Chapitre 5

Premières mesures de difficulté

Sommaire

Les premières expériences réalisées au cours de cette thèse ont consisté à analyser la difficulté de différents gameplays à l'aide d'un joueur synthétique, c'est à dire une intelligence artificielle chargée de remplacer le joueur. Cette première approche, empirique, a pour but de confirmer et d'étendre les recherches présentées dans la section **??**, en nous concentrant plus particulièrement sur une analyse de la difficulté. Ce chapitre présente ces expériences et discute de leurs apports.

5.1 Analyse de la difficulté et joueur synthétique

Durant ces premières expériences, nous avons tenté d'analyser la difficulté de différents jeux, au moyen d'un joueur synthétique. Il est en effet possible de construire un agent logiciel qui soit en mesure de percevoir l'univers du jeu et d'y effectuer des actions, avec l'objectif de maximiser un résultat. Les différents objets de l'univers sont des structures de données à priori accessibles à n'importe quel agent logiciel, et les actions disponibles sont toutes connues et définies par le game designer, étant constitutives des règles du jeu. De plus, de nombreux jeux utilisent déjà des agents logiciels chargés de simuler un adversaire, et l'interface entre l'univers du jeu et ces agents est donc déjà définie.

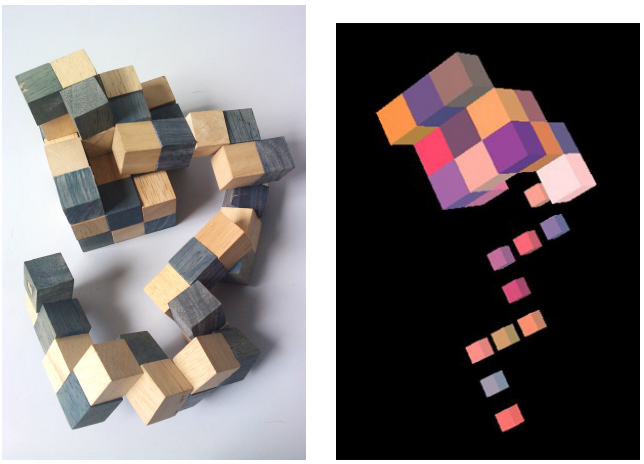
L'utilisation d'un joueur synthétique présente de nombreux attraits, dont celui de supprimer tout facteur humain, ce qui permet organiser des expériences sans avoir à recruter de vrais joueurs. Les itérations entre différents tests sont alors beaucoup plus rapides et moins risquées, ce qui offre la souplesse nécessaire à des expériences réalisées en début de thèse. Cette approche permet également d'évaluer l'intérêt de la conception de joueurs synthétiques pour un studio de jeu. Développer un joueur synthétique demande un investissement, qui peut être rentabilisé par la suite en automatisant une partie des tests effectués sur le gameplay. Ces premières expériences visent à montrer, du point de vue de la difficulté, le type d'informations qu'est capable de fournir un joueur synthétique, ainsi que ses limitations.

5.2 Premières expériences

Nous présentons dans ces sections trois expériences au cours desquelles nous avons analysé le gameplay d'un jeu vidéo au moyen d'un joueur synthétique. Chaque expérience présente un type d'analyse et un gameplay particuliers, puis décrit les résultats obtenus.

5.2.1 Le cube-serpent

Cette première expérience étudie la difficulté d'un jeu aux règles simple mais dont l'objectif est particulièrement difficile à atteindre. Ce jeu n'est pas au départ un jeu vidéo, mais sa simplicité le rend particulièrement facile à simuler dans un univers virtuel. Ce jeu en bois s'appelle le *cube-serpent*, présenté avec son équivalent virtuel figure 5.1.



Cube serpent en bois Cube serpent virtuel

FIGURE 5.1 – Cube-serpent

Le cube serpent se compose de petits cubes de bois attachés les uns aux autres. Chaque cube a au moins une face plaquée à celle d'un autre cube. Ces deux faces peuvent pivoter l'une contre l'autre, en effectuant ainsi une rotation atour de leur normales. L'objectif consiste à replier le serpent formé par cette suite de cubes en un hyper cube d'une taille donnée.

