Université de Montréal

Sur l'utilisation du langage de programmation Scheme pour le développement de jeux video

par
David St-Hilaire

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Décembre, 2009

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Sur l'utilisation du langage de programmation Scheme pour le développement de jeux video

présenté par:

David St-Hilaire

a été évalué par un jury composé des personnes suivantes:

Stefan Monnier
président-rapporteur

Marc Feeley
directeur de recherche

Pierre Poulin
membre du jury

Mémoire accepté le

RÉSUMÉ

Mots clés: Language de programmation fonctionnels, Scheme, jeux vidéo, programmation orientée objet.

ABSTRACT

Keywords: Functional programming languages, Scheme, video games, object oriented programming.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

blablabla

CHAPITRE 1

INTRODUCTION

L'industrie du jeu vidéo devient de plus en plus importante dans le domaine de l'informatique. Cette croissance est bien reflétée par l'augmentation de 28% des revenus provenant de la vente de jeux vidéo aux États-Unis durant l'année 2007 [2]. La place occupée par l'industrie du jeu vidéo durant cette même année s'estime à 76% du marché de tous les logiciels vendus [3].

L'engouement du marché du jeu vidéo incite les compagnies oeuvrant dans le domaine à repousser les limites de l'état de l'art du développement de jeux. La compétition est féroce et beaucoup d'efforts doivent donc être investis dans la création d'un jeu afin qu'il se démarque de la masse et devienne un nouveau « Blockbuster Hit ». Le président de la compagnie française UbiSoft estime que le coût moyen de développement d'un jeu sur une console moderne se situe entre 20 et 30 millions de dollars [4]. Si l'on estime le salaire moyen d'un artiste ou d'un développeur de jeu vidéo à 60 000$ par année, cela reviendrait à un travail d'environ 300 à 500 années/hommes.

Puisque la création d'un jeu vidéo peut nécessiter autant d'efforts, il semble très intéressant de vouloir tenter de faciliter le développement de ces derniers. Avec autant d'efforts mis en place, même une petite amélioration sur le cycle de développement peut engendrer une diminution énorme des coûts de production et

améliorer la qualité des environnements utilisés par les développeurs.

Jusqu'à ce jour, la grande majorité des jeux vidéo sont écrits à l'aide de langages de relativement bas niveau, comme par exemple en C, C++ ou encore C# [5]. Les langages de bas niveaux sont caractérisés par contact facile au matériel du système utilisé pour exécuter le programme. Ces langages sont généralement utilisés parce qu'ils sont déjà bien établis et que la main d'oeuvre est facilement accessible.

Les langages de haut niveau sont généralement caractérisés par le fait qu'ils font une bonne abstraction du système utilisé et permettent d'exprimer un calcul de manière naturelle. Elles facilitent donc le travail de programmation. Le coût de ces abstractions se répercute généralement en un coût de performance du programme. Dans le passé, la performance était critique dans les jeux vidéo, mais les consoles modernes sont devenues plus performantes que la plupart des ordinateurs personnels. Actuellement la performance n'est donc plus aussi critique. Aussi, les améliorations du domaine de la compilation de langages de haut niveau font en sorte que les performances de ces systèmes sont comparables à l'utilisation de langages de plus bas niveau.

Ainsi, l'utilisation de langages de plus haut niveau pourrait potentiellement améliorer le temps de développement des jeux en permettant aux programmeurs et designers de s'exprimer plus facilement. Les langages dans la famille *Lisp* semblent être de bons candidats en tant que langages de haut niveau. Le langage Scheme [6], un *Lisp* moderne et performant, offre ainsi beaucoup ainsi plusieurs fonctionnalités

de haut niveau. On retrouve, entre autre, du typage dynamique, des fonctions de premier ordre, un système de macro évolué, un accès direct aux continuation du calcul et un système de déboggage très efficace. Ces particularités du langage Scheme sont discutées plus en détail dans le chapitre 3.

Le système Gambit-C [7], l'une des implantations de Scheme les plus performantes [8] sera utilisé pour effectuer les expériences pratiques. Ce système comporte de nombreuses extensions pouvant être très utiles au développement de jeux vidéo. On y retrouve entre autres des tables de hachages et des *threads* (processus légers).

Il semble donc qu'une utilisation judicieuse de ce système a le potentiel de faire bénéficier des projets aussi complexes que le sont les productions de jeux vidéo.

1.1 Problématique

Ce mémoire de maîtrise vise à répondre à la problématique suivante :

> Quelles sont les forces et les faiblesses du langages de programmation Scheme pour le développement de jeux vidéo.

1.2 Méthodologie

Afin de pouvoir répondre à la problématique posée, nous avons étudié les caractéristiques de Scheme et du compilateur Gambit-C, ainsi que les besoins au niveau du développement de jeux vidéo. Par la suite, deux jeux ont été développés

en tentant d'utiliser a profit ces caractéristiques dans le but de raffiner nos approches et les évaluer dans un contexte réel.

Le premier jeu a servi de plate-forme d'exploration permettant d'élaborer une méthodologie qui semble efficace pour le développement de jeux. Afin d'obtenir une telle méthodologie, plusieurs itérations de développement ont été effectuées, chacune permettant d'explorer de nouveaux aspects sur la manière de résoudre les problématiques associées à la création de jeux, par exemple comment arriver à synchroniser des entités dans le jeux ou comment arriver à décrire efficacement un système de détection et de résolution de collisions.

Suite à l'écriture de ce jeu, un deuxième jeu, plus complexe, a été développé afin de consolider les techniques précédemment utilisées et étendre ces techniques dans le cadre de ce nouveau jeu comportant de nouveaux défi, comme l'implantation d'une intelligence artificielle.

1.3 Aperçu du mémoire

Ce mémoire est composé de quatre parties. La première partie est une introduction qui traite de programmation fonctionnelle, du langage de programmation Scheme et des outils de développement disponibles, en particulier le compilateur Gambit-C. Ce chapitre donne un aperçu général de ces langages et permet de saisir les concepts fondamentaux du langage Scheme. Une présentation de l'industrie des jeux vidéo et des défis découlant du développement suit ce chapitre.

La deuxième partie du mémoire porte sur des extensions faites au langage Scheme qui ont été utilisées dans le but d'améliorer le développement de jeux vidéo. On y présente la programmation orientée objet et un système d'objets conçu pour répondre aux besoins de la programmation de jeux vidéo. Un système de coroutines conçu dans les mêmes optiques est également présenté.

La troisième partie du mémoire porte sur l'expérience acquise par l'auteur en effectuant l'écriture de deux jeux vidéo simples, mais possédant suffisamment de complexité pour exposer les problèmes associés à la création de jeux vidéo et comment tirer profit du langage de programmation Scheme pour résoudre ces problèmes. Une présentation des travaux reliés aux résultats présentés suit ce dernier chapitre. On y parle de l'utilisation d'autres langages pour le développement de jeux vidéo et cite des exemples d'utilisation des langages Scheme et *Lisp* dans des jeux vidéo commerciaux et de l'expérience tirée de cette utilisation par les développeurs.

Finalement, une conclusion apporte la lumière sur la problématique exposée dans ce mémoire. Le point sur l'expérience acquise pour le développement de jeux vidéo en Scheme y est fait et les avantages et inconvénients ou problèmes obtenus seront exposés.

CHAPITRE 2

DÉVELOPPEMENT DE JEUX VIDÉO

Les jeux vidéo font parti d'un domaine de l'informatique en pleine effervescence grâce à une demande constante de nouveaux produits.

L'histoire des jeux vidéo possède déjà un demi-siècle de créations de tous genres qui ont contribué à générer l'engouement actuel pour ces derniers. Une brève historique des jeux vidéo sera présentée dans ce chapitre afin de pouvoir illustrer l'évolution des jeux vidéo et de permettre de situer dans le temps où se trouvent les jeux développés pour ce mémoire.

L'industrie du jeu vidéo actuelle est le moteur qui permet aux jeux vidéo de continuer à évoluer et se raffiner. Un aperçu global de l'industrie du jeu vidéo est ainsi donné dans ce chapitre.

Finalement, les besoins au niveau de la programmation de jeux vidéo sont énoncés afin de pouvoir cerner les composantes importantes pour un langage de programmation utilisé pour le développement de jeux.

2.1 Historique

L'histoire des jeux vidéo s'étale sur environ un demi-siècle et est une source importante d'informations qui permet d'exposer la manière avec laquelle les jeux vidéo évoluent. Seulement un bref résumé est présenté afin de donner les grandes

lignes de l'évolution des jeux au cours des cinquante dernières années [9] [10].

2.1.1 Préhistoire 1948-1970

Les jeux vidéo ont fait leur apparition avant même les premiers ordinateurs. En effet, le *Cathode Ray Amusement Device* [11] fit sont apparition en 1948. Il s'agissait d'un jeu rendu sur un tube cathodique simulant le lancement de missiles sur des cibles.

Vers le début des années 1960, quelques jeux ont fait leur apparition sur les premiers ordinateurs universitaires, notamment au *MIT* (Massachusetts Institute of Technology) et à l'université de Cambridge. On y retrouve notamment le jeu *Spacewar !* qui est l'un des premiers jeux multi-joueurs mettant les joueurs en adversité dans leurs vaisseaux spatiaux pouvant lancer des missiles. Ce jeu a d'ailleurs été une motivation pour les premiers efforts de développement du système d'exploitation UNICS (UNiplexed Information and Computing Service) [12].

Avec un intérêt soutenu envers le potentiel de divertissement qu'offrait les jeux vidéo de l'époque, le développement de machines dédiées aux jeux vidéo a ainsi débuté.

2.1.2 Système arcade 1970-1985

En 1971, des étudiants de l'université de Stanford ont réimplanté le jeu *Spacewar !* sur une machine fonctionnant avec de l'argent (de la monnaie). Ce fut la

première machine arcade.

Par la suite, le développement de telles machines est devenu très répandu. En 1972, la compagnie *Atari* fut fondée et démarra l'industrie du jeu vidéo sur arcade avec le jeu *Pong*, un jeu de tennis de table permettant à un joueur de se mesurer à un autre joueur dans une simulation de tennis de table qui demeure toujours très célèbre.

Ce fut alors l'âge d'or des machines d'arcade où l'on a créé beaucoup de jeu au contenu diversifié. Parmi ceux-ci, *Space Invaders* (1978) et *PacMan*(1980) furent extrêmement populaires.

Malgré que les jeux d'arcades étaient très simples, ils étaient beaucoup appréciés par les joueurs qui tentaient toujours de se surpasser.

On retrouve toujours des salles d'arcades de nos jours, mais celles-ci ont perdu de la popularité puisque les consoles de jeu se retrouvent dans nos salons.

2.1.3 Premières consoles 1972-1984

La première console de jeu vidéo, le *Magnavox Odyssey* fit son apparition en Amérique du Nord en 1972. Cette console permettait aux joueurs de jouer sur la télévision assis confortablement dans leurs salon. Elle permettait aussi d'insérer des jeux sous forme de cartouches. Un total de 28 jeux ont été disponibles pour cette console, qui malgré l'absence de périphérique audio, a été vendue pour environ 300 000 exemplaires.

Plusieurs autres consoles ont par la suite été créées. Allant d'une version console de *Pong* jusqu'à des consoles plus puissantes comme le *ColecoVision* qui démarrèrent l'ère des consoles 8-bit. La vocation principale de ces consoles n'étaient pas d'innover dans le développement de jeu, mais plutôt de rendre disponible les jeux d'arcades populaires sur les consoles.

2.1.4 Ordinateurs personnels 1977-...

Les premiers ordinateurs personnels firent leur apparition vers la fin des années 1970, notamment l'ordinateur *Apple II* produit par *Apple Inc.* Ces ordinateurs personnels offraient plus de puissance que les consoles de l'époque et offraient la possibilité aux amateurs de créer leurs propres jeux.

Les jeux d'ordinateurs étaient distribués sur beaucoup de média différents. La distribution allait de cassettes, aux disquettes, en passant bien sûr par des échanges postaux de code sources.

En 1982, le jeux *Lode Runner* fut développé pour les ordinateurs *Apple II*. Ce jeux d'action fut l'un des premiers jeu comprenant un éditeur de niveaux.

Aussi, le jeu *Rogue* fut créé durant les années 1980, pour les premiers systèmes Unix. Il fut le pionnier d'un nouveau genre de jeu (surnommé *Roguelike*) qui différait beaucoup des jeux d'actions retrouvés en sales d'arcades ou sur consoles. Il présentait une interface visuelle très minimaliste. La narration, qui était un point central du jeu, était effectuée de manière textuelle et le joueur interagissait aussi

de manière textuelle avec le jeu.

Les jeux d'ordinateurs ont longtemps été supérieurs aux jeux de consoles puisque les ordinateurs étaient toujours à la fine pointe de la technologie, et les consoles traînaient un peu derrière en terme de technologies. Ainsi, on retrouvait des jeux ayant de meilleurs graphiques ou utilisant des périphériques plus variés sur les ordinateurs personnels. Ainsi, les jeux d'ordinateurs existent dans un monde parallèle à celui des consoles et donc, ne se livraient pas de compétition réelle.

Par contre, avec l'avènement des consoles modernes qui sont plus performantes que la plupart des ordinateurs personnels, cet énoncé n'est plus valide. Il en résulte que les jeux sur des consoles actuelles surpassent souvent les jeux d'ordinateurs.

2.1.5 Consoles portables 1980-...

Les toutes premières consoles portables furent développées par la compagnie *Mattel Toys* qui créèrent les jeux *Auto Race* et *Football* qui étaient distribués sur des consoles de la taille d'une calculatrice, ne nécessitant pas de téléviseurs externes à la console et étaient dédiées à un seul jeu. Ces consoles furent un succès rapportant plusieurs centaines de millions de dollars à leurs créateurs.

Par la suite, les grandes compagnies du jeu vidéo comme *Nintendo* et *Bandai* se sont intéressées à de telles consoles et produisirent plusieurs exemplaires des consoles *Game & Watch* qui contenaient des succès d'arcades tel *Mario's Cement Factory* et *Donkey Kong Jr.*

Le même concepteur de ces consoles a par la suite fusionné ces dernières avec le contrôleur du *Nintendo Entertainement System*(NES) pour obtenir la première console à grand succès : le *GameBoy*. Cette console qui offrait un écran monochrome fut vendue à 118 million d'exemplaire dans le monde. Le jeu le plus vendu sur cette console fut nulle autre que le jeu *Tetris* qui a vendu environ 33 millions d'unités.

Les consoles portables ont continuées d'évoluer grandement et elles sont actuellement équivalentes aux consoles d'une ou deux générations précédentes. Par exemple, le Sony *PlayStation Portable* rivalise en matière de puissance de matériel à la précédente console de Sony, le *PlayStation 2*. On peut donc développer des jeux très complexes sur ces consoles portables, mais ils ne peuvent toujours pas rivaliser avec les jeux sur consoles ou d'ordinateurs.

2.1.6 Consoles intermédiaires 1984-2006

Au début des années 1980, plusieurs consoles étaient disponibles sur le marché, mais une saturation de mauvais jeux et des problèmes de gestion ont fait en sorte que l'industrie du jeux vidéo a connu une grande dépression vers en 1983. Par exemple, il y a eu une plus grande production d'unités du jeu *PacMan* qu'il n'y a eu de console d'Atari vendues.

Cette dépression a pris subitement fin avec la sortie de la console produite par *Nintendo*, le *Nintendo Entertainement System* (NES) qui connu un succès fulgurant en vendant 62 millions de consoles dans le monde. La grande qualité des jeux

produits a certainement favorisé l'adoption de cette nouvelle console. On retrouve entre autres des jeux de tout genre comme le jeu de plate-forme *Super Mario Bros*, le jeu d'aventure *The Legend of Zelda* et le jeu d'action aventure *Metroid*.

Par la suite, la compagnie SEGA sortit un compétiteur sérieux au NES, le *Master System*, qui rivalisait en terme de puissance matérielle et de prix. Depuis ce temps se livre la guerre des consoles où lorsqu'une compagnie développe un nouveau jeu ou une nouvelle console, les compétiteurs ne tardent pas à produire un jeu ou console équivalent pour les utilisateurs de leurs produits.

2.1.7 Consoles modernes 2005-...

Les consoles modernes sont généralement conçues avec du matériel à la fine pointe de la technologie et sont adaptées un certain public cible.

Au moment de l'écriture de ce mémoire, on retrouve la console *Wii* de Nintendo conçue pour un public constitué de joueurs occasionnels grâce au contrôleur innovateur de cette console permettant de jouer en effectuant des mouvement plus naturels.

D'un autre côté compétionnent le *PlayStation 3* de Sony et le *XBox 360* de Microsoft qui cherchent à convaincre les joueurs plus assidus d'utiliser leur système en offrant des consoles à la fine pointe de la technologie et à prix très raisonnable en comparaison aux prix des ordinateurs de jeux équivalents. Par exemple, le *PlayStation 3* utilise une architecture *Cell* qui comprend une unité principale de 3,2 GHz

et 8 sous processeurs moins puissant permettant d'exécuter plusieurs processus en parallèle. Ce dernier possède aussi une carte graphique très puissante comprenant une mémoire vidéo de 256 Mo de mémoire vidéo.

