utilisons encore l'arbre BSP et nous devons effectuer un test à chaque déplacement du personnage. Au vu du schéma suivant, il est simple

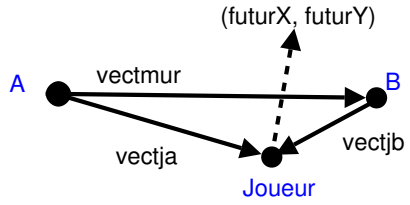

FIG. 3 – En bleu : points, en noir : vecteurs

de voir qu'un changement de signe du produit vectoriel de vectmur et de (vectja+futur) indique que le joueur a traversé la direction du mur. Ensuite, un test sur la position initiale du joueur indique si le joueur se trouvait entre les deux extrémités du mur (et donc qu'il cherchait à traverser le mur) ou non.

Ensuite, nous effectuons une descente de l'arbre BSP en partant du principe que si nous sommes derrière un mur N, alors nous ne pouvons pas traverser ceux qui se trouvent derrière N sans traverser N. Ainsi, seul le test de N suffit. Il n'est donc pas nécessaire de tester tous les murs, mais seulement d'effectuer une descente en profondeur. l'algorithme de gestion de la collision est le suivant :

- si l'arbre est nul on s'arrête
- on teste la position du joueur avec celle du mur
- si on se trouve derrière,
  - on teste si on traverse le mur racine.
  - si on traverse le mur, on bloque le mouvement.

– sinon on teste récursivement les collisions sur les murs qui se trouvent aussi derrière ce mur, c'est à dire sur le fils droit.
– On fait l'inverse si on se trouve devant.

La structure de l'arbre importe ici plus que le simple nombre de noeud. En effet, le nombre maximum d'appel à cette fonction pour un test de collision est égal à la profondeur maximale de l'arbre. Le nombre d'appel à cette fonction varie donc entre $Log(n)$ pour un arbre bien équilibré et n dans le pire des cas, quand l'arbre est en peigne.

2.2 La projection 3D

2.2.1 Affichage des murs

Nous ne reviendrons pas sur le code java dans cette partie, mais simplement sur les principes mathématiques que nous avons utilisés. Tout d'abord, chaque mur sera dessiné à l'écran comme un trapèze (un polygone en java). Il importe donc de connaître les positions à l'écran des deux extrémités du mur. Nous calculons la hauteur à l'écran du mur, représentée par le trait rouge sur le schéma suivant, par de simples formules de trigonométrie. Pour

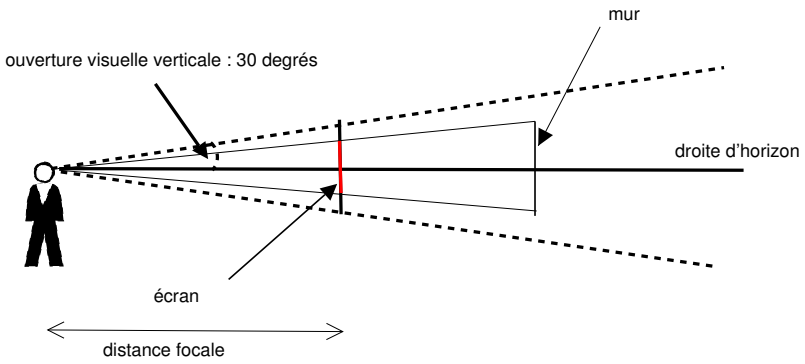

FIG. 4 – Vue en coupe

la position horizontale, le problème est un peu plus compliqué : si le mur comporte une partie visible, il faut effectuer deux projections, l'une sur le cône visible, puis une sur l'écran.

2.2.2 Simulation d'un éclairage

Nous utilisons les fonctions de l'API AWT pour générer un gradient qui va du rouge vif lorsqu'un objet est loin de l'écran au noir lorsqu'il est loin. Le résultat est assez convaincant.

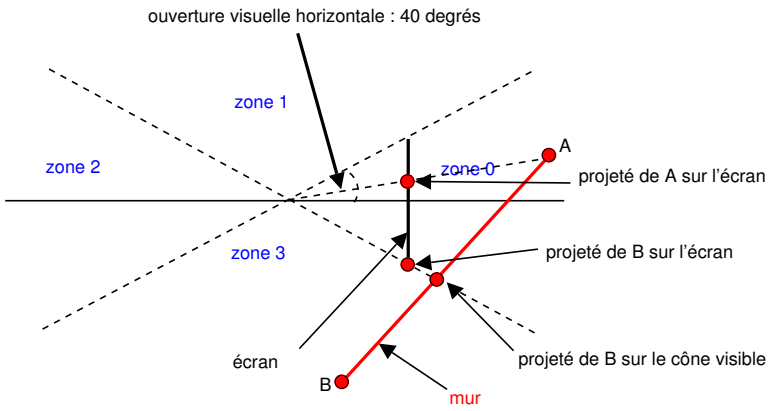

FIG. 5 – Vue de dessus

2.3 La structure de Xquake

2.3.1 La gestion des événements

Comme indiqué dans les consignes, nous ne nous attarderons pas sur ces techniques d'implémentation. Néanmoins, notons que notre programme permet de se mouvoir en temps réel dans le labyrinthe en gérant toutes les touches et la souris en même temps. Nous avons donc :