Il existe deux versions de ce jeu. La plus populaire propose un serpent à replier dans un hypercube de trois petits cubes de côté. Dans une version plus complexe, le serpent doit être replié dans un hypercube taille quatre. En effet, une bonne manière de complexifier un gameplay aux règles aussi basiques que celui du cube-serpent consiste à fournir un serpent plus long, et donc à augmenter le nombre de possibilités lorsqu'on tente de le replier.

Nous avons choisit de réaliser cette expérience après avoir constaté à quel point la difficulté différait entre les deux jeux. Si la version populaire du cube-serpent, de taille trois, permet d'atteindre une solution après quelques heures de recherche, celle du grand cube apparait totalement inatteignable, et ce même en possession de la petite fiche de solution. Nous avons cherché à vérifier si cet écart de difficulté apparaissait aussi important pour un joueur humain que pour un joueur synthétique.

Nous avons donc construit l'équivalent du cube-serpent sous forme virtuelle. Il s'agit d'une application qui exploite la librairie graphique Open-GL pour le rendu du serpent, et encode l'angle de rotation de chaque cube dans un tableau d'octets. Tous les cubes ne sont pas à manipuler par l'algorithme de résolution, car le serpent possède des sections *rigides*, décrites dans la figure 5.2. Dans ces sections, la rotation de l'ensemble des cubes produit le même effet sur le serpent, et de plus, il est impossible de replier ces régions.

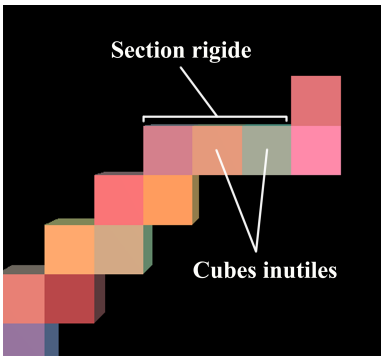


FIGURE 5.2 – Section rigide d'un cube-serpent.

> La section rigide a deux propriétés : elle ne peut pas se replier sur elle même, et tous les cubes qui la composent on un impact similaire sur le reste du serpent quand on les pivote.

Nous avons ensuite écrit un algorithme de recherche brutal, qui progresse section rigide par section rigide, en cherchant à chaque fois un angle de rotation qui maintienne la prochaine

section rigide à l'intérieur de l'hypercube. S'il n'existe aucune position valide pour la section rigide actuelle, c'est qu'un des choix précédents est mauvais, et l'algorithme revient une section en arrière. Lorsque la dernière section rigide est placée dans l'hypercube, l'algorithme a trouvé une solution. Nous avons conçu cet algorithme en cherchant à reproduire notre propre comportement, lorsque nous essayons de trouver une solution au cube-serpent en bois.

Nous avons exécuté l'algorithme plusieurs fois pour chaque type de cube-serpent, sur la même machine. Nous avons ensuite comparé le temps mis par l'algorithme pour parvenir à une solution dans chaque configuration. Ces résultats sont présentés figure 5.3.

Condition	Moyenne	Ecart type	Nombre d'essais
Serpent 3*3	125 ms	48 ms	20
Serpent 4*4	448 heures	244 heures	6

FIGURE 5.3 – Temps d'exécution pour découvrir la solution de cube-serpents.

On remarque que la différence de temps de calcul, et donc l'effort fournit par l'algorithme, passe de quelques millièmes de secondes à plusieurs jours selon qu'il s'agit du serpent 3*3 ou du serpent 4*4. L'algorithme synthétique semble donc confirmer d'une manière écrasante la différence de difficulté que nous avions ressentit entre les deux conditions. Agrandir l'hypercube d'un petit cube de côté a considérablement augmenté la difficulté du casse tête. Bien sûr, il faut noter que l'algorithme de résolution est aussi idiot que son concepteur, et qu'il existe peut être des heuristiques, au delà des simples sections rigides, qui permettent de percer le mystère du cube 4*4. L'algorithme a permis de valuer la différence de difficulté pour une méthode de résolutions donnée, qui n'est peut être tout simplement pas appropriée. Seul un test sur des joueurs humains aurait pu permettre de découvrir si un joueur moyen peut trouver la solution dans un temps raisonnable.