2.2 Industrie du jeu vidéo

Tel que mentionné mentionné dans le chapitre 1, l'industrie du jeu vidéo est très importante et génère des milliards de dollars de revenus par années. Ce profit provient d'une grande diversité de joueurs, allant de jeunes enfants jusqu'à leurs grands-parents avec une population féminine presque aussi importante que celle masculine. Il en résulte donc qu'une grande diversité de jeux et de plates-formes de jeux se retrouvent sur le marché.

La grande quantité de jeux qui ont été produits a fait en sorte que caractéristiques de qualité sont devenues essentielle afin qu'un jeu vidéo puisse se comparer aux jeux déjà existant et ainsi avoir une chance d'être adopté par les consommateurs. Ces attentes du consommateur peuvent se traduire essentiellement par les besoins suivant :

- Exposer un *gameplay* amusant
- Offrir de beaux rendus graphiques et un affichage visuel agréable
- Avoir une composante multi-joueurs
- Optionnellement contenir une narration

Le *gameplay* d'un jeu représente le facteur de plaisir qu'un joueur obtient en jouant à un jeu. Ce facteur est souvent découplé en plusieurs sous catégories comme la diversité et la quantité de niveaux, la difficulté, la réponse des contrôles du jeu, etc...

Ces besoins semblent simple, *a priori*, mais impliquent beaucoup de travail et imposent des contraintes sévères au développeurs de jeux vidéo.

Afin qu'un jeu puisse offrir un *gameplay* intéressant aux joueurs, une étroite collaboration entre les développeurs et les designers doit être établie. Cela implique aussi de faire beaucoup d'essais de possibilités en testant sur des prototypes et en effectuant de nombreux cycles de développement du jeu.

En ce qui concerne le rendu graphique du jeu, en plus des designers du jeu, c'est aussi avec les artistes que les développeurs doivent s'accorder dans le but de concevoir un moteur de rendu répondant bien aux besoins de leurs créations et, leurs créations doivent être bien sûr faites en respectant les contraintes imposées par le matériel où sera déployé le produit. Ainsi, le moteur de rendu doit pouvoir être facilement extensible afin de pouvoir accommoder facilement les nouvelles idées et les nouveaux concepts à intégrer.

Une composante multi-joueur, tout dépendant si elle implique des parties faites sur le réseau ou non, pourrait nécessiter une grande rigueur de programmation afin d'assurer une stabilité de connections et empêcher les joueurs de tricher. Ces sujets ne sont pas discutés dans ce mémoire.

La diversité des jeux développés a mené à une classification des jeux. Les principaux genres de jeux sont :

- Action : Jeux rapides demandant souvent de l'habileté de la part des joueurs.
- Stratégie : Jeux à rythme plus lent, exigeant une réflection plus profonde de la part des joueurs.
- Jeux de Rôles : Jeu où la narration de l'histoire occupe une place importante et où les personnages de l'histoire subissent une évolution graduelle en fonction du temps.
- Simulation : Jeux qui tentent de représenter fidèlement des phénomène réels comme la conduite automobile, des sports, etc...
- *Casual* : Jeux simples visant à être utilisés par des joueurs occasionnels. Le jeu *Tetris* est considéré comme appartenant à ce genre.

Il existe d'innombrables autres genres et d'hybrides de ces catégories. Notre but ici est simplement de donner une idée de la diversité du contenu des jeux vidéo produits.

2.3 Contraintes de programmation

Le développement de jeu vidéo implique une programmation sous des contraintes sévères afin que le jeu respecte les normes du marché et respecte les attentes des joueurs.

Une contrainte importante portant sur les jeux vidéo est reliée à l'efficacité algorithmique résultant du programme développé. En effet, la fluidité d'un jeu est critique face à l'immersion du joueur dans l'univers créé par le jeu dans certains genres. Les jeux doivent donc être des programmes efficaces de manière à inviter le joueur à entrer le plus possible dans la narration du jeu.

Une autre contrainte importante reliée au développement de jeu est la modularité du code produit. En effet, afin que les efforts placés dans la production d'un jeu vidéo puissent être réutilisés dans les productions subséquentes, des abstractions doivent être bien faites afin de faciliter la réutilisation de ces dernières. Souvent, pour obtenir des performances adéquates, les développeurs doivent sacrifier la modularité. La conséquence est que chaque jeu représente un énorme travail car peu de code peut être réutilisé.

2.3.1 Fluidité

2.3.1.1 Taux de rafraîchissement

Une contrainte très importante dans le développement d'un jeu vidéo est la fluidité de celui-ci. Lorsqu'un film est projeté sur un écran de cinéma le taux de rafraîchissement de l'image est d'environ 30 trames par secondes. Par contre, les images captées par les caméra contiennent du « flou de mouvement », effet créé par le mouvement s'étant produit durant la capture de cette image. Ce flou permet à notre cerveau d'estimer ou de faire l'extrapolation du mouvement dans une trame

et donc de croire l'illusion créée par la projection des images.

Par contre, dans un jeu vidéo, les images rendues sur l'écran sont parfaitement nettes et ne contiennent donc pas ce flou de mouvement, ce qui vient réduire la crédibilité de l'illusion faite par la succession des images à l'écran. Il faut donc augmenter le taux de rafraîchissement à environ 60 trames par secondes pour obtenir le niveau de crédibilité d'une projection cinématographique [13].

Ceci étant dit, un taux de rafraîchissement de 60 trames par secondes n'est pas nécessaire pour tous les jeux. Cela varie grandement avec la nature des jeux. Par exemple, un jeu de stratégie où les joueurs jouent à tours de rôles possède peu d'animations et donc pourrait très probablement être satisfaisant avec un taux de rafraîchissement de 30 trames par secondes ou moins.

Aussi, il est possible que les designers du jeu acceptent de réduire volontairement le taux de rafraîchissement afin d'avoir plus de temps pour rendre une image. Un jeu possédant un taux de rafraîchissement de 60 trames par secondes implique que les images sont rafraîchies tous les 16 millisecondes, ce qui ne laisse pas beaucoup de temps pour effectuer le calcul du moteur du jeu en plus du rendu de l'image. Heureusement, les cartes vidéo modernes sont capables de faire une bonne partie du rendu laissant ainsi le ou les processeurs principaux libres pour exécuter le moteur du jeu.

Il n'en demeure pas moins que toute la logique du jeu doit être aussi efficace que possible afin que le processus de rendu des trames soit transparent au joueur.

2.3.1.2 Réponse quasi temps réelle

Une autre implication de la fluidité requise par un jeu vidéo est le délai de réponse face aux entrées du joueur. Ce délai aussi doit être aussi faible que possible pour donner l'impression que le personnage dans le jeu ne soit qu'une extension de la volonté du joueur. Ainsi le traitement des entrées du joueur se doit aussi d'être très efficace afin de ne pas encombrer le moteur du jeu qui n'a déjà pas beaucoup de temps pour effectuer son travail.

2.3.2 Modularité

En plus de devoir être efficacement implanté, un jeu vidéo moderne se doit d'être aussi modulaire que possible. Pour y arriver, le couplage entre les différentes composantes de ces dernières doit être faible.

Le développement de logiciel favorisant la modularité est une qualité standard en génie logiciel, mais elle vient très bien se marier avec le développement des jeu, surtout de jeux modernes, car ce sont des projets de très grande envergure impliquant beaucoup de programmeurs. Il en résulte qu'un bon design modulaire permet de bien séparer les tâches à effectuer de manière parallèle par des équipes spécialisées. Si le projet est bien géré, cela pourrait certainement réduire le coût de développement de tels projets.

Aussi, une bonne modularité permet de créer des composantes qui sont cohésives et qui permettent donc d'être réutilisées par la suite. Une réutilisation de compo-

santes complexes tel un moteur de physique évite de faire un travail supplémentaire non trivial pour chaque nouveau projet. Ainsi, il est clair que l'effort supplémentaire investi pour bien modulariser les composantes d'un jeu seront rentables pour le développement des prochains jeux.

Bien sûr, toute abstraction possède son coût, notamment en coût de performances. Ainsi, il faut bien pouvoir estimer les endroits où l'utilisation de modules externes n'apportera pas de trop grands impacts négatifs de performances.

Il est clair qu'un langage de programmation ayant un bon potentiel d'abstraction, comme c'est généralement le cas pour les langages de haut niveau, facilite cette tâche. Un système orienté objet ou un bon système de module permettraient certainement aussi de bien modulariser les composantes d'un jeu.

2.3.3 Malléabilité

La malléabilité d'un logiciel pourrait se définir comme la facilité avec laquelle ce dernier peut être modifié ou transformé. Cette qualité est importante pour le développement de jeu puisqu'un tel processus implique généralement de faire beaucoup de prototypage. Le code d'un jeu se doit alors d'être très malléable afin d'aider à faire des changements rapidement au moteur du jeu pour tester de nouvelles idées, sans ajouter un trop grand coût pour le faire.

La modularité du jeu aura un effet positif sur la malléabilité en permettant de modifier les mécanismes internes des composantes sans trop affecter le reste du

programme. Par contre, un langage de programmation offrant une bonne puissance d'abstraction serait tout aussi efficace, car il permet de pouvoir ajouter facilement de nouvelles abstractions où les besoins de changement apparaissent.

2.4 Conclusion

Dans ce chapitre, l'industrie du jeu vidéo, avec son histoire, ont été mis en lumière afin d'exposer l'évolution des jeux vidéo et de motiver l'intérêt de travailler à améliorer le développement de ceux-ci.

Les jeux vidéo sont passés rapidement d'idées farfelues comme ce fut le cas pour le *Cathode Ray Amusement Device* à une industrie complète générant des milliards de dollars annuellement. Il y a un grand intérêt à vouloir diminuer le coût de développement de ceux-ci.

Les coûts de développement de jeux modernes sont de l'ordre des millions de dollars et impliquent des dizaines voir des centaines d'artistes, de programmeurs et de designers. Ainsi, même de toutes petites accélérations d'un cycle de développement, pourrait avoir un l'impact sur le travail de beaucoup de gens et ainsi se traduire en de grandes économies sur les coûts de développement.

Les contraintes impliquées au niveau de la programmation par le développement de jeux vidéo ont été étudiées. Ces dernières se résument par les concepts de fluidité, modularité et de malléabilité du jeu vidéo. Le respect de la contrainte de fluidité résulte en une bonne immersion pour le joueur, qui est nécessaire afin que le jeu

soit considéré comme « jouable ». La modularité facilite le développement collaboratif et parallèle tout en permettant d'améliorer la réutilisation des composantes communes à plusieurs jeux. L'aspect de la malléabilité du code permet d'accélérer le prototypage du jeu.

Il semble donc que le choix d'un langage permettant de pouvoir facilement répondre à ces contraintes pourrait certainement faciliter le développement de jeux vidéo et ainsi, engendrer des économies substantielles aux compagnies de l'industrie et rendant plus accessible le développement de jeux au plus petites compagnies possédant des budgets moins importants.

CHAPITRE 3

LE LANGAGE SCHEME

Dans le cadre d'un projet donné, le choix d'un langage de programmation a beaucoup d'influence sur la structure du code écrit pour sa réalisation. Plusieurs facteurs influencent ces différences. Par exemple, un langage à typage statique résultera en du code qui a une structure adaptée aux besoins du moment, mais qui sera beaucoup moins malléable par la suite. Ainsi, afin d'optimiser les efforts de développement pour un projet, une bonne analyse des besoins exprimés par le projet est de rigueur pour pouvoir faire le choix d'un langage répondant le mieux possible à ceux-ci.

En se basant sur les besoins et les contraintes énoncés dans le chapitre 2, nous proposons l'utilisation du langage de programmation Scheme comme outil principal de développement de jeux vidéo. Il s'agit d'un langage de très haut niveau qui semble être un candidat idéal pour de permettre un développement efficace de jeux vidéo. En effet, en plus de posséder plusieurs implantations très performantes, ce langage offre une très grande quantité de concepts de programmation de très haut niveau. Parmi ceux-ci, l'on retrouve entre autre les concepts de la programmation fonctionnelle qui offre un grand potentiel d'abstractions au niveau de la structure du code, un système de macro évolué qui permet d'augmenter Scheme avec des langages spécifiques au domaine, l'accès aux continuations du calculs qui permettent une manipulation très puissante du flot de contrôle du programme et un système

de gestion mémoire automatique.

Ce chapitre donne un historique du langage Scheme afin de mettre en situation le langage et donner un aperçu de son origine. Par la suite, une brève explication des concepts de base du langage est donnée dans le but d'illustrer la simplicité du langage Scheme. Plus d'informations sur les bases de Scheme peuvent être obtenues dans la littérature [6] [14]. Finalement, les particularités du langage sont mises en lumière pour rendre plus clair l'ampleur de celles-ci dans le contexte du développement de jeux vidéo.

3.1 Héritage *Lisp*

Le langage Scheme est une forme épurée du tout premier langage de programmation de haut niveau, le langage *Lisp* . Ainsi, pour illustrer les origines de Scheme, une courte histoire du langage *Lisp* est présentée. Par la suite, les langages de programmation qui furent inspirés du langage Scheme sont mentionnés en expliquant brièvement leurs racines commune à Scheme et leurs principales disparités.

3.1.1 Histoire de *Lisp*

Le langage *Lisp* est le tout premier langage de programmation de haut niveau qui fut conçu et est le deuxième plus vieux langage de programmation toujours utilisé. Sa première implantation fut réalisée en 1958 dans un laboratoire d'intelligence artificielle du MIT (*Massachusetts Institute of Technology*) sous la direction

de John McCarthy. Le langage original était relativement du *Lisp* d'aujourd'hui mais introduisait déjà des concepts fondamentaux comme l'utilisation de listes pour représenter du code, de la programmation fonctionnelle ainsi que le concept de gestion de mémoire automatique. Un article décrivant le langage *Lisp* comme un langage de programmation ainsi que comme un formalisme de définition de fonctions récursives fut publié un peu plus tard [15].

Le langage fut aussitôt adopté par la communauté de recherche en intelligence artificielle, puisque l'utilisation de traitement symbolique permettait de pouvoir facilement développer de nouveaux algorithmes dans ce domaine.

Lisp évolua par la suite en de nombreux dialectes différents, chacun apportant ses propres extensions au langage. Vers le début des années 1980, un comité d'unification de ces multiples dialectes est formé pour donner naissance au standard *Common Lisp* . En 1984, Guy Steele publia un livre [16] donnant une description détaillée du standard établi par la communauté *Lisp* . Ce standard fut par la suite la fondation d'une standardisation officielle créé pour le langage *Common Lisp* [17]. Il en résulte donc une spécification très complexe du langage qui s'est trouvé gonflée par les multiples dialectes lui ayant donné naissance. Ainsi, *Common Lisp* possède beaucoup de bagages, et il devient difficile pour un programmeur de pouvoir maîtriser tous les concepts de ce langage.

3.1.2 Avènement de Scheme

Peu avant la standardisation de *Common Lisp* , Guy Steele et Gerald J. Sussman conçurent la spécification d'un nouveau dialecte du langage *Lisp* appelé Scheme [18]. L'origine du développement de Scheme provient de la tentative de compréhension d'un formalisme modulaire d'agents en intelligence artificielle [19] [20]. Ils ont donc bâti un interprète minimaliste basé sur *Lisp* afin de pouvoir comprendre ce formalisme qui s'exprimait mal avec les langages disponibles à l'époque. Ce petit interprète supportait ainsi une portée statique appropriée pour la description d'acteurs, divergeant ainsi de *Lisp* en remplaçant de simples fonctions récursives par les fermetures du lambda calcul. Les deux auteurs furent très surpris de constater que le langage expérimental créé répondait parfaitement à leurs besoins et se trouvait à être très simple, beaucoup plus simple que ce qu'ils avaient anticipé.

Scheme devint ainsi populaire et est présentement l'un des deux dialectes les plus utilisés de *Lisp* , en concurrence avec *Common Lisp* . Même après avoir subi 5 révisions, le langage Scheme demeure très simple avec un standard ne comptant qu'environ une trentaine de pages [6]. Il en résulte qu'on ne retrouve actuellement pas moins d'une cinquantaine d'implantations disponibles, dont une douzaine sont considérées comme étant des implantations majeures. Ces implantations majeures offrent non seulement plusieurs extensions du langage, mais aussi de bonnes performances grâce à l'implantation d'optimisations de compilation avancées.

Le langage Scheme impose une philosophie de minimalisme à ces programmeurs

en ne leur fournissant que la base dans le standard. Cette philosophie est souvent appréciée car elle mène à utiliser ce qui est vraiment nécessaire et faire soi même ce qui n'est pas disponible, créant ainsi une bonne autonomie chez les programmeurs Scheme. De plus, le langage est accompagné d'une centaine librairies externes de traitement générique dénommées *Scheme Request For Implementation* (SRFI [21]).

3.1.3 Langages inspirés de Scheme

Plusieurs langages de programmation récents furent inspirés des langages *Lisp* . Pour la plupart, il s'agit de langages de script comme JavaScript [22], Ruby [23] ou Perl [24]. Ces langages s'inspirent notamment de l'aspect dynamique de *Lisp* afin de pouvoir être interprétés. Ils utilisent aussi une gestion de mémoire automatique afin de simplifier leur utilisation. Finalement, certains d'entre eux offrent des fonctions en tant que données de première classe, permettant ainsi l'utilisation de la programmation fonctionnelle.

Ces langages demeurent par contre toujours des sous-ensembles, en terme de fonctionnalités, des langages *Lisp* . Ils sont devenus populaires pour leur simplicité d'utilisation, mais un bon apprentissage de Scheme pourrait certainement être plus bénéfique à long terme. Entre autre, Scheme est un langage interprété et compilé, permettant de développer des applications très efficaces. Il en résulte que le code peut être développé de manière dynamique en utilisant une *Read Eval Print Loop* (*REPL*) afin de pouvoir tester et corriger facilement un programme développé.