- Quatre booléens indiquant l'état (enfoncé ou relaché) des flèches de directions.
- Deux variables entières de position de la souris.
- Un KeyListener mettant à jour les booléens
- Un MouseMotionListener mettant à jour les deux entiers
- Un MouseListener gérant le clic de la souris
- Un thread gérant les événements clavier
- Un thread gérant les événements souris.
- Un FocusListener gérant la perte de focus de l'application
- Un robot (classe Robot) utilisé dans le thread de la souris, qui permet de recentrer la souris au centre de la fenêtre lorsque celle-ci s'approche trop des bords. Ceci permet une jouabilité aussi proche que possible de celle rencontrée dans les jeux habituels.

2.3.2 Les techniques d'affichage

Tout d'abord, nous avons un thread d'affichage qui fait appel continuellement à la fonction paint() du Canvas principal. Le temps entre deux appels est de 12 ms, ce qui permet au mieux d'avoir 85 frames par secondes. En réa-

lité, ce nombre n'est presque jamais atteint. Mettre un temps d'attente trop faible ne sert à rien car en réalité l'application ne peut pas afficher un nombre trop important d'image par secondes. Il en résulterait un nombre trop grand d'appels à paint() qui ne seraient pas traités, mais aussi une consommation inutile de temps processeur.

Ensuite, la deuxième technique que nous avons employée et qui mérite d'être soulignée concerne le "double buffering", qui augmente grandement les performances du jeu et la qualité visuelle. Elle consiste à travailler et dessiner sur un buffer "caché" qui ne s'affiche pas à l'écran et à le faire afficher dans le buffer principal lorsque l'image à afficher est prête. Typiquement, une image sera "prête" lorsque les murs auront tous été dessinés. Pour cela, nous avons utilisé la classe BufferStrategy.

3 Etude de performance

Les tests ont été effectués sur la machine suivante :
- athlon 2800+ (2088 Mhz)
- 512 Mo RAM
- carte vidéo ATI
- OS : Linux Mandrake cooker, kernel 2.6.6

3.1 Optimisation en taille de l'arbre

Voici un tableau récapitulatif des tests que nous avons effectués, en testant un parcours identique sur trois mondes de tailles différentes. La résolution de la fenêtre était de 400x300. Notre programme retourne le nombre moyen de frames par seconde qu'il a affiché lorsque nous le fermons.

nombre de sprites	0	2	4	5
petit optimisé	40.8	39.6	37.9	38.9
petit non optimisé	40.3	39.6	39	39.3
moyen optimisé	36.8	36	33.2	34
moyen non optimisé	35.7	34	32	32.84
grand optimisé	24.3	23.4	22.6	22.54
grand non optimisé	20.4	19	20.5	19.4

Ces chiffres sont à prendre avec précaution, car les sprites à l'écran avaient un parcours aléatoire, ce qui pouvait faire varier les résultats. Néanmoins, une tendance se dégage et il est à peu près clair que l'optimisation permet de gagner plus de deux frames par seconde dans le cas où le monde est grand, ce qui n'est pas négligeable.

3.2 Problématique du drawImage()

La principale limitation que nous avons rencontrée en utilisant la libraire AWT résidait dans la lenteur de la fonction drawImage(), qui nous sert à afficher la main en avant-plan et les sprites. En effet, dès que nous appelons la fonction drawImage() en lui demandant d'afficher une image occupant une place sur l'écran supérieure à une certaine taille critique, le nombre de frames par seconde se trouve divisé par 10. Voici le résultat de nos tests avec 5 sprites, dans le monde petit, en faisant varier la résolution.

Résolution	320x240	400x300	420x320	430x330	440x340
FPS	50	37	34	4	4

Résolution	450x350	460x360	500x400	800x600	
FPS	4	4	3.7	1.5	

Le résultat est étonnant. En fait ce ralentissement est dû à l'affichage de la main en avant-plan. Lorsque celle-ci dépasse la taille critique, le ralentissement se fait sentir. Le problème était aussi présent avec les sprites. Pour nous en sortir, nous avons découpé leurs images en quatre parties et nous les affichons donc en quatre fois. Le problème est alors résolu pour une résolution allant jusqu'au 400x300. Si l'on souhaite monter dans des résolutions supérieures, il faudrait faire de même avec la main d'avant-plan. Afin de confirmer ce fait, nous avons supprimmé l'affichage de la main et tous les sprites. Dans une résolution de 800x600, nous avions alors 10 FPS.

3.3 Optimisation en profondeur de l'arbre

En résumé, l'optimisation par taille se révèle un peu meilleure en moyenne. En fait, il faut vraiment choisir visuellement laquelle choisir : si au départ on dispose d'un arbre BSP "en peigne", c'est à dire très profond, alors il vaut mieux choisir l'option "optimiser en profondeur". Inversement, si l'arbre est très grand, si les segments se coupent beaucoup, ce qui arrive dans beaucoup de cas, il vaut mieux choisir optimisation en taille.