5.2.2 Skywar

Notre seconde expérience a consisté à étudier le gameplay d'un jeu vidéo pour tenter d'y déceler une erreur d'équilibrage de la difficulté. Ce genre d'expérience est en général particulièrement complexe a réaliser sur un gameplay conçu par un studio de jeu vidéo : il faut être en mesure de simuler le gameplay ou d'accéder au moteur du jeu, ce qui demande des informations ou des outils logiciels que les studios, en perpétuelle compétition, se gardent bien d'offrir. Il existe cependant certaines exceptions, comme par exemple le **First Person Shooter**[1] Unreal Tournament, dont le moteur est commercialisé et dont l'interface logicielle est ainsi accessible. Mais dans leur grande majorité, les jeux mis sur le marché sont particulièrement protégés.

1. First Person Shooter, jeu de tir en vue subjective.

FIGURE 5.4 – Skywar (MotionTwin)

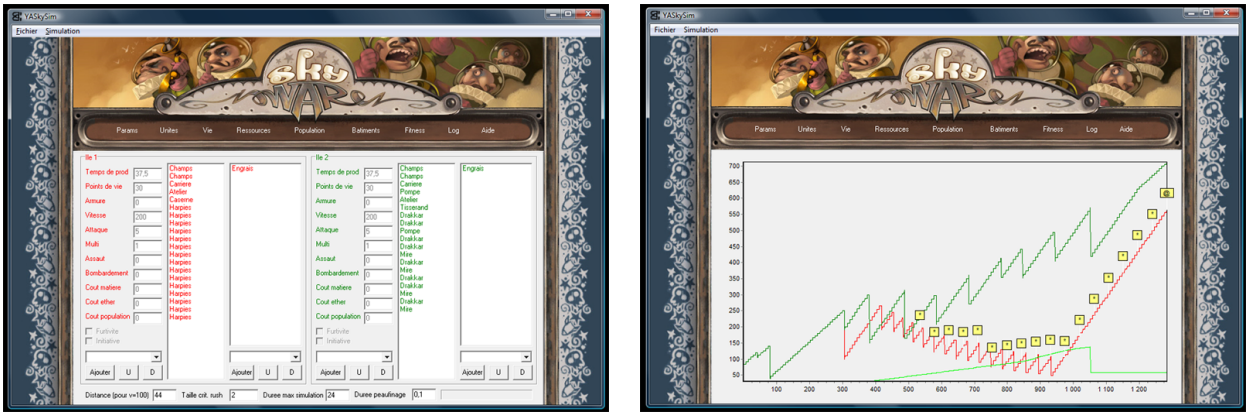
> Chaque joueur possède un certain nombre d'ilôts, sur lesquels il construit divers bâtiments. Ces bâtiments produisent des ressources et des unités de combats qu'il utilise pour attaquer les îles adverses. Un partie dure plusieurs jours, chaque action pouvant prendre plusieurs heures pour s'exécuter.

Certains jeux échappent cependant à cette règle, ce qui est le cas du jeu que nous avons souhaité étudier : Skywar (MotionTwin - fig. 5.4). Tout d'abord, Skywar est un jeu de stratégie temps réel, ce qui oblige ses concepteurs à fournir de nombreux détails concernant son gameplay. Les joueur doivent en effet connaitre l'ensemble des caractéristiques de chaque unité pour pouvoir élaborer leur stratégie, le jeu serait sinon sûrement bien trop complexe : ce serait comme jouer aux échecs tout en essayant de découvrir les règles de manipulation des pièces. Le premier avantage du jeu Skywar est donc tout d'abord de disposer d'une documentation précise.

Ensuite, Skywar a été conçu pour être joué sur le web, et réalisé en flash. La logique du jeu est calculée côté serveur, et transmise au client flash pour un simple rendu. Cette logique est donc particulièrement simple, elle ne demande qu'un calcul léger au serveur. Elle est de plus précisément décrite dans la documentation et les forums du jeu. Certains jeux plus complexes, comme Starcraft (Blizzard) par exemple, fournissent également une description des unités, mais la logique du jeu est bien plus sophistiqué et n'est précisément décrit nulle part.

Toutes ces caractéristiques rendent Skywar particulièrement intéressant pour cette expérience. Les RTS sont des jeux au gameplay émergeant, particulièrement sensibles aux problèmes d'équilibrage. La simplicité du gameplay de Skywar ne diminue en rien sa qualité de jeu de stratégie, mais nous permet d'en construire facilement un simulateur, et d'en étudier l'équilibre.