La présence d'une *REPL* permet aussi un mode de récupération d'erreurs très dynamique et donnant accès à l'évaluation d'expressions Scheme lorsqu'une erreur se produit dans une telle boucle.

3.2 Introduction au langage

Tout en bénéficiant du riche héritage des langages *Lisp* , Scheme demeure un langage épuré et très simple. Cette section vise à présenter les bases du langage Scheme et de son implantation par le système Gambit-C [7] afin de familiariser le lecteur avec non seulement la syntaxe du langage, mais aussi les fonctionnalités de base et les extensions au langage fournies par Gambit-C.

3.2.1 Syntaxe

Les langages de la famille *Lisp* dont Scheme fait parti adoptent une notation préfixe parenthésée. Il en résulte que la syntaxe du langage est consistante et simple. La figure 3.1 illustre un exemple simple de calcul arithmétique.

```
(+ (expt 2 5) (sin (/ 3.141592653589793 2)))
```

33.

FIGURE 3.1 – Exemple simple d'appels de fonctions en Scheme

Cette syntaxe apporte plusieurs bénéfices au langage. Puisque chaque appel de fonction (ou de forme spéciale) est parenthésé, il en résulte qu'un programme Scheme est très structuré et peut être manipulé efficacement en utilisant des ou-

tils de développement performant tel *VI* ou *Gnu Emacs*. De plus, Scheme utilise *la même syntaxe* pour les données d'un programme que pour le programme en lui même. Il devient ainsi possible concevoir des programmes qui créent des programmes Scheme. C'est ce qui constitue la base du système de macro de Scheme. On dit ainsi qu'un programme Scheme est représenté par des expressions symboliques, fréquemment appelées S-expression. La distinction dans un programme Scheme entre une donnée et le programme se fait par l'entremise des formes spéciales `quote` et `quasiquote` spécifient respectivement que l'expression doit être entièrement ou en partie traité comme une donnée et donc, ne doit pas être évaluée comme un programme. La figure 3.2 donne deux exemples d'utilisation de ces formes spéciales. Le premier exemple illustre l'utilisation de `quote` qui fait en sorte que la liste spécifié est considérée comme une donnée et non un appel de la fonction `abc`. Le deuxième exemple utilise la forme `quasiquote` qui permet de spécifier des données dont certaines parties peuvent être évaluées à l'aide de la forme spéciale `unquote`. Les formes spéciales `quote`, `quasiquote` et `unquote` peuvent être aussi utilisées en utilisant les caractères respectifs `'`, `` ` `` et `,`.

```
(+ 1 (list-ref '(abc 123 "toto") 1))
```

124

```
(list-ref `(abc ,(+ 122 1) "toto") 1)
```

(abc 123 "toto")

FIGURE 3.2 – Exemples d'utilisation de listes sous forme de S-expression

3.2.2 Aperçu des fonctionnalités requise par le standard

Le standard du langage Scheme [6] est très minimaliste, mais requiert toutes les fonctionnalités de base requises par un langage de haut niveau comme Scheme. On y spécifie que Scheme est un langage typé dynamiquement, *i.e.* que les vérifications de types sont faites durant l'exécution d'un programme, et non durant la compilation. Cette particularité fait en sorte que Scheme est un langage qui peut être facilement interprété ou compilé. Les types de base du langage sont tous disjoints entre eux (aucun objet ne peut appartenir a plus d'un de ces types). Ces types sont vérifiés par les prédicats `boolean?`, `pair?`, `symbol?`, `number?`, `char?`, `string?`, `vector?`, `port?`, `procedure?` et `null?`. Chacun de ces types sont documentés et possèdent des fonctions standards de création et d'utilisation. Entre autre, la forme spéciale `lambda` permet la création de fermeture telle qu'elles existent dans le lambda calcul. La figure 3.3 illustre l'utilisation des quelques uns parmi ces types de base.

```
((lambda (x y) (+ (car x) (vector-ref y 2))) '(1 2 3) '#(4 5 6))
```
7
```
(substring (list->string '(#\ a #\ l #\ l #\ o #\ !)) 0 3)
```
"all"

FIGURE 3.3 – Exemples d'utilisation des type de bases de Scheme

3.3 Programmation fonctionnelle

Le langage Scheme est à la base un langage fonctionnel. Cette famille de langage de programmation est caractérisée par une programmation centrée sur l'appel de fonctions dites avec une transparence référentielle, où la valeur de retour d'une fonction dépend uniquement des paramètres et non de l'état du système. Ce type de programmation fait contraste à la programmation impérative qui est centrée sur la notion d'état et de mutation de l'état de l'exécution d'un programme. Les programmes écrits en utilisant des langages sont généralement beaucoup plus faciles à comprendre, ce qui attire un grand nombre de chercheurs à contribuer au développement et l'utilisation de tels langages dans le milieu commercial.

3.3.1 Programmation fonctionnelle pure

Une programmation dite purement fonctionnelle respecte de manière stricte le paradigme de la programmation fonctionnelle, *i.e.* que les fonctions sont des entités mathématique qui ne sont pas affectées par l'état du système. Ce type de programmation est originaire d'avant même l'apparition des premiers ordinateurs. C'est dans les années 1936 qu'Alonzo Church a introduit le formalisme mathématique de définition de fonctions mathématique dénommé lambda calcul non typé [25]. Ce formalisme introduit un formalisme de définition de fonctions qui sert de base aux langages de programmation fonctionnels. La figure 3.4 illustre la définition de la fonction `flip` définie en Scheme dans la figure 3.6.

$$\lambda f \to \lambda y \to \lambda x \to fxy$$

FIGURE 3.4 – Définition de la fonction `flip` en lambda calcul non typé

Ces fonctions sont considérées comme pures car elles sont exemptes de mutations de l'état du calcul. Ainsi, le résultat de l'application d'une fonction (appellée β-reduction en lambda calcul) ne dépend que des paramètres de ces dernières.

Une caractéristique importante présente dans le lambda calcul est la présence de fonctions de première classe, *i.e.* que les fonctions sont des objets appartenant aux valeurs du langages. Elles peuvent donc être créées et manipulées comme tout autre valeurs. Ces fonctions sont aussi appelées fermetures, car elles implantent une portée statique de variable, *i.e.* qu'elles conservent l'environnement d'exécution dans lequel elles se trouvaient lors de leur création. Cette capacité de retenir l'environnement est très puissante et permet l'utilisation de fermeture comme structure de données. Par exemple, la figure 3.5 illustre l'implantation de paires similaire à celles retrouvées dans Scheme en lambda calcul.

$$\begin{aligned} cons &= \lambda car \to \lambda cdr \to \lambda f \to fxy \\ car &= \lambda pair \to pair(\lambda car \to \lambda cdr \to car) \\ car &= \lambda pair \to pair(\lambda car \to \lambda cdr \to cdr) \end{aligned}$$

$$car(cons12) \rightsquigarrow 1$$

FIGURE 3.5 – Implantation de paires en utilisant des fermetures du lambda calcul

Comme déjà mentionné dans la section 3.1, le lambda calcul a donné naissance

à une famille de langages dont la racine est le langage *Lisp* . Dans cette famille, les langages ML [26] et Haskell [27] sont parmi les plus connus de ceux qui adhèrent fidèlement au modèle de calcul de Alonzo Church. Le langage Scheme est lui aussi un langage fonctionnel, mais il n'est pas considéré comme étant pur puisque les mutations de l'état du programme sont permises. En effet, des opérateurs tels que `set!`, `set-car!` ou encore `vector-set!` permettent la modification de liaison ou du contenu de cellules mémoires. Ainsi, Scheme partage le meilleur des deux mondes en laissant la liberté au programmeur de choisir le paradigme de programmation qui lui apparaît le plus approprié. Par contre, un abus de mutation est considéré comme étant de très mauvais style en Scheme.

3.3.2 Programmation fonctionnelle en Scheme

Le lambda calcul ouvre la porte à la création d'abstractions très intéressantes : les fonctions d'ordre supérieure. Celles-ci reçoivent des fonctions en paramètre, les manipulent et en retournent de nouvelles. La figure 3.6 illustre un exemple d'écriture et d'utilisation de fonctions d'ordre supérieur. La première fonction définie prend une fonction en paramètre et applique cette fonction sur chacun des éléments de la liste passée comme deuxième paramètre. La deuxième fonction prend une fonction à deux paramètres comme premier argument et prend le deuxième paramètre à appliquer à cette fonction en deuxième argument, dans le but de faire une application partielle de cet argument à cette fonction. L'exemple

utilise ensuite ces deux fonctions afin de soustraire un à chaque élément de la liste spécifiée.

```
(define (map f lst)
    (if (pair? lst)
        (cons (f (car lst)) (map f (cdr lst)))
        '()))
(define (flip f y)
    (lambda (x) (f x y)))
(map (flip - 1) '(1 2 3 4 5))
```

(0 1 2 3 4)

FIGURE 3.6 – Exemple de création et d'utilisation de fonctions d'ordres supérieures.

L'utilisation de fonctions d'ordre supérieur permet l'abstraction d'algorithmes de manière très générique. Une autre fonction d'ordre supérieur très répandue est la fonction `foldl` qui permet d'utiliser chacun des éléments d'une liste pour calculer un résultat unique. La figure 3.7 illustre l'utilisation de cette fonction pour faire la somme et le produit des nombres dans une liste.

```
(define (foldl f acc lst)
    (if (not (pair? lst))
        acc
        (foldl f (f acc (car lst)) (cdr lst))))
(foldl + 0 '(1 2 3 4 5))
```

15

```
(foldl * 1 '(1 2 3 4 5))
```

120

FIGURE 3.7 – Fonction d'ordre supérieure `foldl`.

Ainsi, l'utilisation de fonctions en tant que données de premières classes per-

met la création de puissantes abstractions algorithmiques. De plus, l'utilisation des paradigmes de programmation fonctionnelle facilite beaucoup la modularisation du logiciel. En effet, puisque peu ou pas de mutations d'état sont utilisées, il devient alors facile de pouvoir combiner différents modules, sans avoir d'interférences dans l'état du programme, ce qui n'est pas nécessairement le cas avec des modules impératifs dépendant directement de l'état du programme pour leur exécution.

3.4 Macros

Les macros d'un langage de programmation sont des systèmes utilisés pour faire des abstractions qui sont résolues durant la compilation d'un fichier source. Dans sa forme la plus élémentaire, un système de macro peut être implanté sous la forme d'un pré-processeur de code qui ne fait que du remplacement de patrons par une expansion textuelle directe. C'est ce genre de système de macro que l'on retrouve entre autres dans le langage C. Ce type de macro, quoique très simple, permet de faire déjà beaucoup d'abstractions allant des abstractions d'optimisations à des expansions conditionnelles de code permettant de développer des librairies sur plusieurs plates-formes. La figure 3.8 illustre des exemples de macros C. Dans cet exemple, on défini la macro `swap` qui interchange la valeur de deux variables entre elles. Le résultat de l'expansion sera donné par la substitution de l'appel de la macro par sont corps où les occurrences des paramètres sont intégralement substitués par les paramètres actuels, soient `a` et `b`.

```
#define swap(x,y) do { int tmp; tmp = x; x = y; y = tmp; } while(0)
int main(){
  int a = 1; int b = 2;
  swap(a,b);
  return a - (b<<1);
}
```

FIGURE 3.8 – Exemple de macro C avec expansion conditionnelle.

De telles macros comportent beaucoup de problèmes potentiels. Non seulement, elle ne sont limitées qu'a faire des remplacements syntaxiques très primitifs, mais aussi sont souvent la source de problèmes reliés au changement du contexte syntaxique des arguments de la macro ou à une expansion dans un contexte qui n'était pas prévu. Ces problèmes sont souvent très difficiles à trouver car ils ne surviennent uniquement lorsque le programme est compilé, bien après l'expansion des macros. Il en résulte que le code compilé ne contient aucune trace des appels macros et donc le compilateur donne souvent des erreurs qui ne sont pas reliées avec la vraie source du problème, soit la définition de la macro en question.

L'utilisation d'expressions symboliques pour représenter de la même manière les données d'un programme et le code source de celui-ci en Scheme la porte à l'implanter un système de macro très puissant, un système de macro procédural. Plusieurs de ces systèmes d'expansion de macro sont disponibles en Scheme. Le plus simple d'entre eux est celui qui provient directement de *Lisp* , soit la forme spéciale `define-macro`. Avec cet expanseur, une macro Scheme est une fonction dont les

paramètres *ne sont pas évalués* avant d'être passés au corps de la fonction. Il en résulte ainsi que le code étant en position d'argument est considéré par la macro comme étant des données Scheme (listes, symboles, nombres, etc...). Puisque la macro Scheme est une fonction, elle possède toute la puissance du langage à sa disposition. La S expression retournée par la macro sera le résultat de l'expansion de celle-ci et sera donc évalué durant l'exécution du programme à l'endroit où l'appel de la macro se retrouvait. Un exemple simple de macro tirant profit du fait que l'argument est passé par nom est illustré dans la figure 3.9. Cette macro très simple utilise le passage par nom pour permettre d'incrémenter la valeur d'une variable, ce qui ne serait pas possible autrement en Scheme puisque le passage d'argument se fait normalement par valeur.

```
(define-macro (inc! var) `(set! ,var (+ ,var 1)))
(let ((x 1))
    (inc! x)
    x)
```

2

FIGURE 3.9 – Exemple simple de macro utilisant le passage par nom à profit.

Un autre exemple de macro utilisant le passage par nom est donné dans la figure 3.10. Cette macro permet d'augmenter le langage Scheme d'une nouvelle forme spéciale qui correspond à une forme simple d'itération comme l'on retrouve souvent dans les langages impératifs. L'expansion de l'appel à cette macro illustré dans la figure 3.10 est présentée dans la figure 3.11.

```
(define-macro (for var init limit . body)
  `(let loop ((,var ,init))
     (if (< ,var ,limit)
         (begin ,@body
                (loop (+ ,var 1))))))
(let ((v (make-vector 5)))
  (for x 0 5 (vector-set! v x x))
  v)
```

#(0 1 2 3 4)

FIGURE 3.10 – Exemple de macro ajoutant une nouvelle forme spéciale au langage.

```
(let ((v (make-vector 5)))
  (let loop ((x 0))
    (if (< x 5) (begin (vector-set! v x x)
                       (loop (+ x 1)))))
  v)
```

FIGURE 3.11 – Résultat de l'expansion macro de l'appel présenté dans la figure **??**.

Ces macros demeurent très simples et se rapprochent de ce qui peut être exprimé à l'aide de macro C. Toutefois, la macro `for` comporte plusieurs problèmes. L'un d'entre eux est relié à l'introduction de la variable `loop`. L'ajout de cette variable peut causer le problème connu sous le nom de *Capture de noms*. En effet, la macro n'aura pas le comportement escompté si l'un de ses paramètres possède une référence à une variable déjà existante nommée `loop`. Le problème de capture de variables était tout aussi présent dans les macro de C et constituait pour ce type de macro un problème insolvable. Par contre, il est possible d'éviter ce problème en Scheme en générant un nom de variable pour `loop` qui est assurément unique. L'implantation Gambit-C offre la fonction `gensym` pour ces besoins.

Le problème de capture de nom est associé au phénomène appelé hygiène des macros. Ce problème peut être résolu manuellement comme suggéré, mais il existe aussi d'autres expanseurs de macros assurant l'hygiène des macros, dont entre autres, la forme spéciale `syntax-rules` [28]. La réécriture de la macro `for` en utilisant un système de macro hygiénique est donnée dans la figure 3.12. Cette figure donne aussi un exemple d'utilisation de la macro mettant à l'épreuve l'hygiène de celle-ci. On constate que malgré le fait qu'aucun effort n'a dû être mis en place pour éviter les collisions de noms pour la variable `loop` introduite, la macro réalise le comportement attendu dans des conditions potentiellement problématiques.

```
(define-syntax for
    (syntax-rules ()
      ((for var init limit body ...)
       (let loop ((var init))
         (if (< var limit)
             (begin body ...
                    (loop (+ var 1))))))))
(let ((v (make-vector 5)))
    (for loop 0 5 (vector-set! v loop loop))
    v)
```

#(0 1 2 3 4)

FIGURE 3.12 – Macro `for` hygiénique

Bien que fait que l'expansion de macro faite avec `define-macro` puisse impliquer des collisions de variables, elle permet une plus grande liberté d'écriture. En effet, parfois la collision de nom peut, être intentionnelle. Aussi, le problème est facilement résolu avec l'utilisation de symboles uniques.

Un exemple plus convaincant d'une macro Scheme utilisant la puissance de calcul du langage est illustré dans la figure 3.13. Cette macro résulte en la création d'un masque binaire ayant les bits des puissances de deux spécifiées en argument. Ainsi, l'expansion de l'appel `(mask 1 3 8)` résulte à être directement `266`. Il est intéressant de noter qu'un appel à cette macro tel que `(mask 1 (+ 1 2))`, est une erreur. En effet, puisque la macro s'attend à avoir un nombre en paramètre, mais elle reçoit plutôt de ce cas ci la liste `(+ 1 2)`, puisque les arguments de macros ne sont pas évalués.

```
(define-macro (mask . bits)
  (define (curry2 f x) (lambda (y) (f x y)))
  (apply + (map (curry2 expt 2) bits)))
(number->string (mask 1 3 8) 2)
```

"100001010"

FIGURE 3.13 – Exemple plus avancé de macro Scheme.