4 Extensions

4.1 Utilisation de sprites

Comme on peut le voir sur la capture d'écran de couverture, nous avons mis des sprites dans notre programme. Ceux-ci sont en réalité des threads et ont une direction aléatoire qui change dès qu'ils touchent un mur. La gestion des collisions n'a pas posé de problème, car il s'agissait des mêmes fonctions que pour le joueur (nous considérons que les sprites sont ponctuels).

Cependant, pour gérer leur affichage, nous avons rajouté des feuilles aux arbres BSP contenant la liste des sprites de la zone considérée. Au tout début, nous insérons les sprites dans l'arbre BSP général. Ensuite, leur fonction de collision a également le mérite de tester le changement de zone dans l'arbre BSP. Lorsqu'un sprite change de zone, une fonction est donc appelée qui le retire de la feuille de l'arbre BSP où il se trouvait, puis le réinsère dans l'arbre BSP.

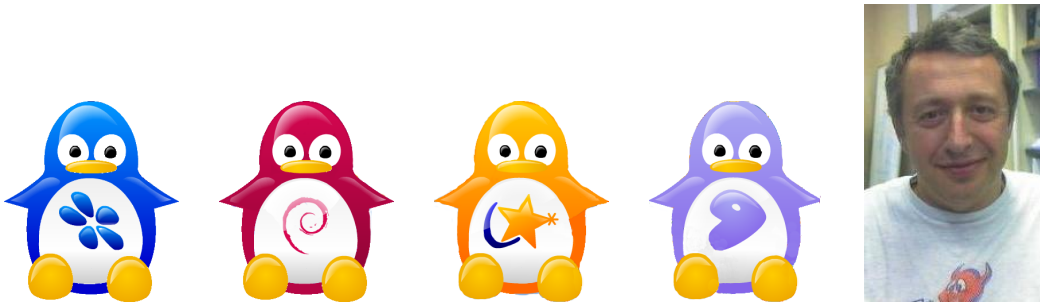

FIG. 6 – Les différents sprites du jeu

4.2 Le début d'une extension réseau

Dans l'optique de rendre notre jeu multijoueurs, nous avons commencé une interface de communication client/serveur pour Xquake. Nous n'avons pas mené cette extension à son terme car cela demandait beaucoup trop de travail au vu du temps qu'il nous restait et de l'ambition d'un Projet Informatique ; cependant, nous avons réalisé une architecture de base qui permettrait assez facilement d'obtenir un jeu multijoueurs.

Une classe XQuakeServer permet, une fois lancée, d'écouter le port TCP 1234 et de communiquer avec les clients XQuakeClient selon un protocole maison très simple. Une fois connectés au serveur, les clients peuvent etre soit observateur, soit rejoindre le jeu courant (dans la limite du nombre de joueurs maximum accepté par le serveur). Par un jeu de requetes au serveur et un thread d'écoute des réponses, les clients savent en temps réel quels sont les autres joueurs connectés, si ceux-ci sont dans la partie ou pas, etc. Nous avons réalisé une petite interface graphique XQuakeClientUI pour gérer un client (sinon on peut tester le server avec telnet mais c'est moins convivial).

Il resterait à créer le même genre d'interface pour créer et administrer le serveur de jeu, et à adapter le protocole pour que le serveur puisse envoyer la carte (au départ) aux clients, et pour que les clients puissent communiquer leurs données importantes (position du joueur, vie restante, tout ce qui peut

etre utile aux autres clients en somme) aux autres clients par l'intermédiaire d'un broadcast du serveur. On pourrait alors ajouter aux sprites SpritePingouin des sprites SpriteJoueur[5], les insérer de la même façon dans l'arbre BSP et en gérer l'affichage.

Conclusion

En conclusion, nous sommes arrivés à développer des applications permettant d'éditer des mondes en 2D et d'utiliser la technique des arbres BSP pour se déplacer dans ces univers en créant l'illusion de la troisième dimension. Nous avons obtenu un jeu fluide dans une résolution de 400x300 et la principale limitation que nous avons eue est due à la lenteur de certaines fonctions d'AWT, comme drawImage(). Néanmoins, le futur laisse présager de grandes améliorations. Ainsi, la gestion des images s'est considérablement améliorée de la version 1.3 à la version 1.4 de java. Sur le site de Sun est disponible une version 1.5 beta qui acceptait d'utiliser automatiquement l'openGL sur nos machines pour effectuer les fonctions de dessins, ce qui aurait été beaucoup plus rapide, car nous aurions eu une accélération matérielle. Cependant, celle-ci n'était pas encore au point comme nous l'avons constaté en essayant de l'utiliser. En attendant, nous avons tout de meme un programme jouable ou l'utilisateur se déplace dans un monde en 3D et peut attraper des personnages controlés par le programme, et qui pourrait etre étendu à un mode multijoueurs en réseau grace au début d'extension proposé, ce qui lui conférerait un intéret encore supérieur.

[5]dans cette optique, ces deux types d'entité dérivent de la même classe abstraite Sprite