Skywar est un jeu de stratégie temps réel, qui se joue en ligne, contre trois autres joueurs. Le terrain de jeu est un chapelet d'îles volantes. Chaque joueur choisit une île de départ, et peut ensuite coloniser d'autres îles. L'objectif de chaque joueur est de détruire tous les bâtiments et unités des autres joueurs, ou de les forcer à abandonner. Comme dans la plupart des RTS, le principe de Skywar consiste à construire des bâtiments, qui offrent au joueur différentes capacités : défendre l'île sur laquelle ils sont construits, produire des unités de combat ou récolter les ressources de l'île. En parallèle, les joueurs choisissent une suite d'améliorations à développer. Les améliorations sont activées les unes après les autres, en suivant un ordre défini par le joueur, et lui apportent différents bonus. Skywar propose un gameplay lent, c'est à dire que les actions du joueur demandent plusieurs dizaines de minutes avant d'être totalement exécutées, et donc les parties durent plusieurs jours.



Ecran principal Construction des unités dans le temps

FIGURE 5.5 – Simulateur de SkyWar

Notre objectif, pour cette expérience, n'est pas de mesurer la difficulté de skywar dans un contexte précis, mais d'en vérifier l'équilibre. Skywar est un jeu multijoueurs, et donc la difficulté d'une partie repose sur l'habileté des opposants, si tant est que le gameplay permette toujours aux opposants de s'affronter. S'il existe une stratégie simple qui garantit la victoire au joueur qui l'exécute, alors la difficulté deviendra nulle pour ce joueur, dès la stratégie découverte. Notre objectif est donc d'explorer le gameplay de Skywar, à la recherche d'une telle stratégie.

Nous avons programmé un simulateur capable de reproduire la logique de Skywar. Ce logiciel simule une partie en quelques millisecondes, ce qui permet de tester le gameplay beaucoup plus rapidement qu'en jouant sur le serveur. Bien sûr, nous ne sommes pas en mesure de reproduire l'aspect stratégique de ce jeu, car nous ne pouvons pas simuler le comportement des autres joueurs. Néanmoins, nous sommes capables de simuler un scénario particulier, qui nous permet de comparer les capacités des bâtiments et unités, étant donné

un comportement basique des joueurs.

En effet, une partie de l'équilibrage d'un RTS peut être exprimé de la façon suivante :

> « *Etant donné deux joueurs s'affrontant de manière très basique, en envoyant par exemple toutes leurs unités attaquer l'autre joueur lorsque le nombre total d'unités en leur possession atteint un certain seuil, existe t'il un type de bâtiment et un type d'unité qui permette d'obtenir le plus souvent la victoire ?* »

Les joueurs appellent **build** [2] la liste de bâtiments et d'unités construites au cours d'une partie. Les premiers instants de jeu sont toujours consacrés à la mise en place du build car les joueurs ne disposent d'aucune unité et doivent tout d'abord construire leur base puis un stock d'unités. Dans un RTS bien équilibré, il ne doit pas exister de build qui garantisse la victoire du joueur, il doit toujours exister une faiblesse qu'un autre build pourra exploiter, à la manière d'un jeu pierre-feuille-ciseaux.

Nous avons donc codé un scénario ne comportant que deux îles et deux joueurs. Les paramètres modulant le comportement de ces deux joueurs sont les suivants :

- La liste des bâtiments et unités construits par le joueur
- La liste des améliorations activées par le joueur
- Le nombre d'unités total nécessaire avant de les envoyer attaquer l'ennemi

Le simulateur permet ainsi de comparer deux builds très rapidement. De manière à accélérer encore l'étude du gameplay, et pour permettre de faire évoluer un build automatiquement, nous avons programmé un algorithme d'optimisation de type génétique. Il est en effet possible d'encoder un build sous la forme d'un gène, et d'opérer une mutation, c'est à dire une modification aléatoire d'un élément de ce build. Notre algorithme est très basique et n'utilise pas d'opérateur de croisement ni ne renouvelle la population. Il initialise simplement une population avec la stratégie fournie, puis fait muter l'ensemble de la population. Pour chaque individu, si la mutation fournit un build plus performant, celui-ci est sauvegardé, sinon, la mutation est annulée et une autre mutation est réalisée. La performance d'un build correspond au temps nécessaire pour vaincre l'ennemi lorsqu'il est utilisé.