Le calcul de la somme des puissance de deux se fait donc durant la compilation du programme, donnant accès à des optimisations intéressantes, sans toutefois perdre au niveau de l'élégance du code. Ce genre de macros mènent directement vers la construction de langages spécifiques au domaine (*Domain Specific Languages* ou simplement *DSL*). Ces derniers constituent l'ajout de nouvelles constructions à Scheme permettant d'exprimer facilement des idées relié à un concept précis de manière élégante, sans perte de performances. La macro `mask` pourrait constituer le pilier de base d'un *DSL* effectuant la création et la manipulation de masques

binaires. La figure 3.14 illustre une deuxième macro effectuant durant l'expansion macro la combinaison de masques binaires. On obtient un petit langage pour la déclaration efficace de tels masques.

```
(define-macro (mask-or . masks)
  (apply bitwise-ior (map eval masks)))
(mask-or (mask 1 3) (mask 8))
```

266

FIGURE 3.14 – Macro venant augmenter le petit *DSL* portant sur la manipulation de masques binaires

On peut maintenant imaginer de tels langages spécifiques au domaine pour n'importe quel problème associé aux jeux vidéo. Par exemple, une macro pour être écrite pour la description détaillée d'un modèle tridimensionnel et une autre pour la combinaison de ces dernières. Ainsi, ces macros pourraient permettre de décrire le modèle d'une manière simple pour les artistes, en utilisant un syntaxe près de d'un langage naturel et, durant l'expansion, transformer ces déclaration et combinaisons, en des structures de données efficaces tant en utilisation mémoire qu'en rapidité d'accès et de manipulation. Un autre exemple pourrait être la création d'un système de programmation orienté objet à l'aide de macros Scheme. La puissance de calcul disponible durant l'expansion pourrait ainsi servir à la création des fonctions auxiliaire d'accès aux membres, à la détermination des membres hérités, etc...

En conclusion, les macros Scheme donnent accès à une très grande puissance

expressive en permettant l'utilisation de la puissance complète du langage durant l'expansion de ces dernières. Cette puissance de calcul et cette simplicité provient du fait que le code source de Scheme et les données du langage Scheme utilisent la même syntaxe, soit les S-expressions. Ainsi, en utilisant ce système de macros, il est possible d'ajouter des nouvelles formes spéciales au langages et ainsi créer des langages spécifiques au domaine (*Domain specific languages*). Ces derniers sont créés au dessus de Scheme dans le but de résoudre un problème précis de manière efficace, tout en offrant une syntaxe facilitant leur utilisation.

3.5 Continuations

La forme d'écriture de code appelée CPS (*Continuation Passing Style*) est un style d'écriture utilisé dans les langages fonctionnels qui permet d'expliciter la suite du calcul d'une fonction explicitement. La figure 3.15 illustre une implantation simple de la fonction factoriel. La suite du calcul effectuée après un appel à la fonction `fact` est implicite dans l'écriture de son appel. Dans l'exemple d'appel présent dans cette figure, on constate que la suite de ce calcul sera l'addition de un au résultat obtenu.

Le corps de la fonction `fact` présentée possède deux types d'appels de fonctions, soient les appels terminaux et non-terminaux. La distinction entre ces deux types d'appels est la suivante : un appel terminal est un appel qui termine l'exécution d'une fonction et un appel non-terminal est tel que le résultat de cet appel nécessitera

```
(define (fact n)
  (if (< n 2)
      1
      (* n (fact (- n 1)))))
(+ (fact 10) 1)
```

3628801

FIGURE 3.15 – Implantation simple de la fonction factoriel

du traitement ultérieur dans la fonction avant que celle-ci termine. Ainsi, seul l'appel à la fonction `*` est terminal dans la fonction `fact`. Le standard du langage Scheme requiert l'implantation de l'optimisation d'appels terminaux. Cette dernière fait en sorte que le résultat d'un appel terminal est passé directement à la continuation de l'appel à la fonction en question. Il en résulte que le résultat d'un appel terminal ne sera pas attendu par la fonction qui le contient. Il est ainsi possible de pouvoir effectuer des itérations à l'aide d'appels terminaux de fonctions.

La transformation de cette fonction en style CPS est donnée dans la figure 3.16. Dans cette nouvelle version, la suite de l'exécution d'un appel est explicite, il s'agit d'un appel à la continuation passée en paramètre, nommée `k`.

Une propriété intéressante de cette transformation de code est reliée au fait que tous les appels de fonctions deviennent des appels terminaux. La transformation CPS est souvent utilisée par les implantations de Scheme car elle facilite l'implantation de l'optimisation d'appels terminaux. Ce style de programmation permet aussi d'utiliser de manière élégante la puissance d'abstraction des langages fonctionnels en permettant de diviser le calcul en petites parties facilement interchangeables et

```
(define (minusk n1 n2 k) (k (- n1 n2)))
(define (timesk n1 n2 k) (k (* n1 n2)))
(define (factk n k)
  (if (< n 2)
      (k 1)
      (minusk n 1 (lambda (r)
                    (factk r (lambda (r2)
                               (timesk n r2 k)))))))
(factk 10 (lambda (x) x))
```

3628800

FIGURE 3.16 – Forme CPS de la fonction factoriel

modulaires.

Le langage Scheme offre la possibilité de réifier automatiquement la continuation d'un calcul donné par l'entremise de la forme spéciale `call-with-current-continuation` souvent nommée aussi `call/cc`. Cette forme spéciale s'attend à recevoir une fermeture en argument, cette dernière n'attendant qu'un seul paramètre : la continuation réifiée du calcul au point d'appel de `call/cc`. La figure 3.17 donne un exemple de la réification d'un calcul similaire à celui présenté dans la figure **??**. La mutation de la variable `k` est utilisée afin de permettre une sauvegarde de la continuation et dans le but de la réutiliser plus tard.

Ainsi, la sauvegarde d'une continuation permet de conserver l'état du calcul et la réutilisation de celui-ci.

```
(define k #f)
(define f (lambda (x y) (* x y)))
(+ 3 (- (call/cc (lambda (cont) (set! k cont) [f 5 6])) 2))
```

31

```
(k 10)
```

11

FIGURE 3.17 – Réification d'une continuation à l'aide `call/cc`

3.5.1 Exemples d'utilisations

L'utilisation de continuations ajoute beaucoup de puissance au niveau de la programmation en Scheme, si elle est habilement exécutée. Cette section présente quelques utilisations intéressantes permettant d'entrevoir les possibilités qu'offre cet aspect de Scheme.

3.5.1.1 Échappement au flux de contrôle

Un exemple simple d'utilisation de continuation est pour l'échappement au contrôle d'une fonction. Bien sûr, un programmeur a toujours le contrôle du programme qu'il écrit, mais il peut y avoir des situation qui exigent l'utilisation de fonctions externes dont on ne peut pas modifier le contenu. De telles situations peuvent mener à des inefficacités algorithmiques. La figure 3.18 illustre une utilisation de la fonction d'ordre supérieur `foldl` présentée dans la figure 3.7 afin d'implanter un algorithme permettant de déterminer s'il existe au moins une valeur supérieure à 10 dans une liste. En utilisant la fonctionnalité de traces qu'offre

Gambit-C, on constate que malgré le fait que l'utilisation d'une fonction d'ordre supérieur permet d'implanter facilement cet algorithme, il en résulte en du code inefficace, puisque l'appel à `foldl` va analyser tous les éléments de la lite, même après avoir trouvé une valeur supérieure à 10.

```
(trace foldl)
(define (exists-greater-than val lst)
    (foldl (lambda (false x) (if (> x val) x false))
           #f
           lst))
(exists-greater-than 100 (list 1 102 3 2 7 12))
```

```
|>(foldl proc #f '(1 102 3 2 7 12))
|>(foldl proc #f '(102 3 2 7 12))
|>(foldl proc 102 '(3 2 7 12))
|>(foldl proc 102 '(2 7 12))
|>(foldl proc 102 '(7 12))
|>(foldl proc 102 '(12))
|>(foldl proc 102 '())
|102
102
```

FIGURE 3.18 – Utilisation non efficace de `foldl`.

On doit donc ré-écrire le méta comportement offert par `foldl` afin de pouvoir arrêter la recherche aussitôt qu'une valeur supérieure à 10 est trouvée. Cela vient à l'encontre du principe d'abstraction mène à une mauvaise maintenabilité du code.

Les continuations nous offres par contre la possibilité d'utiliser la méta fonction `foldl`, sans perte de performances. La figure 3.19 indique comment s'y prendre. En réifiant la continuation au début du calcul de la fonction `exists-greater-than`, cette continuation capture le reste du calcul à faire après cet appel de fonction et

donc, lorsque cette continuation, liée à la variable `return`, est appelée, il en résulte que le flot de contrôle branche vers la suite du calcul en prenant comme valeur de retour la valeur trouvée.

```
(trace foldl)
(define (exists-greater-than val lst)
    (call/cc
     (lambda (return)
       (foldl (lambda (retval x) (if (> x val)
                                     (return x)
                                     retval))
              #f
              lst))))
(exists-greater-than 100 (list 1 102 3 2 7 12))
```

|>(foldl proc #f '(1 102 3 2 7 12))
|>(foldl proc #f '(102 3 2 7 12))
102

FIGURE 3.19 – Utilisation d'une continuation pour échapper au flot de contrôle.

3.5.1.2 Recherche par retour arrière

Un autre exemple d'utilisation intéressant de continuations est dans la création d'un système de recherche par retour arrière (*backtracking*). La figure 3.20 illustre un exemple simple de recherche d'un triplet de nombres entiers tel que $x^2 = y^2 + z^2$.

Dans cet exemple simple, la fonction `fail` représente une liste chaînée de retour arrières permettant de revenir à une étape de décision précédente et de poursuivre le calcul en prenant une nouvelle décision à cet endroit. Cet exemple illustre bien l'élégance qui peut résulter de l'utilisation judicieuse de la forme spéciale `call/cc`.

```
(define fail (lambda () (error "can't backtrack")))

(define (in-range a b) (call/cc (lambda (cont) (enumerate a b cont))))

(define (enumerate a b cont)
  (if (> a b)
      (fail)
      (let ((save fail))
         (set! fail (lambda () (set! fail save)
                               (enumerate (+ a 1) b cont)))
         (cont a))))

(let ((x  (in-range 1 9))
      (y  (in-range 1 9))
      (z  (in-range 1 9)))
  (if (= (* x x) (+ (* y y) (* z z)))
      (list x y z)
      (fail)))
```

(5 3 4)

FIGURE 3.20 – Exemple de recherche par retour arrière

3.5.2 Conclusion

Ainsi, la réification de continuation implicites ou encore l'utilisation d'un style CPS utilisant des continuation explicites apporte beaucoup de puissance au langage Scheme. L'utilisation de continuation permet la sauvegarde de l'état d'un calcul en cours dans une fermeture et permet sa réutilisation ultérieure. On peut ainsi créer sans trop d'efforts des systèmes d'exceptions, de retour arrière ou même de coroutines, systèmes qui seraient tous pertinent à utiliser dans le cadre de la programmation de jeux vidéo.

3.6 Gestion mémoire automatique

L'écriture de code effectuant la gestion de la mémoire d'un programme a toujours été une partie délicate du développement d'un logiciel. Non seulement, il est très facile de faire des erreurs, mais ces erreurs sont très difficiles à détecter et à être trouvées. Les problèmes reliés à une mauvaise gestion de la mémoire sont classifiés en deux catégories : des fuites de mémoire ou des pointeurs fous. Les fuites de mémoire sont causées par la rétention de mémoire par des objets qui ne sont plus utilisés par le programme et les pointeurs fous sont des pointeurs toujours accessibles par le programme, mais dont l'espace mémoire correspondant a été récupéré par le système d'exploitation. Ces erreurs sont souvent les conséquence d'un problème plus difficile encore, le problème de modularité. En effet, la durée de vie d'un objet en mémoire est normalement facile à déterminer à petite échelle. Par contre, lorsque le code est écrit de manière modulaire, il est difficile de prévoir toutes les utilisations des objets et souvent la responsabilité de la désallocation des objets est difficile à déterminer.

Ces erreurs sont très problématiques. C'est sans doute pour cette raison que déjà très tôt, cette gestion a été automatisée par des systèmes de gestions automatiques de mémoire. Les premiers systèmes de gestion de mémoires automatiques, appellés aussi ramasses miettes, apparurent avec le langage *Lisp* . Le principe de base est simple, il s'agit de parcourir la mémoire utilisée par le programme en partant des *racines* de celui-ci. Les racines sont tous les points d'accès aux données

accessibles, comme par exemple les variables globales ou locales d'une fonction en cours d'exécution. Lorsque la mémoire est parcourue, on garde trace des zones accessibles. Après avoir parcouru toute la mémoire accessible, il ne suffit que de récupérer automatiquement les zones qui ne sont plus accessibles par le programme.

Bien sûr, la gestion de mémoire automatique se fait avec un certain *coût en temps de calcul*. La répartition de ce temps dépend grandement de la technique de gestion utilisée. Aussi, les algorithmes de gestion de mémoire se doivent d'être *conservateurs* dans la désallocation d'objets puisque la détermination de la vivacité d'un objet est, dans la généralité du problème, indécidable. Ainsi, il peut y avoir des cas où le système de gestion pense qu'un pointeur est toujours accessible, alors qu'en réalité, il ne sera plus jamais utilisé par le programme créant ainsi des fuites de mémoires. Ces fuites sont toutefois beaucoup plus difficile à créer que celles obtenues lorsqu'une gestion manuelle de mémoire est utilisée.

3.6.1 Survol des techniques

Plusieurs techniques de gestion de mémoire automatiques ont été développées au cours des années. Cette section effectue un bref survol de ces différentes techniques. Plusieurs articles effectuant une revue des techniques sont disponibles pour fournir plus de détails sur ces techniques sur le sujet [29] [30] [31].

3.6.1.1 *Mark And Sweep*

Une des techniques la plus simple de gestion de mémoire automatique est le *Mark And Sweep* [15]. Cette technique consiste à traverser la mémoire à partir des racines et à marquer chacun des espaces mémoire rencontrés lors du parcours. Ensuite, tout les espaces mémoires alloués sont parcourus et tout ceux qui ne sont pas marqués sont récupérés. Cette technique est très simple, mais telle quelle, peut mener à des problèmes de fragmentations de la mémoire. De plus, elle requiert de traverser toute la mémoire utilisée par deux fois, impliquant donc un certaine pause dans l'exécution du programme.

3.6.1.2 *Stop And Copy*

Une autre technique classique se nomme *Stop And Copy* [32]. De manière similaire au *Mark And Sweep*, cette technique parcoure les zones mémoires utilisées à partir des racines. Par contre, le *Stop And Copy* doit séparer la mémoire disponible en deux zones distinctes, l'une dans laquelle les allocations sont effectuées et l'autre demeure réservée. Lorsque le ramasse miettes atteint un objet en mémoire, il le copie vers la nouvelle zone réservée et laisse une indication de sa nouvelle position dans son ancien emplacement. Toutes les zones mémoires sont ainsi copiées et compactées et donc, les allocations subséquentes sont par la suite dans la nouvelle zone mémoire, résultant en une désallocation implicite des espaces inutilisés dans la zone précédente. Cette technique est légèrement plus efficace que le *Mark And*

Sweep traditionnel, mais nécessite le double d'espace mémoire pour faire la gestion de la mémoire. Cette technique est utilisée dans l'implantation Scheme Gambit-C.

3.6.1.3 Gestionnaires générationnels

Une amélioration notable de la technique du *Mark And Sweep* a donné naissance aux ramasses miettes générationnels [33]. Sans entrer dans les détails, cette technique divise la mémoire en plusieurs générations. Les objets récemment alloués sont conservés dans la pouponnière puis, lorsque ceux-ci sont conservés plus longtemps, sont déplacés vers une génération plus ancienne, et ainsi de suite. Seul la pouponnière est parcourue lors de chaque récupération et lorsque celle-ci devient remplie, elle déclenche une récupération dans la génération au dessus d'elle. Ainsi, cet algorithme est beaucoup plus mieux adapté aux cas d'utilisations les plus fréquents, car les objets récemment alloués ont souvent une courte durée de vie moyenne, contrairement aux objets déjà retenus qui eux possèdent une meilleur chance d'être retenus.

Tout comme les deux techniques précédentes, les ramasses miettes générationnels nécessitent l'arrêt du programme durant l'opération récupération, qui peu, dans le pire cas, s'effectuer sur toute la mémoire. Ainsi, les gestionnaires de mémoires générationnels améliorent les performances du temps de récupération moyen, en augmentant le temps de récupération de toute la mémoire (pire cas).