Une alternance de recherche manuelle et d'optimisation automatique nous a permis d'obtenir un build particulièrement performant, auquel nous n'avons pu trouver aucun équivalent. Ce build est plutôt inattendu car il exploite à outrance une unité en apparence plutôt faible, la *harpie*, qui dispose de peu de points de vie est n'est réellement efficace que contre d'autres unités volantes. Néanmoins, cette unité peut être produite très rapidement, et pour un faible coût, ce qui lui permet d'établir rapidement sa suprématie.

2. Suite de bâtiments, d'améliorations et d'unités construites par un joueur dans un jeu de stratégie temps réel. Un build retrace les premiers instants de jeu, fournissant une méthode pas à pas pour développer la base d'un joueur.

Nous avons cherché à confirmer cette analyse en utilisant ce build contre de vrais joueurs, sur le serveur de jeu. Nous avons obtenu les résultats suivants, récapitulés figure 5.6.

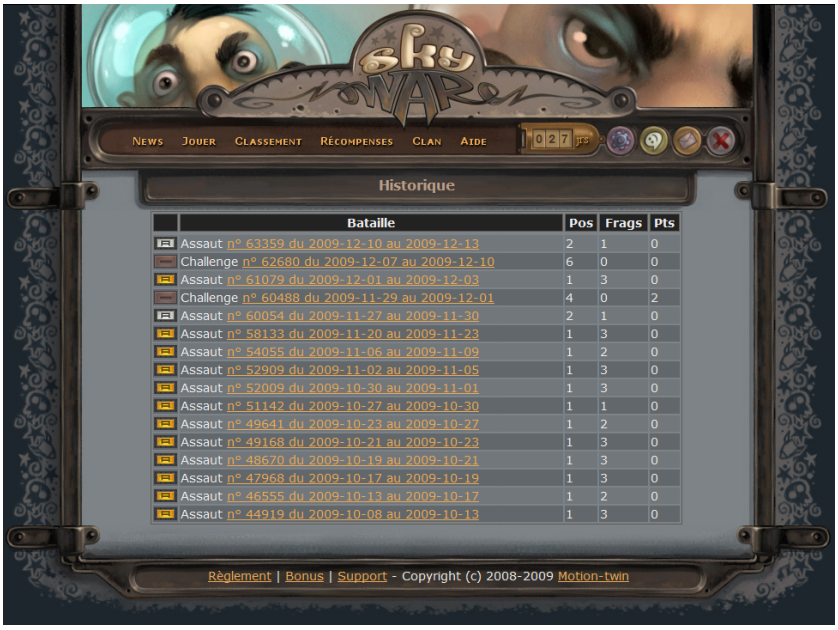


FIGURE 5.6 – Résultats obtenus grâce au *build harpies*.

Seules les parties de type assaut correspondent à des tests du *build harpies*. Les parties challenges ont été jouées contre un autre type de joueurs, avec des unités différentes que nous n'avons pas étudiées. La couleur or correspond à la première place, argent à la seconde.

Sur les 14 parties effectuées en mode assaut, c'est à dire contre trois nouveaux joueurs humains pour chaque partie, nous en avons gagné 12, et terminé deux fois en seconde position, ce qui tend à confirmer l'efficacité du *build harpies*. Bien sûr, d'autres facteurs entrent en considération : les joueurs peuvent communiquer et décider de s'allier contre un seul joueur, ou la configuration de départ des îles, aléatoire, peut favoriser un joueur plus que les autres. Mais le gameplay de Skywar pourrait être équilibré en rallongeant par exemple légèrement la durée de fabrication des harpies. Notre application a bel et bien permis de mettre à jour une erreur d'équilibrage du gameplay.

Cette expérience démontre qu'avec un faible investissement, il est possible de découvrir certaines faiblesses d'équilibrage du gameplay développé par un studio reconnu. Ce type de déséquilibre peut nuire profondément au gameplay du jeu, en éliminant toute forme de challenge pour le joueur qui a découvert la super stratégie.

5.2.3 Pacman

Nous présentons ici une troisième expérience, au cours de laquelle nous avons étudié le gameplay du célèbre jeu Pacman (Namco) en développant un joueur synthétique chargé de diriger Pacman. Dans cette expérience, nous avons analysé une version simplifiée du gameplay d'origine, de manière a pouvoir interpréter plus facilement les résultats. Pacman n'est pourchassé que par un seul fantôme : *Blinky*, dont l'algorithme consiste à trouver le plus court chemin pour intercepter Pacman. Dans la version originale, quatre fantômes aux comportements différents peuvent intercepter Pacman, qui peut utiliser des jetons de puissance pour renverser la situation et être capable de manger les fantômes. Le gameplay original alterne également des phases de poursuites avec des phases de *dispersion* au cours desquelles les fantômes regagnent chacun un point précis du niveau et laissent Pacman évoluer assez librement. Dans notre version simplifiée, l'objectif du joueur reste de manger le plus possible de pastilles, dispersées dans tout le niveau.

Pacman et Blinky utilisent tous les deux une recherche de chemin basée sur l'algorithme A^*. Le graphe des chemins les plus courts est construit et calculé au départ, de manière à limiter les phases de calcul à l'exécution. Le joueur synthétique a une vision partielle de l'univers du jeu, construite à partir de cinq paramètres. Le premier paramètre donne la direction de la pastille la plus proche, et puisque Pacman ne peut aller que dans quatre directions, ce paramètre n'a que quatre valeurs possibles. Les quatre autres paramètres décrivent les quatre directions que peut emprunter Pacman. Pour chaque direction, Pacman sait s'il est face à un mur (0), face à un fantôme se trouvant à une (1) ou deux (2) cases de distance, face à un chemin libre de fantômes de moins de 18 cases de long (3), ou face à un chemin libre de fantômes de plus de 18 cases de long (4). Nous pensons que ces informations fournissent une bonne abstraction de la situation telle que perçue par le joueur.

L'**Intelligence Artificielle**[3] du joueur synthétique a été implantée sous la forme d'un Processus de Decision Markovien[4], construit grâce à l'algorithme d'apprentissage $Q(\lambda)$, avec traces d'éligibilité [Sutton 98]. Les paramètres de $Q(\lambda)$ employés sont les suivants : $\alpha = 0.1$, $\gamma = 0.95$, $\lambda = 0.90$. Nous avons équilibré les phase d'exploration et d'exploitation à l'aide d'une selection d'action ϵ-greedy, avec $\epsilon = 0.05$.

Nous avons évalué la difficulté de ce gameplay vis à vis d'un paramètre donné : la vitesse du joueur. Nous considérons que le niveau de difficulté peut être évalué à partir du nombre de pastilles mangées par Pacman. Nous avons choisit d'étudier l'impact de la vitesse de Pacman sur la difficulté du jeu car la vitesse relative de Pacman et des fantômes est un paramètre utilisé dans la version originale du jeu pour manipuler sa difficulté [Pittman 09]. Tous les 14 cycles, Blinky change sa position, en se déplaçant d'une case dans la direction choisie.

Nous avons évalué la performance du joueur synthétique, et donc la difficulté du gameplay,

3. Intelligence Artificielle.
4. MDP, Markov Decision Process

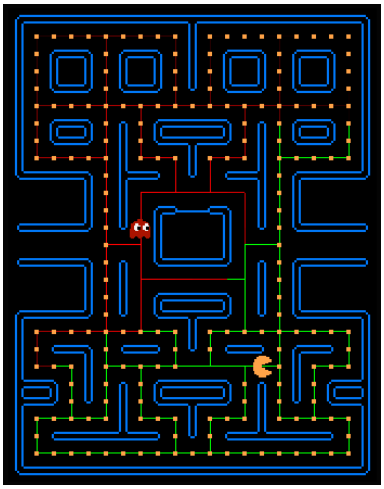


FIGURE 5.7 – Pacman chassé par Blinky.

sous différentes configurations. Nous avons réalisé six expériences, avec une vitesse de Pacman variant de 0 (Pacman et Blinky se déplacent tous les 14 cycles) à 7 (Blinky se déplace tous les 14 cycles mais Pacman se déplace tous les 7 cycles). Nous avons laissé le joueur synthétique développer sa politique pendant 50 000 jeux, Pacman ayant trois vies par jeu.