3.6.1.4 Gestionnaires incrémentaux

Afin de remédier au problème d'arrêt complet de l'exécution du programme, d'un nouveaux types de ramasses miettes a été développé : des gestionnaires de mémoires incrémentaux ou temps réels [34]. Ceux-ci peuvent utiliser un algorithme générique de marquage à trois couleurs [35] de la mémoire afin d'effectuer partiellement le travail de récupération et de poursuivre l'application dans le but de continuer ultérieurement. Cet algorithme fonctionne en subdivisant la mémoire en trois ensemble : l'ensemble blanc, l'ensemble gris et l'ensemble noir. Initialement toutes les zones sont blanches, à l'exception des racines qui sont marquées comme grises. Ensuite, pour chaque élément de l'ensemble gris, l'algorithme marque chacune des zones accessibles par ceux-ci comme gris et marque finalement l'élément gris comme noir. À la fin, les zones mémoire toujours blanches sont inaccessibles et donc récupérées. Le travail de marquage peut être fait en séparant le travail en ne traversant qu'un nombre fini de noeuds. Il en résulte donc que les pauses dues à la récupération peuvent être bornées supérieurement.

Afin de pouvoir entremêler le travail du récupérateur de mémoire et du programme exécuté (nommé le mutateur), certaines précautions doivent être prises afin d'éviter que le mutateur modifie des pointeurs en mémoire tel qu'il serait possible d'accéder à des espaces marqués comme étant non-accessibles (et donc qui seront ultérieurement récupérés). Deux approches sont alors envisageables, des barrières de lectures ou d'écriture sur les zones mémoires. Les barrières de

lectures assurent que seuls de pointeurs noirs sont retournés. Ces dernières sont généralement très coûteuse car les accès mémoire constituent environ 15% des instructions /citeZorn90barriermethods. Par contre, celles-ci doivent être utilisés pour des récupérateurs incrémentaux copiant, tel un *Stop And Copy* incrémental. Les barrières en écritures possèdent plusieurs variantes qui effectuent des compromis entre l'efficacité algorithmique et l'efficacité de récupération.

3.6.2 Conclusion

Malgré que les techniques de récupération automatique de mémoire impliquent des coût supplémentaires en utilisation du processeur et de la mémoire, le gain en productivité pour le développement est tellement supérieur qu'ils sont devenus adoptés par presque tous les nouveaux langages de programmations.

Le fait que l'exécution du programme puisse être arrêté durant le moment de la récupération est potentiellement problématique pour l'écriture de jeux vidéo, car ceux-ci se doivent d'avoir des bonnes performances et ce, de manière continue.

Les algorithmes de gestion de mémoire incrémentaux améliorent le pire cas du temps de récupération en augmentant le coût moyen. Ainsi, les coûts apportés à l'exécution du programme doivent être considérés. Ainsi, l'utilisation de récupérateurs générationnels semblent un bon choix pour la programmation de jeux vidéo car ils offrent de bonnes performances pour les utilisations normales de la mémoire, comme c'est généralement le cas pour des jeux vidéo.

3.7 Dynamisme du langage

Le langage Scheme est un langage à typage dynamique, faisant ainsi contraste aux langages fonctionnels de la famille *ML*. Un typage dynamique se distingue par des vérifications de types effectuées durant l'exécution d'un programme. Un typage statique, quant à lui, effectuera les vérifications de types durant la compilation. Ainsi, une importante différence entre les deux approches est le moment où les erreurs de programmation apparaissent. D'autre part, dû à la nature beaucoup plus stricte du typage statique, la récupération d'erreurs est beaucoup plus difficile, car elle dépend uniquement d'informations connues statiquement.

Un typage statique fournira des erreurs de types beaucoup plus tôt dans le processus de développement et permettent de pouvoir exécuter un code libre de toutes vérifications de types. Une particularité des systèmes typés statiquement consiste à avoir des structures de code plus rigides, demandant possiblement beaucoup d'efforts pour être modifiées.

Pour un typage dynamique, les erreurs ne surviendront que durant l'exécution du programme et, pourraient ne jamais se produire, ou difficilement se produire. De plus, un typage dynamique implique des pertes de performances puisque les vérifications de types sont effectuées pendant l'exécution du programme. Par contre, des erreurs apparaissant durant l'exécution peuvent être beaucoup plus faciles à corriger si l'implantation est muni de bon système de déboggage. Aussi, un typage dynamique rend plus accessible la possibilité d'interpréter le langage en plus de

pouvoir le compiler. L'interprétation permet un développement rapide et facilite le prototypage grâce à la possibilité de faire de la programmation en direct (*live coding*).

Les deux systèmes de typages peuvent être comparés à l'aide de métaphores simples. Des programmes écrits dans des langages typés statiquement peuvent être considérés comme des murs de briques. Les types agissent comme du ciment entre les fonctions et donc la modification de la structure nécessite des efforts pour la refaire. Par contre, lorsque ces bases sont stables et solides, le typage statique assure un adhération parfaite entre les blocs. En opposition, le typage dynamique peut être vu comme un système de plantes vivantes, où les feuilles sont des fonctions ou des modules. Le système permet une grande flexibilité d'évolution en s'adaptant plus facilement aux changements et permet de pouvoir être facilement modifié en cours de route pour donner le résultat escompté. Un entretient régulier doit être par contre fait par la suite pour éviter une évolution qui pourrait aller dans une mauvaise direction.

3.7.1 Interprétation et débuggage efficace

Un langage typé dynamiquement à la possibilité d'être interprété puisque les types ne doivent pas nécessairement être correct avant l'exécution. On peut ainsi évaluer les instruction les unes à la suite des autres dans un environnement global d'interprétation (*toplevel*). Cette particularité infuse beaucoup de puissance au

langage. En effet, cela permet de pouvoir tester facilement les modules écrits de manière indépendante et interactive. Les cas extrêmes peuvent ainsi facilement être vérifiés et donc il est possible d'avoir une bonne assurance sur le fonctionnement correct des algorithmes utilisés.

En cas d'erreur, le système est conçu afin de permettre de détecter les erreurs et de pouvoir les diagnostiquer. Puisque l'environnement est dynamique, une grande puissance d'introspection et d'analyse est disponible. Le système Gambit-C permet de pouvoir non seulement inspecter la pile de continuations courantes lors d'une exception, mais permet l'introspection d'environnement de fermetures et la possibilité de rétablir ces environnement et de pouvoir exécuter du code arbitraire dans ceux-ci dans le but de pouvoir trouver la source de l'erreur.

[TODO: *Exemple de debuggage ? ?*]

De plus, le dynamisme du langage peut pousser la puissance de récupération d'erreurs encore plus loin en combinant le système de débuggage de Gambit-C avec un protocole établi sur *TCP-IP* permettant de pouvoir faire du dégoggage à distance. On peut ainsi analyser une erreur s'étant produite sur un processus distant, qui pourrait potentiellement être exécuté sur un système embarqué ou un appareil ne disposant pas d'interface permettant la récupération d'erreurs. Par exemple, ce système a été utilisé afin de pouvoir débugger des processus Scheme exécutés sur un téléphone mobile.

Il en résulte donc que malgré un faible coût en performance lors de l'exécution

d'un programme, un système utilisant le typage dynamique apporte beaucoup de puissance aux programmeurs et leurs fourni des outils efficaces pour le développement de logiciels.

[TODO: *ajouter ref pour tcpip ?*]

3.8 Extension utiles

Cette section présente un recueil des extension du langage Scheme qui sont utilisées dans les chapitres suivant de ce mémoire. Ces dernière ne constituent pas nécessairement un atout précis quand à la programmation de jeu vidéo, mais apporteront plutôt une meilleure compréhension des exemples de codes fournis dans ce mémoire.

3.8.1 Fermetures

Le système Gambit-C permet la définition de paramètres avec l'utilisation de mots clés. La figure 3.21 illustre un exemple de définition d'une telle fonction avec quelques appels à celle-ci. On constate qu'il est alors possible de définir des valeurs optionnelles qui peuvent posséder une valeur par défaut.

```
(define (f x y #!key z (sum (+ x y)))
  (list x y z sum))
(f 1 2)                      => (1 2 #f 3)
(f 1 2 z: 'allo)             => (1 2 allo 3)
(f 1 2 z: 'allo sum: 'salut) => (1 2 allo salut)
```

FIGURE 3.21 – Utilisation du passage de paramètre par mots clés

La syntaxe des fonctions a aussi été étendue de manière à permettre la spécification d'un nombre de paramètre arbitraire. Pour ce faire, le mot clé `#!rest` ou encore le point (`.`) est utilisé pour délimiter les paramètres normaux du paramètre reste, qui contiendra une liste de tous les paramètres supplémentaires utilisés. La figure 3.22 illustre un exemple d'utilisation du paramètre reste.

```
(define (g x y . z) (list x y z))
(g 1 2)       => (1 2 ())
(g 1 2 3 4 5) => (1 2 (3 4 5))
```

FIGURE 3.22 – Exemple d'utilisation du paramètre reste

Lorsqu'une fonction prends plusieurs arguments, il est possible d'utiliser la procédure `apply` afin de pouvoir utiliser cette fonction avec *une liste* d'arguments. Cette fonction attend une fonction comme premier argument et un nombre arbitraire d'arguments par la suite. Le dernier argument passé doit être une liste. Le contenu de cette liste sera inséré dans un appel à la fonction spécifiée. La figure 3.23 illustre quelques utilisations de la procédure `apply`.

```
(+ 1 2 3 4 5)                          => 15
(apply + 1 2 '(3 4 5))                 => 15
(apply + (map (flip + 1) '(1 2 3 4 5))) => 20
```

FIGURE 3.23 – Exemple d'utilisation de la fonction `apply`

3.8.2 Tables de hachages

Le système Gambit-C supporte aussi l'utilisation de tables de hachages. Ces dernières peuvent être créées par la fonction `make-table` et des valeurs peuvent y être insérées ou inspectées avec les fonctions `(table-set! <table> <key> <value>)` et `(table-ref <table> <key> [<default-value>])`. Il est aussi possible de spécialiser ces tables afin d'être plus efficaces pour des clés étant des symboles ou des chaînes de caractères.

3.8.3 Tests d'égalité et opérations conditionnelles

L'égalité en Scheme entre deux objets appartenant aux types de base du langage peut toujours être vérifiée par la fonction `equal?`. Cette dernière supporte ainsi la vérification d'égalité sur plusieurs types d'objets et se trouve ainsi peu efficace. La fonction `eq?` permet la vérification d'égalité sur des pointeurs et puisque, en Scheme, un symbole ou un mot clé ne possède qu'un instance unique, ce prédicat peut être également utilisé pour distinguer ces derniers types.

L'opération conditionnelle de base en Scheme est effectué par la forme spéciale `(if <cond> <true> [<false>])`. Plusieurs formes spéciales dérivent de cette dernière afin de pouvoir exprimer plus facilement certains cas typiques. Par exemple, lorsque plusieurs conditions sont imbriquées les unes dans les autres, la forme spéciale `(cond (<pred> <body>) ... [(else <body>)])` peut être utilisée. Cette dernière permet aussi de pouvoir récupérer la valeur retournée par un prédicat s'évaluant à vrai

en ajoutant => tel qu'illustré dans la figure 3.24.

```
(let ((t (make-table test: eq?)))
  (table-set! t (list 'a) 'a)
  (table-set! t 'b 'bonjour)
  (cond ((table-ref t (list 'a) #f) 'allo)
        ((table-ref t 'b #f) => (lambda (msg) msg))
        (else 'bonsoir)))
```

=> *bonjour*

FIGURE 3.24 – Exemple d'utilisation d'une table de hachage spécialisée pour des clés symboliques et d'utilisation de la forme spéciale `cond`

3.8.4 Déclaration et utilisation d'objets

Le système Gambit-C apporte la définitions d'objets implantant des fonctionnalités minimales d'un système de programmation orienté objet. La définition de telles structures se fait en utilisant la forme spéciale

```
(define-type <type-id> [<member-id>] ...  [extender:  <extender-name>])
```

Celle-ci effectue la création de fonctions d'accès et de modification de membres, de création d'instances et de vérification de type. L'utilisation du mot clé `extender:` permet aussi la création d'une nouvelle forme spéciale (au nom spécifié) qui permet la définition de sous-types héritant des membres de la classe en question, implantant ainsi un héritage simple. La figure 3.25 illustre l'utilisation de cette forme spéciale.

```
(define-type point x y extender: define-type-of-point)
(define-type-of-point circle r)
(define (add p1 p2) (make-point (+ (point-x p1) (point-x p2))
                                (+ (point-y p1) (point-y p2))))
(add (make-point 1 2) (make-circle 3 4 5)) => #< point #4 x: 1 y: 1>
(if (circle? (make-point 1 2)) 'ok 'no)    => no
```

FIGURE 3.25 – Exemle de création d'objets dans Gambit-C

3.8.5 Affichage de résultats

Plusieurs fonctions sont disponible en Scheme afin d'effectuer l'affichage du résultat de l'évaluation d'une expression. Le standard Scheme fourni entre autre les fonctions `write` et `display` qui permettent un affichage bas niveau de données Scheme. Le système Gambit-C offre aussi les fonctions `pretty-print` (ou `pp`) afin de faire un affichage enjolivé d'une sortie de `write`. La fonction `println` afin de permettre l'affichage d'expressions composées. La figure 3.26 illustre des exemples d'utilisations de ces fonctions.

```
(write (list 1 'a "a")) => (1 a "a") (display (list 1
    'a "a")) => (1 a a) (write (list 1 'a "a")) =>
    (1 a "a")\ n (println "Bonjour " (+ 1
    2) (map (flip + 1) '(1 2 3))) => Bonjour 3234
```

FIGURE 3.26 – Exemples d'utilisation de fonctions d'affichage

3.8.6 Interfaces aux fonctions étrangères

Une interfaces aux fonctions étrangères (*Foreign Function Interface* ou FFI) est disponible dans le système Gambit-C. D'une part, cette dernière permet l'importa-

tion et l'utilisation de fonctions C ou C++ présentent dans de librairies. La figure 3.27 illustre une interface minimaliste à la librairie C OpenGL et une utilisation de cette interface. On constate donc qu'une fois importées, les fonctions de cette dernière peuvent être utilisées comme n'importe qu'elle fonction Scheme.

```
(c-declare "#include <gl.h>")

(c-define-type GLenum unsigned-int)

(define GL_QUADS ((c-lambda () int "___result = GL_QUADS;")))

(define glVertex2d (c-lambda ( GLdouble GLdouble ) void "glVertex2d"))
(define glBegin (c-lambda ( GLenum ) void "glBegin"))
(define glEnd (c-lambda () void "glEnd"))

(glBegin GL_QUADS)
(glVertex2d 0 0) (glVertex2d 0 1)
(glVertex2d 1 0) (glVertex2d 1 1)
(glEnd)
```

FIGURE 3.27 – Exemple d'utilisation de l'interface à des fonctions étrangère de Gambit-C

D'autre part, il est aussi possible d'exporter des fonctions écrites en Scheme de manière à ce qu'elles soient utilisées à partir de code C ou C++.

3.9 Conclusion

Cet aperçu des particularités du langage fonctionnel Scheme démontrent clairement que le langage répond grandement aux besoins établis par les jeux vidéo. Notamment, la puissance d'abstraction provenant de l'aspect de programmation fonc-

tionnel du langage permettrait d'obtenir une bonne modularisation du code source. Aussi son système de macro ajoute la possibilité de créer de langages spécifiques aux domaines répondant à des problèmes précis pour l'écriture de jeux vidéo. Le dynamisme du langage apporte des outils de développement puissant, comme un interprète et un déboggeur à distance, mais aussi permet de faciliter le développement itératif avec un approche par prototypage. La présence d'un gestionnaire automatique de mémoire évite la présence potentielle de beaucoup d'erreurs reliées à la gestion de mémoire, mais pourrait causer des problèmes de ralentissement lors de l'exécution de jeux. Finalement, la possibilité de pouvoir réifier les continuations d'un programme en cours d'exécution offre beaucoup de puissance expressive au programmeur.

Le langage Scheme étant très simple à la base, il est essentiel de développer des extensions de ce langage afin de pouvoir développer sur une base solide créée pour les besoins spécifiques du types d'application développé. L'expressivité du langage rend cette tâche facile. Ainsi, deux extensions au langages Scheme ont été développées dans le but de pouvoir répondre spécifiquement aux besoins des jeux vidéo.

CHAPITRE 4

PROGRAMMATION ORIENTÉE OBJET

Scheme est un langage offrant aux programmeurs beaucoup de ressources à l'état brut et leur permet de se bâtir des outils personnalisés pour mieux répondre leurs à besoins.

Un paradigme qui n'est pas supporté directement par le langage Scheme est celui de la programmation orientée objet. Ce paradigme possède des structures de données hiérarchiques, les objets, favorisant la modularité et l'encapsulation de celles-ci. Cette modularité provient des abstractions résultantes de la hiérarchie des méta-objets appelés classes et l'encapsulation permet de pouvoir cacher les données qui devraient demeurer internes au module créé. Le sous-typage résultant de la création de classes permet entre autre l'utilisation générique de fonctions (souvent appelées méthodes ou fonctions génériques).

Ces objets sont centrés sur la notion d'état qui évolue au fil de l'exécution du programme. Malgré le fait que la notion d'état d'un programme ne se marie pas bien avec la programmation fonctionnelle, l'utilisation d'objets pour représenter les entités dans un jeu est très naturelle et appropriée. En effet, ces entités possèdent des caractéristiques propres à elles et qui évoluent dans le temps. Par exemple, on peut imaginer une balle qui possède une vitesse de déplacement et une position.

De plus, l'utilisation de méthodes permet, d'une part, de pouvoir facilement

réutiliser des comportements communs à des objets d'une même hiérarchie de classe et, d'autre part, à pouvoir facilement choisir un comportement approprié pour un objet donné, via le *dispatch* de méthode qui sélectionne automatiquement la méthode la plus appropriée disponible pour un objet donné.