La figure **??** présente une vue synthétique des résultats obtenus. Nous pouvons déduire de ces résultats que lorsque nous modifions la vitesse de Pacman, la difficulté du jeu n'évolue pas de manière linéaire. Il y'a beaucoup moins de différence de score entre les vitesses 0 et 5 qu'entre les vitesses 5 et 7. Cette analyse pourrait être utile pour un game designer chargé d'équilibrer la difficulté d'un tel jeu. Ces résultats lui révèlent que lorsque la vitesse de Pacman approche de deux fois celle de Blinky, le jeu devient facile beaucoup plus rapidement. Entre 0 et 5, la difficulté évolue de manière linéaire.

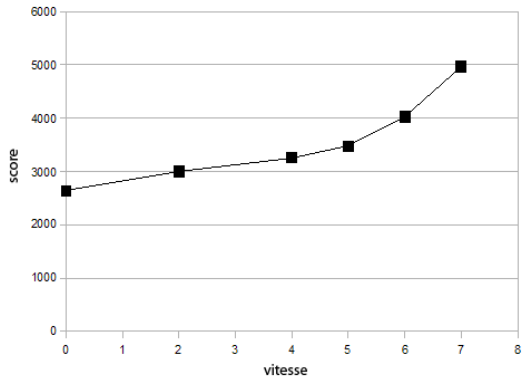


FIGURE 5.8 – Score de Pacman à différentes vitesses

Le score représenté et le score moyen obtenu par le joueur synthétique sur les 5000 dernières parties jouées à une vitesse donnée. A chaque pastille récupérée, le joueur marque 10 points.

5.3 Limitations d'un joueur synthétique

Les trois expériences précédentes montrent qu'il peut être utile de remplacer le joueur par une intelligence artificielle pour analyser un gameplay. Ces trois expériences ont confirmé ou mis en évidence des propriétés de l'équilibrage de gameplays, qui peuvent servir de base à la réflexion d'un game designer. Le développement d'un joueur synthétique demande certes un investissement, mais celui-ci peut être minimisé si le jeu dispose déjà d'un système d'agents intelligents, et donc de l'interface logicielle nécessaire au développement d'un joueur synthétique.

Néanmoins, dans l'optique d'analyser la difficulté d'un gameplay, le joueur synthétique montre rapidement ses limites. En effet, la difficulté du gameplay doit être évaluée pour un joueur humain, qu'une intelligence artificielle n'est pas en mesure de simuler. Les trois phases qui décrivent l'activité du joueur, c'est à dire la *perception*, la *décision* et l'*action*, permettent de mettre en évidence les failles du joueur synthétique.

La phase de perception est par exemple particulièrement complexe a simuler. Un joueur synthétique perçoit l'univers du jeu à partir d'une interface logicielle qui lui permet d'accéder aux structures de données qui décrivent cet univers. A priori, un agent peut donc obtenir une connaissance totale de l'état du jeu, à laquelle un joueur n'a pas forcément accès. Mais même si l'on limite l'accès de l'agent aux données qu'on suppose perceptibles pour le joueur humain, les deux perceptions restent fondamentalement différentes. Le joueur humain perçoit l'univers à partir d'une analyse complexe des signaux visuels et sonores, que nous somme incapables de reproduire algorithmiquement. Les effets graphiques, le choix des textures, ont

par exemple un impact direct sur la perception de l'univers par le joueur humain, auquel est totalement insensible le joueur synthétique.

La phase de décision d'un humain est elle aussi très difficile à reproduire. Il existe plusieurs théories qui cherchent à décrire le raisonnement humain [Noveck 07]. Les théories qui s'appuient sur une logique de raisonnement innée, comme la théorie de la logique mentale [Braine 90], semblent les plus évidentes à simuler, à partir de l'implantation sous forme algorithmique des différents axiomes qui la composent. Mais les théories basées sur l'expérience, comme celles des schémas pragmatiques, sous-tendent que chaque individu construit ses propres méthodes de raisonnement par l'expérience, ce qui semble bien plus difficile à simuler. De même, le développement d'un agent raisonnant à partir de la construction de modèles mentaux [Johnson-Laird 02] semble particulièrement complexe.

De plus, outre le fait que les modèles fondamentaux du raisonnement humain soient toujours débattus, les nombreux biais de raisonnement aujourd'hui identifiés complexifient encore la modélisation d'un joueur synthétique [Noveck 07]. Les notions de fixité fonctionnelle, d'effet de halo, les biais de confirmation, de disponibilité, d'appariement, sont autant de particularités du raisonnement humain qu'un joueur synthétique devrait savoir simuler.