Le SRFI-9 [36] (*Scheme Request For Implementation*) suggère un système très rudimentaire d'objets qui fut implémenté et étendu par le système Gambit-C [7]. Ce système permet la construction de manière générique d'objets typés avec héritage simple. L'exemple de la figure 3.25 illustre un exemple d'utilisation de ce système. L'accès aux membres des objets issus de la même hiérarchie est ainsi possible et donne accès à une polymorphisme minimal.

Ainsi, le système Gambit-C nous permet d'utiliser des concepts de la programmation orientée objet de manière rudimentaire. Cette approche ne fournie aucune possibilité d'attacher des méthodes aux objets qui sont au coeur du polymorphisme.

L'approche plus traditionnelle quant à la déclaration de méthodes de classes est celle utilisée par le langage de programmation Java [37]. Cette approche lie les méthodes à une et une classe et ses descendantes. Les appels de ces méthodes possèdent ainsi un argument caché (souvent nommé *self* ou *this*) qui se trouve à être une instance de cette classe. Le *dispatch* dynamique s'effectue *uniquement sur le type de ce premier argument*. C'est l'approche appelée *dispatch* simple (*Single Dispatch*). Cette approche est bien adaptée pour une interaction centrée sur un seul objet à la fois, mais est très limitée pour des interactions *génériques* sur de

multiple objets différents. L'exemple de la figure 4.1 illustre la manière d'utiliser des méthodes traditionnelles pour effectuer du *dispatch* multiple. On constate qu'une discrimination se fait implicitement sur l'objet instance de la classe associée à la méthode, mais que le *dispatch* sur d'autres objets doit être fait explicitement. L'exemple illustré demeure encore très simple puisqu'il s'agit de *dispatch* double sur deux types, mais pourrait devenir très complexe s'il s'agissait de faire une discrimination sur plus que deux objets.

Le langage *Common Lisp* [16] possède un système orienté objet, le Common Lisp Object System [38] (CLOS), qui diverge beaucoup de l'approche traditionnelle. Entre autre, ce système offre une grande flexibilité de comportement et une riche introspection via un protocole de méta-objets [39] (*Meta Object Protocol*). Par contre, l'aspect le plus intéressant de ce système est l'utilisation de fonctions génériques, éliminant l'utilisation de méthodes traditionnelles. Une fonction générique est un ensemble de fonctions possédant le même nom et qui possèdent des spécifications optionnelles de types pour ses arguments. Lorsqu'un appel de fonction est fait sur une fonction générique, le système décide alors de l'instance de la fonction générique la plus appropriée à utiliser en fonction des arguments actuels passés dans l'appel, du type de ces arguments et de leur nombre. On obtient donc un système très dynamique effectuant du *dispatch multiple*. La figure 4.2 contient la définition d'une fonction générique effectuant automatiquement le choix de la bonne instance en se basant dynamiquement sur les types des arguments passés.

```
class Toto {
  public static void main(String[] args){
    Titi t = new Titi(1); Blub b = new Blub(2);
    System.out.println("t.add(t) = " + t.add(t));
    System.out.println("t.add(b) = " + t.add(b));
  }
}
class Blub {
  int val;
  Blub(int x){ val = x; }
}
class Titi {
  int x;
  Titi (int x){ this.x = x; }
  int add(Object other){
    int y = -1;
    if (other instanceof Titi) { y = ((Titi)other).x; }
    else if (other instanceof Blub) { y = ((Blub)other).val; }
    return x+y;
  }
}
```

t.add(t) = 2
t.add(b) = 3

FIGURE 4.1 – Exemple d'utilisation de méthodes à la manière traditionnelle (en Java)

Il est ainsi possible d'écrire de manière concise des fonctions génériques effectuant un *dispatch* sur plusieurs arguments à la fois. On peut aussi constater que les fonctions génériques ne sont pas définies dans l'espace de nom d'une classe en particulier. Ceci résulte en un meilleur découplage entre les classes et les opérateurs interagissant avec leurs instances.

La programmation orientée objet apporte ainsi beaucoup d'abstractions intéressantes.

```
(defclass Titi () ((x :accessor x :initarg :x)))
(defclass Blub () ((val :accessor val :initarg :val)))
(defgeneric add (p1 p2))
(defmethod add ((p1 Titi) (p2 Titi)) (+ (x p1) (x p2)))
(defmethod add ((p1 Titi) (p2 Blub)) (+ (x p1) (val p2)))
(let ((p1 (make-instance 'Titi :x 1))
      (p2 (make-instance 'Blub :val 2)))

  (print (add p1 p1))  => 2
  (print (add p1 p2))) => 3
```

FIGURE 4.2 – Exemple de *dispatch multiple* écrit en CLOS.

D'une part, les entités présentes dans un jeu vidéo sont aisément représentés par l'utilisation d'objets. En effet, la métaphore des objets possédant dans caractéristiques d'états se confond parfaitement à la perception d'entités de jeux qui possèdent elles aussi des états variant en fonction de l'évolution du jeu. D'autre part, une programmation orientée objet permettrait de facilement abstraire les comportements communs à plusieurs types d'entités présentes dans un jeu vidéo tout en gérant de manière générique leurs états. L'utilisation de fonctions génériques permettrait ainsi de pouvoir facilement modulariser ces comportements. Il serait donc très intéressant de pouvoir utiliser la programmation orientée objet afin de pouvoir écrire des jeux vidéo en Scheme de manière naturelle et modulaire.

Plusieurs système objets sont déjà disponible pour l'implantation de Scheme Gambit-C. La plupart sont inspirés de la puissance expressive du système CLOS. On retrouve entre autre le système orienté objet Meroon [40]. Ce système impose plusieurs limitations dont le fait que la hiérarchie de classe est limitée à un

Fonctionnalités désirées	`define-type`	Meroon	OOPS
Héritage multiple			X
Constructeurs génériques membres d'instances	X	X	X
membres de classe fonctions génériques polymorphique		X	X
dispatch sur des valeurs			X

TABLE 4.1 – Comparaison de la présence des fonctionnalités désirées par les implantation existantes de systèmes d'objets disponibles sur Gambit-C

héritage simple. Aussi, il n'est pas possible d'allouer des attributs de classes, communs à toutes les instances. OOPS [41] est un autre système objet disponible pour Gambit-C. Celui-ci offre plus de fonctionnalités que Meroon, dont la possibilité de créer des classes avec héritage multiples. Cette fonctionnalité est très pertinente pour l'écriture de logiciels, car elle permet l'agrégation de plusieurs modules de manière très élégante et offre beaucoup de flexibilité. Le tableau 4.1 effectue une comparaison de ces systèmes objets sur la présence de fonctionnalités utiles pour le développement de jeux vidéo.

Le tableau 4.3 de la section **??** illustre les performances de ces deux systèmes. On constate que les systèmes *define-type* de Gambit-C et Meroon sont beaucoup plus performant que OOPS. Il en résulte donc qu'aucun de ces systèmes ne semblent offrir de bonnes performances tout en donnant accès aux fonctionnalités de la programmation orientée objet de haut niveaux énoncés dans le tableau 4.1. Ainsi, afin d'obtenir un compromis entre l'expressivité et les performances, un nouveau système d'objet a été développé. Ce dernier aura comme vocation de répondre le

mieux possible aux besoins en efficacité d'un jeu vidéo, tout en apportant le plus d'outils et de puissance d'abstraction que possible. Ce dernier est décrit en détails dans le reste de ce chapitre.

4.1 Description du système

Le système de programmation orientée objet développé est fortement inspiré du Common Lisp Object System [38] et possède la plupart des propriétés intéressantes de ce dernier. On retrouve entre autre de l'héritage multiple de classes, du polymorphisme d'instance de classes et surtout, des fonctions génériques à *dispatch* multiple personnalisé. Malgré ces fonctionnalités de haut niveaux, une attention a été apportée aux performances, tout spécialement à l'accès aux membres d'instances, puisque ces opérations sont fréquentes dans un jeu vidéo. En effet, le développement préliminaire de *Space Invaders* (chapitre 6)nous laissait déjà sous-entendre que l'accès aux membres d'instances est une opération très fréquente qui se doit d'être efficace afin de ne pas ralentir la cadence du jeu.

Le reste de cette section fourni un API (*Application Programming Interface*) au système, tout en motivant l'ajout des fonctionnalités clés de celui-ci et fournissant des exemples d'utilisations.

4.1.1 Définition de classes

La définition de classes, faite via la forme spéciale *define-class*, est une version simplifiée de la forme `defclass` de CLOS. La syntaxe de `define-class` est présentée dans la figure 4.3.

```
(define-class <class-name> (<super1> ...) <member1> ...)
```

FIGURE 4.3 – Syntaxe de la forme spéciale `define-type`.

Le méta symbole `<class-name>` représente le nom que portera la classe définie. Suivant ce nom se trouve la liste de super classes, données par leurs noms respectifs. Par la suite, les membres de la classe sont donnés. Ces membres peuvent être un attribut d'instance `(slot: <slot-name>)`, un attribut de classe `(class-slot: <slot-name>)` ou un constructeur d'instances `(constructor: <fun>)`.

Cette forme spéciale, à la base, est très similaire avec la forme spéciale `define-type` fournie par Gambit-C et fut conçue pour que les fonctions utilitaires générées soient compatibles avec celles générées par `define-type`. Ainsi, une conversion d'un système basé sur les objets de *define-type* est extrêmement simple, il ne suffit qu'à remplacer la définition du type par une définition de classe de notre système d'objets. Les figures 3.25 et 4.4 illustrent la conversion de code nécessaire pour passer d'une forme `define-type` à notre système objet. Aussi, le choix d'utiliser des fonctions compatibles à `define-type` rend la base du système d'objets triviale à utiliser par des nouveaux utilisateurs déjà familier Gambit-C.

```
(define-class point () (slot: x) (slot: y))
(define-class circle (point) (slot: r))
(define (add p1 p2) (make-point (+ (point-x p1) (point-x p2))
                                (+ (point-y p1) (point-y p2))))
(add (make-point 1 2) (make-circle 3 4 5)) => #(¡class-desc¿ 4 6)
(if (circle? (make-point 1 2)) 'ok 'no)    => no
```

FIGURE 4.4 – Utilisation simple du système objets pour une utilisation similaire à celle pouvant être fait avec la forme `define-type`.

Les fonctions générées par la définition de la classe `point` sont :

- `(make-point <x-val> <y-val>)` : Construit une nouvelle instance de la classe avec `<x-val>` et `<y-val>` comme valeurs initiales d'attributs.
- `(make-point-instance)` : Construit une nouvelle instance *non initialisée* de la classe `point`.
- `(point-x <instance>)` : Accesseur de l'attribut `x` de l'instance passée en argument. Une autre fonction est similairement générée pour l'attribut `y`.
- `(point-x-set! <instance> <new-x-val>)` : Fonction de mutation de l'attribut `x`. Une autre fonction est similairement générée pour l'attribut `y`.
- `(point? <any-value>)` : Prédicat de type de la classe `point`. Cette fonction retourne la valeur `#t` si la valeur passée est une instance de la classe `point` ou de n'importe qu'elle sous-classe de celle-ci. La fonction retourne `#f` sinon.

En plus de pouvoir définir des attributs d'instances, il est possible de définir des attributs de classes, communs à toutes les instances de la classe. Ainsi les

informations communes à toutes les instances peuvent être factorisées, tout en conservant l'encapsulation de l'information dans l'objet. Il est important de noter qu'après la déclaration de la classe, tous les attributs de classes se retrouvent dans un état non-initialisé et doivent donc être initialisés avant de pouvoir être utilisés. L'héritage d'attributs de classe fait en sorte que chaque sous-classe possède sa propre version de l'attribut, permettant ainsi des personnalisations de chacune de ces sous-classes avec cet attribut.

Puisque ces attributs sont communs à toutes les instances, ils ne nécessitent pas de ces dernières pour être accédés ou modifiés. Toutefois, ces fonctions d'accès ou de modification de valeur d'attributs de classes permettent de prendre optionnellement une instance afin d'utiliser ces fonctions de manière polymorphique. La figure 4.5 illustre ce processus.

```
(define-class toto () (class-slot: t))
(define-class blub (toto))

(toto-t-set! 10) (blub-t-set! 'allo)
(toto-t)             => 10
(blub-t)             => allo
(toto-t (make-toto)) => 10
(toto-t (make-blub)) => allo
(toto-t-set! (make-blub) 'salut)
(toto-t (make-blub)) => salut
```

FIGURE 4.5 – Exemple d'utilisation dynamique d'attributs de classes.

En plus des créateurs d'instances à la `define-type`, le système objet permet la définition d'un ou de plusieurs constructeurs pour la création de nouvelles instances

simplifiées. Ces constructeurs sont plutôt des fonctions d'initialisation de nouvelles instances. Elles *doivent* avoir un premier argument correspondant à l'instance à initialiser et peuvent posséder un nombre arbitraire d'arguments supplémentaires. Ces constructeurs sont en fait utilisés pour créer une nouvelle instance de la fonction générique `init!`, et donc, il est également possible d'utiliser la discrimination de type faite par les fonctions générique, comme décrit dans la section 4.1.2. L'accès aux constructeurs se fait par la forme spéciale `new` qui s'occupe de créer la nouvelle instance de l'objet et d'appeler la fonction générique `init!`. La figure 4.6 illustre l'utilisation de constructeurs génériques d'objets.

```
(define-class titi () (slot: i))
(define-class toto ()
  (slot: t)
  (constructor: (lambda (self t) (toto-t-set! self t)))
  (constructor: (lambda (self (t titi)) (toto-t-set! self (titi-i t)))))

(let ((obj1 (new toto 1))
      (obj2 (new toto (new titi 2))))
  (toto-t obj1)  => 1
  (toto-t obj2)) => 2
```

FIGURE 4.6 – Utilisation générique de constructeurs

Finalement, l'utilisation de *hooks* sur les fonctions accesseur et de mutation d'attributs est possible en ajoutant des mots clés appropriés à l'intérieur des déclarations attributs en question. Il est possible de définir des *hooks* en lecture ou en écriture avec respectivement les mots clés `read-hooks:` et `write-hooks:`. La figure 4.7 illustre l'utilisation de ceux-ci.

```
(define-class hooked ()
  (slot: s
    (read-hooks:  (lambda (obj x) (display "r") (display x)))
    (write-hooks: (lambda (obj x) (display "w") (display x))))
  (class-slot: cs
    (read-hooks:  (lambda (x) (display "r") (display x)))
    (write-hooks: (lambda (x) (display "w") (display x)))))
(let ((o (make-hooked 1)))
  (hooked-cs-set! 'allo)   => wallo
  (hooked-s o)             => r1
  (hooked-s-set! o 2)      => w2
  (hooked-s o)             => r2
  (hooked-cs)              => rallo
  (hooked-cs-set! 'salut)  => wsalut
  (hooked-cs))             => rsalut
```

FIGURE 4.7 – Exemple d'utilisation de *hooks* sur des membres d'instances et de classes

4.1.2 Définition de fonctions génériques

Toujours avec une forte inspiration du modèle éprouvé du système objets de *Common Lisp* [38], le système d'objet développé possède des fonctions génériques permettant de faire du *dispatch* multiple sur les arguments de celles-ci. Le choix d'adhérer à la philosophie de *Lisp* s'explique non seulement par le fait que nous souhaitons que notre système objet puisse offrir le plus de puissance expressive que possible aux programmeurs de jeux vidéo, mais surtout puisque plusieurs problèmes se retrouvant dans le développement de jeux vidéo peuvent être exprimés de manières très concise avec des fonctions génériques. Par exemple, la résolution de collision est un comportement qui dépend des deux entités impliquées et donc l'utilisation d'une fonction générique pour effectuer la résolution du travail à accomplir se fait

de manière très naturelle.

La déclaration d'une nouvelle fonction générique se fait par le biais de la forme spéciale

```
(define-generic <gen-fun-name>)
```

qui ne prend que le nom de la fonction générique définie, puisque le nombre de ses arguments peut être variable. Les déclarations d'instances de fonctions génériques se font par l'entremise de la forme spéciale

```
(define-method (<gen-fun-name> <arg-desc1> ...)  <body>)
```

qui possède une syntaxe similaire à une définition de fonction, mais où la syntaxe des arguments a été étendue. Une description d'argument est généralement une liste de la forme `(<arg-name> <arg-type>)`, où le premier élément est le nom de la variable et le deuxième le type de celle-ci. Le type est normalement le nom de la classe à laquelle l'instance peut appartenir (incluant le polymorphisme).

Une description d'argument peut être uniquement un symbole qui sera le nom de la variable liée à la valeur passée en argument. Puisque dans un tel cas, aucune information n'est donnée sur l'argument à recevoir, aucune discrimination d'instance ne sera fait en utilisant cet argument, outre le fait qu'il s'agit d'un argument de plus. Le type de l'argument sera implicitement `*` et donc, une telle déclaration est strictement équivalente à `(<arg-name> *)`. La figure 4.8 présente un exemple de déclarations et d'utilisations d'instances d'une fonction générique effectuant des additions sur des objets s'étalant sur plusieurs types et avec un nombre variable de

paramètres.

```
(define-class length () (slot: l))
(define-class point () (slot: x) (slot: y))
(define-class circle (point) (slot: radius))
(define-generic add)
(define-method (add x y) (+ x y))
(define-method (add x y z) (+ x y z))
(define-method (add (p1 point) (p2 point))
  (new point (+ (point-x p1) (point-x p2))
             (+ (point-y p1) (point-y p2))))
(define-method (add (p point) (l length)) (+ (point-x p) (length-l l)))

(let ((p (new point 1 2)) (l (new length 5)))
    (add 10 11)         => 21
    (add 10 11 12)      => 33
    (point-x (add p p)) => 2
    (add p l))          => 6
```

FIGURE 4.8 – Exemple de déclarations et d'utilisations d'une fonction générique.

Le type d'un argument d'une méthode, peut être membre d'une syntaxe étendue permettant de faire de la discrimination d'instances de fonctions génériques sur la *valeur* des paramètres passés. Cette syntaxe est l'une parmi :

- `(match-value: <value>)` : La valeur du paramètre actuel doit être égal (selon `equal?`) à `<value>` pour pouvoir discriminer ce paramètre d'une fonction générique. L'expression `<value>` est considérée constante.

- `(match-member-value: <class> <slot-name> <value>)` : Le paramètre actuel doit être une sous-classe de `<class>` et la valeur de son attribut `<slot-name>` doit être égale (selon `equal?`) à `<value>`. L'expression `<value>`

est considérée constante.

- `(or <match1> ...)` : La valeur du paramètre doit correspondre à *au moins une* des clause `<match>` pour pouvoir discriminer l'instance de la fonction générique correspondante. Les clauses `<match>` doivent être une déclaration de type correcte. On peut retrouver ainsi un nom de classe ou l'une de ces quatre clauses spéciales.

- `(and <match1> ...)` : La valeur du paramètre doit correspondre à *toutes* les clauses `<match>` pour pouvoir discriminer l'instance de la fonction générique correspondante. Les clauses `<match>` doivent être une déclaration de type correcte. On peut retrouver ainsi un nom de classe ou l'une de ces quatre clauses spéciales.

Un exemple simple de discrimination par valeur de paramètre est donné dans la figure 4.9. Il est important de noter que la discrimination fonctionne bien lorsque les instances effectuent de la discrimination orthogonale, *i.e.* que les possibilités ne se recoupent pas entre elles. Une discrimination arbitraire pourrait se produire si plusieurs instances peuvent être discriminées de manières équivalentes pour un même appel.

Une note importante reliée à l'utilisation des fonctions génériques est que les instances discriminant sur une classe données doivent apparaître *après* la déclaration de celle-ci, sinon le comportement du système est indéterminé.

```
(define-class point () (slot: x) (slot: y) (class-slot: zero))
(point-zero-set! (new point 0 0))

(define-method (div (p point) (n (match-value: 0)))
  'division-by-zero!)

(define-method (div (p point) n)
  (new point (/ (point-x p) n) (/ (point-y p) n)))

(define-method (div (p (and (match-member: point x 0)
                            (match-member: point y 0)))
                    n)
  (point-zero))

(point-x (div (new point 1 2) 2))          => 1/2
(div (new point 1 2) 0)                    => division-by-zero!
(eq? (div (new point 0 0) 5) (point-zero)) => #t
```

FIGURE 4.9 – Exemple de discrimination d'instance de fonctions génériques par la valeur des arguments.

Puisque le système d'objet supporte de l'héritage multiple, il est possible qu'une fonction générique possède uniquement deux instances correspondant à des super classes d'une classe donnée, comme c'est le cas dans la figure 4.10. Les classes `colored` et `circle` sont toutes deux des super classes de `colored-circle` et il n'existe pas d'instance de la fonction générique `test` discriminant directement sur le type `colored-circle`, ainsi le système doit choisir entre les deux instances disponibles puisque ces deux dernières sont utilisables pour une instance de la classe `colored-circle`. L'algorithme discrimine alors comme suit : il considère que l'instance de la fonction générique qui possède *la plus profonde hiérarchie de classe* est la plus spécifique. Cette profondeur est approximée par la somme du nombre de su-

per classe que possède chacun des arguments de l'instance de la fonction générique en question. Ainsi, l'instance `(test (x colored))` obtient une valeur de 0, puisque `colored` ne possède aucune super classe et l'instance `(test (x circle))` obtient une valeur de 1, puisque la classe `circle` est une sous-classe de `point`. C'est donc l'instance utilisant un objet de type `circle` qui est utilisée. Si les valeurs de discrimination sont égales pour plusieurs instances, alors c'est la méthode définie en dernier qui sera utilisée.

```
(define-class point () (slot: x) (slot: y))
(define-class circle (point) (slot: radius))
(define-class colored () (slot: color))
(define-class colored-circle (circle colored))

(define-method (test (x colored)) 'colored)
(define-method (test (x circle)) 'circle)

(test (new colored-circle 1 2 3 'red)) => circle
```

FIGURE 4.10 – Exemple illustrant l'algorithme de sélection d'instances de fonctions génériques.

Finalement, il est possible d'utiliser un *cast* pour effectuer l'appel spécifiquement désiré. La syntaxe d'utilisation est la suivante : `(<genfun> cast: (<arg1-ty> ...) <arg1> ...)`Le *cast* est passé comme mot clé dans l'appel de la fonction générique et son argument doit être une liste des types de l'instance à choisir. Un exemple d'utilisation du *cast* est illustré dans la figure 4.11. Les arguments de la fonction générique `init!` sont d'abord évalués. Il en résulte qu'une instance non initialisée de la classe `container` est créée ainsi qu'une instance de la classe

`circle`. Le constructeur de la classe `circle` utilise un *cast* afin de pouvoir utiliser également le constructeur de la classe `point`. Ensuite, l'objet conteneur est initialisé en choisissant explicitement le constructeur attendant un argument de type `point`. L'objet correctement initialisé est finalement retourné.

4.1.3 Fonctions et formes spéciales utilitaires

Plusieurs fonctions et formes spéciales utilitaires sont disponibles afin de simplifier les tâches répétitives ou bien de permettre une introspection limitée avec le système objet. On retrouve plusieurs fonctions permettant d'obtenir de l'information sur les types d'instances ou de classes. Parmi celles-ci, les principales sont :

1. `(instance-of? <any-value> <class-name>)` : Prédicat similaire au mot clé de même nom en Java qui détermine si la valeur passée est une instance de la classe spécifiée. Ce prédicat retourne vrai si et seulement si cette valeur est une instance directe (et non polymorphe) de cette classe. Cette fonction est donc beaucoup plus efficace que sa contrepartie `is-subclass?`.

2. `(is-subclass? <any-value> <class-name>)` : Prédicat similaire au précédent, qui retourne vrai si et seulement si la valeur est une instance (incluant les instance polymorphes) de la classe spécifiée.

3. `(find-class? <class-name>)` : Retourne le descripteur de la classe spécifiée si elle existe, ou `#f` sinon.

```
(define-class point () (slot: x) (slot: y)
  (constructor: (lambda (self x y)
                  (pp 'creating-point)
                  (point-x-set! self x) (point-y-set! self y))))
(define-class circle (point) (slot: radius)
  (constructor: (lambda (self x y r)
                  (pp 'creating-circle)
                  (circle-radius-set! self r)
                  (init! cast: '(point * *) self x y))))
(define-class container () (slot: obj)
  (constructor: (lambda (self (p point))
                  (pp 'containing-a-point)
                  (container-obj-set! self p)))
  (constructor: (lambda (self (c circle))
                  (pp 'containing-a-circle)
                  (container-obj-set! self c))))

(init! cast: '(container point) (make-container-instance) (new circle 1 2 3))
  => creating-circle
     creating-point
     containing-a-point
     < container-instance ... >
```

FIGURE 4.11 – Exemple d'utilisation de *cast* pour l'appel de fonctions génériques.

4. `(get-class-id <any-value>)` : Retourne le nom de la classe associée à la valeur spécifiée. S'il s'agit d'une instance de classe, alors le nom de la classe sera retourné, sinon le type universel `*` sera retourné.

5. `(get-supers <class-name>)` : Retourne la liste de toutes les super classes associées à la classe spécifiée.

Ainsi, l'introspection demeure limitée en comparaison à celle rendue disponible par le protocole de méta objet de CLOS, mais elle demeure tout de même fonctionnelle. Une meilleure introspection permettrait de pouvoir développer des classes encore plus génériques, mais cette fonctionnalité n'est pas nécessaire pour le développement de jeu.

D'autre part, trois formes spéciales sont fournies pour simplifier les tâches répétitives pour la gestions d'instances de classes. Celles-ci sont :

- `(new <class-name> <value1> ...)` : Forme spéciale créant une nouvelle instance non-initialisée et appelant la fonction générique d'initialisation `init!` avec cette nouvelle instance et les valeur actuelles spécifiées.

- `(update! <instance> <class-name> <slot-name> <fun>)` : Cette forme spéciale permet de modifier facilement la valeur d'un attribut en fonction de sa valeur précédente. La fonction passée doit prendre un seul argument en paramètre, qui est la valeur courante de cet attribut. La valeur retournée par la fonction sera alors utilisée comme nouvelle valeur de cet attribut.

- `(set-fields! <instance> <class-name> ((<slot-name> <value>) ...))` : Cette forme spéciale permet de modifier de manière plus concise la valeur de plusieurs attributs en même temps. Elle est entre autre très utile dans l'écriture de constructeurs d'instances.

La figure 4.12 illustre un exemple d'utilisation de ces formes spéciales.

```
(define-class point () (slot: x) (slot: y))
(define-class colored-circle (point) (slot: radius) (slot: color)
  (constructor:
   (lambda (self x y r c)
     (init! cast: '(point * *) self x y)
     (set-fields! self colored-circle ((radius r) (color c))))))
(let ((p (new colored-circle 1 2 3 'red)))
  (update! p point x (flip + 1))

  (point-x p)) => 2
```

FIGURE 4.12 – Exemple d'utilisation des formes spéciales fournies par le système objet.

4.2 Implantation

Afin de pouvoir implanter ce système objet comme une extension directe du langage Scheme, les macros Scheme ont dû être utilisées. Il en résulte donc en l'ajout d'un langage spécifique au domaine de la programmation orientée objet.

Puisque l'expansion de macros Scheme se fait lors de la compilation, il devient très avantageux de faire le plus de travail possible durant cette expansion afin de pouvoir obtenir de bonnes performances lors de l'exécution. Par contre, afin de

pouvoir effectuer beaucoup de travail lors de cette expansion, beaucoup d'informations doivent être disponibles de manière statique, ce qui brime la philosophie dynamique du système objet. Ainsi, nous avons opté pour un compromis entre la division du travail effectué lors de la compilation et de celui fait lors de l'exécution du système d'objets.

De plus, les informations recueillies lors de l'expansion macro sont souvent nécessaire au fonctionnement du système lors de son exécution et donc, un passage d'informations entre ces deux mondes doit être fait. Afin d'y arriver, les structures correspondantes (descripteurs de classes, descripteurs de fonction génériques, etc...) sont recréés durant l'exécution. Afin de pouvoir faire plus facilement la distinction entre ces deux types de structures, les préfixes `mt` (*macro expansion time* et `rt` (*runtime*) sont utilisés dans le code source.

Les sections qui suivent expliquent les grandes lignes de l'implantation de ce système objet et justifient les choix d'implantations qui ont dû être faits.

4.2.1 Implantation de define-class

Le fonctionnement de `define-class` est intimement lié aux structures de données utilisées pour conserver l'information sur les classes afin de transmettre celle-ci aux instances. L'expansion de la forme spéciale `define-class` résulte en la définition de plusieurs fonctions de création et d'accès aux données d'instances, mais aussi en la création d'un *descripteur de classe*. Ce dernier à pour rôle de conserver non

seulement les super classes associées à la classe définie, mais aussi de conserver l'information permettant d'accéder aux attributs d'une instance. Les descripteurs de classes utilisés durant l'exécution sont une version légèrement simplifié des descripteurs utilisés durant l'expansion de macro.

Le polymorphisme d'accès aux attributs d'instances est possible grâce à une indirection à partir de ce descripteur de classe. Malgré le fait qu'une telle indirection ralentie légèrement l'accès aux attributs, cette dernière permet aussi d'avoir des instances qui ne possèdent pas de d'espaces inutilisés. Ainsi, des index donnant l'emplacement des attributs dans les instances sont conservés dans les descripteurs de classes. Puisque l'héritage multiple est permis, ces index peuvent varier d'une sous-classe à l'autre. Il en résulte donc que ces index d'indirection doivent toujours se retrouver au même endroit dans les descripteurs de classes afin que les accesseurs des super-classes sachent où trouver cette information. La figure 4.13 contient un exemple de définitions de classes permettant d'illustrer les structures utilisées lors de l'exécution du système. La figure 4.14 schématise ces structures.

Il aurait été possible de faire en sorte que ces indexes soit toujours au même endroit pour une hiérarchie de classe donnée, mais nous avons simplifier le problème dans notre implantation de la manière suivante : les index d'indirection se retrouvent au même endroit dans le descripteur de classe, peu importe la classe. La conséquence de cette simplification est que la taille des nouveaux descripteurs de classes augmente linéairement en fonction du nombre d'attributs qui ont été définis

```
(define-class point () (slot: x) (slot: y) (class-slot: id))
(define-class length () (slot: l))
(define-class point3d (point) (slot: z))

(point-zero-set! 'toto)
(point3d-zero-set! 'titi)

(define p (new point 1 2))
   => #(#(point () 1 2 toto) 1 2)
(define distance (new length 5))
   => #(#(length () unknown-slot unknown-slot unknown-slot 1) 5)
(define p2 (new point3d 4 5 6))
   => #(#(point3d (point) 1 2 titi unknown-slot 3) 4 5 6)
```

FIGURE 4.13 – Exemple concret de structure d'instance, descripteur de classe et de fonction d'accès à un attribut par indirection.

pour les classes précédemment définies. Une autre possibilité aurait été de toujours placer les attributs au même endroit dans les instances. Puisqu'en général peu de classe sont définies en comparaison à la quantité d'instances créées, le choix effectué est un bon compromis entre l'utilisation de la mémoire et la rapidité d'accès aux attributs.

Ces structures de données sont implantées en utilisant des vecteurs Scheme. Ceux-ci ont été choisis dans le but de donner un accès rapide aux attributs de ces derniers.

Lorsque la forme spéciale `define-class` est expansée, plusieurs informations sont extraites et traitées afin de pouvoir générer correctement les fonctions de création et d'accès aux attributs. Entre autre, il faut obtenir la liste complète de tous les attributs hérités par les classes parents à la classe définie. Cette informa-

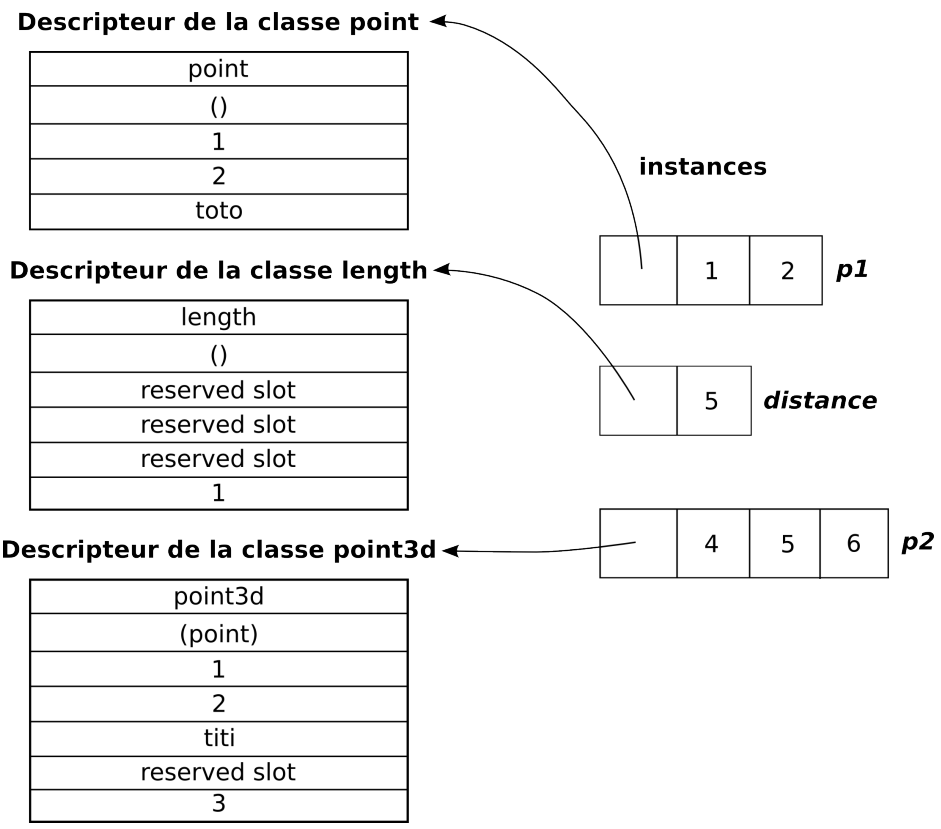

FIGURE 4.14 – Schématisation des structures créées dans la figure 4.13

tion est stockée dans une table de hachage globale à l'expansion macro nommée `mt-class-table` qui conserve l'information disponible pour chaque classe déjà créées. Cette information, stockée sous forme de *méta descripteurs de classes*, n'est disponible que pour l'expansion macro. Ces descripteurs gardent trace du nom des classes créées, de leurs hiérarchie et des positions relatives aux descripteurs de classes de l'exécution indiquant où trouver les indexes d'indirection associés à leurs attributs.