Encore à titre d'exemple, la théorie des marqueurs somatiques, proposée par Antonio Damasio [Damasio 94], cherche à expliquer la place des émotions dans notre système de raisonnement. Selon cette théorie, les émotions nous servent d'heuristiques, et nous permettent de simplifier nos raisonnements à partir du ressenti lié à certaines solutions envisagées. Ce type de théorie semble également très complexe à simuler pour un agent logiciel.

Pour finir, la phase d'action est elle aussi très différente chez l'agent et le joueur humain. L'agent exécute ses actions au moyen d'un appel logiciel, d'une précision parfaite. Le joueur humain doit interagir au moyen d'un contrôleur, spécifique à chaque plateforme de jeu. Il est par exemple admis qu'un même First Person Shooter réalisé pour console nécessite un équilibrage particulier pour pouvoir être joué sur un PC. Sur un ordinateur, les joueurs utilisent une souris, un dispositif de pointage absolu bien plus précis qu'une manette, qui ne fournit qu'un déplacement relatif du système de visée. Pour être capable de remplacer le joueur par un agent, il faudrait être en mesure de simuler les propriétés des différentes interfaces.

Un joueur synthétique reste donc un outil intéressant dans l'étude d'un gameplay, mais ne remplace que très partiellement la mise en place de véritables playtests. Le développement d'un joueur synthétique capable de simuler le raisonnement humain est aujourd'hui une tâche aussi titanesque qu'inaccessible, tant il nous reste à apprendre le raisonnement humain avant d'espérer parvenir à le simuler. Dans la suite de ce document, nous partirons du principe que nos mesures sont réalisées grâce à des tests réalisés avec de vrais joueurs, notre objectif étant plus de décrire la difficulté d'un gameplay que d'en rechercher les éventuelles failles.

Finalement, ces premières expériences mettent en évidence un autre problème fondamental. Dans chacune des expériences réalisées, nous avons mesuré la difficulté à partir d'une

fonction heuristique, propre à chaque jeu : le nombre de pastilles mangées par Pacman, la vitesse de destruction de la base ennemie pour Skywar ou le temps de résolution du casse tête. Rien ne nous permet de valider ces mesures, elles correspondent à une vision intuitive de la difficulté mais sont aussi recevables à priori que n'importe quelle autre fonction de l'état du jeu. Si nous voulons fournir une mesure correcte de la difficulté, il est avant tout nécessaire d'en fournir un modèle le plus générique et acceptable possible, ainsi que d'établir une méthode de mesure valide sous des hypothèses identifiées. Dans le prochain chapitre, nous proposons donc notre modèle de la difficulté d'un jeu vidéo.

Glossary

build Suite de bâtiments, d'améliorations et d'unités construites par un joueur dans un jeu de stratégie temps réel. Un build retrace les premiers instants de jeu, fournissant une méthode pas à pas pour développer la base d'un joueur.. 7

First Person Shooter First Person Shooter, jeu de tir en vue subjective.. 4, 12

Intelligence Artificielle Intelligence Artificielle.. 9

Bibliographie

[Braine 90] M.D.S. Braine. Reasoning, necessity and logic : Developmental perspectives, chapitre The "natural logic" approach to reasoning, pages 135–158. Hillsdale : Lawrence Erlbaum Associates, 1990.

[Damasio 94] Antonio Damasio. Desartes' error : Emotion, reason, and the human brain. Avon Books, 1994.

[Johnson-Laird 02] P. N. Johnson-Laird & R. M. Byrne. *Conditionals : a theory of meaning, pragmatics, and inference.* Psychol Rev, vol. 109, no. 4, pages 646–678, October 2002.

[Noveck 07] Ira Noveck, Hugo Mercier, Sandrine Rossi & Jean Baptiste Van der Henst. Psychologies du raisonnement, chapitre Psychologie cognitive du raisonnement, pages 40–76. De Boeck, 2007.

[Pittman 09] Jamey Pittman. *The Pac-Man Dossier.* Gamasutra : http ://www.gamasutra.com/ (last access 01/2009), 2009.

[Sutton 98] Richard S. Sutton & Andrew G. Barto. Reinforcement learning : An introduction. The MIT Press, March 1998.