Lorsque les nouveaux attributs (attributs non-hérités) sont inclus dans descrip-

teurs de classe de l'exécution, un compteur global (expansion macro) est utilisé pour obtenir la position du nouveau champs dans le vecteur correspondant à ce descripteur. Une fois toutes les positions déterminées, la génération du descripteur de classe de l'exécution peut être entamée. Un deuxième index, local celui-ci, sera utilisé pour trouvé l'index d'indirection des attributs dans l'instance. Ainsi, l'index global pointe vers l'endroit où placer l'index d'indirection des attributs d'instance dans le descripteur de classe de l'exécution et les index d'indirection eux indiquent où trouver les attributs dans les instances. La création du descripteur de classe utilisé durant l'exécution. La variable `field-indices` est une structure de donnée contenant l'information sur les attributs (nom, type, position globale dans le descripteur de classe).

Une fois cette information en main, toutes les fonctions d'accès et de modification d'attributs peuvent être générées sans problèmes. La plupart contiendront des indirections pour l'accès aux champs, sauf pour les attributs de classes qui eux, se retrouvent directement dans les descripteurs de classes. La figure 4.16 donne un exemple concret de code généré effectuant l'accès au champs `x` de la classe `point` définie dans la figure 4.13. Au total, trois indirection sont nécessaire à l'obtention de la valeur d'un attribut d'instance. La première indirection obtient le descripteur de classe de l'instance en question. La deuxième cherche dans ce descripteur l'index d'indirection qui donnera la position de l'attribut dans l'instance. Puisque la position de cet index d'indirection est globale, elle est directement insérée dans le code

```
(let* ((instance-index 1) ; 0 -> class-desc
       (desc (make-class-desc
              name supers
              (if (pair? field-indices)
                  (- (slot-index
                      (cdar (take-right field-indices 1)))
                     1)
                  0))))
  (for-each
   (lambda (fi)
     (let ((index (slot-index fi)))
       (cond ((is-instance-slot? fi)
              (vector-set! desc index (instance-index++)))
             ((is-class-slot? fi)
              (vector-set! desc index 'unbound-class-slot))
             (else
              (error "Unknown slot type")))))
   (map cdr field-indices))
  desc)
```

FIGURE 4.15 – Création du descripteur de classe

généré. Dans cette exemple, cette position se trouve à l'index 2 dans le descripteur de classe. Finalement, la valeur de l'attribut est obtenue en effectuant l'indirection sur l'instance.

Deux constructeurs d'instances sont aussi générés. Le premier constructeur (`make-<class-name>`) construit et initialise *tous* les champs d'une instance de la même manière que procède le constructeur d'instance pour la forme `define-type` de Gambit-C. Le deuxième (`make-<class-name>-instance`) ne prend aucun argument et produit une instance non-initialisée qui pourra être passée à `init!`, la fonction générique d'initialisation d'objets. Par défaut, une instance de cette fonction

```
(lambda (#:obj518)
  (vector-ref #:obj518
              (vector-ref (instance-class-descriptor #:obj518)
                          2)))
```

FIGURE 4.16 – Code généré pour l'accès à l'attribut `x` de la classe `point` de la figure 4.13

générique est créée, et fait exactement le même travail que le premier constructeur. Aussi, une fonction très rudimentaire d'introspection d'instance de classe est aussi implantée sous la forme d'instances de la fonction générique `describe`.

Les choix d'implantation qui viennent d'être énumérés ont été fait dans le but d'obtenir un compromis entre la rapidité du système et la puissance expressive qu'il donne accès. Par exemple, on constate qu'une indirection est faite sur une instance afin d'obtenir son descripteur de classe, tandis qu'un pointeur vers ce descripteur de classe aurait pu être placé directement (voir la figure 4.16). Par contre, si un pointeur s'était retrouvé directement dans la fonction d'accès `point-x`, il n'aurait alors plus été possible de l'utiliser de manière polymorphique avec les sous-classes de la classe `point` puisqu'il n'est pas certain que l'attribut `x` se retrouvera au même endroit dans une instance de ces sous-classes. Puisqu'un accès au champs d'un vecteur est une opération efficace en Scheme, nous avons préféré opté pour la présence du polymorphisme qui apporte beaucoup en terme de puissance expressive.

Puisque le système objet est implanté en utilisant l'expanseur de macro `define-macro` de Gambit-C, les données conservées sur les classes définies durant cette expansion

sont perdues lorsque l'expansion est terminée. Il en résulte ainsi qu'il n'est pas possible de modifier la hiérarchie de classe définie dans un module Scheme (un fichier) dans un autre module. Par exemple, une classe `Alpha` définie dans un fichier `A` ne pourrait pas être sous-classée dans un autre fichier `B`. Toutefois, il est possible de créer et d'utiliser les hiérarchies de classes créées dans un autre module. L'implantation du support intermodulaire du système objet n'a pas été abordé dû à la complexité de la tâche et à la potentielle baisse de performance qui pourrait être apporté par un tel support.

4.2.2 Implantation de define-generic

De manière similaire à la définition de nouvelles classes, des informations sur les fonctions génériques définies et sur leurs instances sont conservées dans une table de hachage globale durant l'expansion macro et se nomme `mt-meth-table`. Ces informations seront par la suite transférées vers l'exécution du programme sous la forme de structures propres à chaque fonctions génériques qui se nomment `<genfun-name>-meth-table`. Ces structures contiennent le nom de la fonction générique, une table de hachage permettant de vérifier rapidement l'existence d'une instance en utilisant le type des arguments de cette fonction générique comme clé de la table et une liste *triée* des instances permettant d'accélérer le polymorphisme de fonction génériques.

L'expansion de la définition d'une nouvelle fonction générique résulte donc en

la création de cette structure et en la création d'une fonction, portant le nom de la fonction générique, qui a pour but de faire le choix de la bonne instance, en fonction des paramètres qui lui sont passés. La fonction de *dispatch* générée pour la fonction générique `init!` est illustrée dans la figure 4.17. Le code présenté a été légèrement modifié afin d'illustrer uniquement l'essence du *dispatch* effectué. On constate que dans un premier temps, on cherche dans la table de hachage de cette fonction générique si une instance associée aux types des arguments passés (ou du *cast* effectué) existe, et si ce ne pas le cas, on cherche explicitement une instance admissible de manière polymorphe à ces arguments.

```
(lambda (#!key cast . args)
  (let ((types (cond ((pair? cast) cast)
                     (else (map get-class-id args)))))
    (cond
      ((or (generic-function-get-instance init!-meth-table types))
           (find-polymorphic-instance? init!-meth-table
                                       args types))
           => (lambda (method)
                (apply (method-body method) args)))
      (else (error ...)))))
```

FIGURE 4.17 – Code expansé effectuant le *dispatch* dynamique pour la fonction générique `init!`

L'algorithme qui détermine qu'elle instance est la plus spécifique pour un ensemble d'argument donné est simple : On cherche a trouver la première instance dont chacun des types est considéré équivalent au type des arguments actuels *dans une liste triée* des instances en fonction de leur spécificité. Le critère de spécificité

est déterminé par la profondeur dans la hiérarchie du type des arguments de l'instance, *i.e.* par la somme du nombre de classes parents pour chacun des types des arguments passés. En ce qui concerne les types spéciaux comme `match-value`, ces types sont considérés comme étant très spécifiques et donc prioritaires à un simple correspondance du type d'un objet. La figure 4.18 illustre le code utilisé pour réaliser ce *dispatch* polymorphique.

```
(define (find-polymorphic-instance? genfun actual-params actual-types)
  (let ((args-nb (length actual-params))
        (sorted-instances (generic-function-sorted-instances genfun)))
    (exists (lambda (method)
              (equivalent-types? (method-types method)
                                 actual-params
                                 actual-types))
            (filter (lambda (i) (= (length (method-types i)) args-nb))
                    sorted-instances))))
```

FIGURE 4.18 – Implantation de la recherche d'instances polymorphiques d'une fonction générique

À un certain moment la fonction `call-next-method` avait été implantée en utilisant le mécanisme de variables à portée dynamique de Gambit-C. Par contre, l'exécution du corps des méthodes à l'intérieur d'un environnement dynamique ralentissait l'exécution de celles-ci d'un facteur d'environ 15% et donc, l'implantation de la fonction `call-next-method` a été abandonnée. En effet, cette fonction permet d'écrire des corps de méthodes légèrement plus génériques, mais l'utilisation du `cast` permet de faire sensiblement le même travail sans avoir à subir des pertes de performances.

4.2.3 Implantation de define-method

L'expansion de la macro `define-method` est très simple, elle consiste en la création d'une structure de donnée qui contient le corps de l'instance définie, ainsi que les types associés aux arguments attendus. Cette structure existera durant l'expansion et sera recréée lors de l'exécution du programme afin de pouvoir être appelée par la fonction de *dispatch*.

4.3 Performances

Ce système de programmation orienté objet a été écrit dans le but d'obtenir un compromis entre l'accès à des concepts de haut niveaux et de bonnes performances lors de l'exécution, notamment en ce qui à attrait à l'accès aux membres d'instances de classes.

Dans un premier temps, les performances du système sont analysées de manière théorique. Le tableau 4.2 donne la complexité algorithmiques des opérations utilisées. La création d'instances directe, *i.e.* utilisant les fonctions `make-<class-name>`, s'effectuent en temps constant puisqu'il ne s'agit que de la création d'un vecteur. Par contre, la création d'instances utilisant des constructeurs, *i.e.* en utilisant `(new <class-name> ...)`, possède la complexité des appels de fonctions génériques directs. Par « appel direct », il est sous-entendu que le types des arguments de l'appel correspondent aux types attendu directement (et non de manière polymorphe) par une instance de fonction générique existante. Ces appels possèdent une com-

opération	complexité algorithmique
création d'instances directe	$O(1)$
création d'instances avec constructeurs	$O(p)$
accès aux membres	$O(1)$
modification de membres	$O(1)$
dispatch direct	$O(p)$
dispatch polymorphique	$O(mp)$

TABLE 4.2 – Complexité algorithmique des opérations du système objets effectuées durant l'exécution

plexité algorithmique linéaire en fonction du nombre de paramètres (appelons le p) que possède l'instance. En effet, l'obtention d'une instance directe est faite par l'entremise d'une recherche dans une table de hachage utilisant `equal?` à titre de comparaison. Cette complexité augmente d'un ordre supplémentaire lorsqu'aucune instance directe n'existe et qu'alors une instance polymorphique est recherchée puisque une recherche linéaire est effectuée sur chacune des instance existantes (m), puis la compatibilité du type des arguments doit être effectué (p). Il est intéressant de noter que malgré la complexité quadratique de cette opération, le nombre de paramètre (p) est normalement très bas. L'accès et la modification de membres de classe se fait toutefois de manière constante.

Une comparaison pratique de performances a été effectués avec d'autres systèmes objets disponibles pour Gambit-C afin d'avoir une meilleure idée des constantes cachées dans la complexité des opération effectuées durant l'exécution. Il est aussi intéressant de pouvoir comparer les performances du systèmes par rapports aux autres afin de pouvoir confirmer l'obtention du résultat désiré. Ces résultats sont

Opération	define-type	Meroon	OOPS	class
création d'instances directe	0.182	0.111	0.819	0.044
création d'instances avec constructeurs	ND	ND	ND	1.397
accès aux membres	0.064	0.070	1.517	0.039
modification de membres	0.064	0.074	1.926	0.044
dispatch direct	0.144	0.350	9.709	1.115
dispatch polymorphique	ND	0.382	9.398	3.841

TABLE 4.3 – Temps en secondes nécessaire pour l'exécution de tests comparatifs pour plusieurs système objets disponibles sur Gambit-C

Opération	define-type	Meroon	OOPS	class
création d'instances directe	4.1	2.5	18.6	1.0
création d'instances avec constructeurs	ND	ND	ND	1.0
accès aux membres	1.6	1.8	38.9	1.0
modification de membres	1.5	1.7	43.8	1.0
dispatch direct	1.0	2.4	67.4	7.7
dispatch polymorphique	ND	1.0	24.6	10.0

TABLE 4.4 – Comparaison de performances relative au système le plus rapide (1.0) pour chaque opérations du système objet.

présentés dans le tableau 4.3. Ces derniers correspondent aux temps nécessaire à 500000 exécutions de chacunes des opérations principales du système objet. Les systèmes objets Meroon [40], OOPS [41] ainsi que les objets obtenus avec l'utilisation de la forme spéciale `define-type` de Gambit-C ont été utilisés pour effectuer cette comparaison. Le tableau 4.4 présente ces résultats d'une manière relative au meilleur temps d'exécution du test pour chaque opération vérifiée.

Ces résultats démontrent bien que les objectifs d'implantation ont été atteint. En effet, le système développé (`class`) offre de très bonnes performances quant à la création d'instances et la manipulation de membres de celles-ci. Malgré le fait que

le système est environ une ordre de grandeur plus lent que le système Meroon pour effectuer le *dispatch* dynamique de fonctions génériques, ce dernier offre beaucoup plus d'expressivité dont l'utilisation de constructeurs ou encore de discrimination sur des valeurs. Ainsi, le système développé en plus d'offrir plusieurs concepts de hauts niveau quant à la programmation orientée objet, se compare très bien aux autres systèmes d'objets performants disponibles pour Gambit-C.

4.4 Conclusion

Ainsi un système objet complet a été développé dans le but d'étendre le langage Scheme et d'y inclure le paradigme de la programmation orientée objet. Ce système objet permet la déclaration de classes avec héritage multiple, polymorphisme et fonctions génériques effectuant du *dispatch* multiple. Ces choix de caractéristiques ont été faits dans le but de donner le plus de liberté aux programmeurs de jeux vidéo, tout en fournissant de bonnes performances, tant en temps sur processeur qu'en utilisation de la mémoire.

Pour y arriver, les structures de données utilisées pour l'implantation d'instances de classes ne possèdent pas d'espaces inutilisés, et ne requièrent que trois indirection vectorielles afin de pouvoir accéder aux champs de celles-ci. De même, les fonctions génériques utilisent des mécanismes de tri pré-calculé afin d'accélérer le *dispatch* d'instances de fonctions génériques de manière polymorphe.

Il serait maintenant très intéressant de modifier le système afin qu'il supporte un

protocole de méta objets. Un tel protocole donnerait accès à une introspection très développée et permettrait aux utilisateurs de modifier facilement le comportement du système selon leurs besoins.

Aussi, une modification du système afin de le rendre compatible avec des classes de manière inter-modulaire serait une grande amélioration en ce qui a attrait à la modularité du code.

Par contre, une attention particulière devrait être portée aux coûts en performance que ces modifications pourraient impliquer afin de respecter la philosophie de base de ce système.

CHAPITRE 5

SYSTÈME DE COROUTINES

Les *threads* constituent un outil de programmation important permettant l'exécution parallèle matérielle ou logicielle de code. L'utilisation de *threads* dans un jeu vidéo peut être très pertinente et permettrait de pouvoir exprimer de manière concise beaucoup de concepts clés. Par exemple l'implantation de parties multi-joueurs où ces derniers jouent à tours de rôles se représente bien par un modèle parallèle, puisqu'il s'agit vraiment de deux parties distinctes qui sont jouées en même temps.

Le langage Scheme tel que décrit par le standard [6] ne fourni pas de système de *threads*. Par contre, le document SRFI 18 [42] (*Scheme Request For Implementation*) décrit un *API* (*Application Programming Interface*) de système de *thread* concurrents. Le système Gambit-C [7] supporte cet *API* sous forme de *threads* verts, *i.e.* sous formes de concurrence logicielle et non matérielle. Malheureusement, le coût d'utilisation des mécanismes de synchronisation explicites est très élevé en temps de développement et en utilisation du processeur lors de l'exécution.

Par contre, un système de coroutines (appelé aussi *threads* coopératifs) permettrait d'éviter d'avoir à spécifier explicitement ces synchronisations entre entités. En effet, si le flot de contrôle est changé durant des moments opportuns connus du programmeur, aucune synchronisation supplémentaire n'est requise pour assurer la validité d'accès concurrents à des sections critiques. En fait, le problème ne

se pose même plus, mais disparaît complètement. Ceci rend donc très attrayant de tels systèmes pour le développement de jeux vidéo. Il permettrait de pouvoir exprimer de manière simple des changements de contextes dans le jeu, ou même, de modulariser le comportement de chacune des entités du jeu.

Malheureusement, ni Scheme, ni Gambit-C n'offrent de tels système de coroutines. Par contre, l'expressivité du langage Scheme, notamment grâce à la réification de continuations, rend la tâche tout à fait accessible et réalisable. Ce chapitre vise donc à présenter un système de *threads coopératifs* développé dans le but de bien répondre aux besoins de jeux vidéo quant à l'écriture de code s'exécutant de manière parallèle de manière sécuritaire.

5.1 Description du langage

Un système de *threads* coopératifs implique donc que le changement de contexte d'exécution associés aux changements de *threads* doivent être faits de manière explicite par les utilisateurs. Ainsi, ces changements de contextes peuvent être faits aux moments opportuns, assurant ainsi l'intégrité des données. Toutefois, des synchronisations entre les différentes coroutines pourraient être encore nécessaires afin de bien orchestrer l'exécution de ces dernières. Afin de permettre aux coroutines de pouvoir communiquer entre elles, une synchronisation par passage de message avec *pattern matching* à la Termite [43] a été adoptée. Ce mécanisme permet de pouvoir synchroniser de manière élégante les coroutines d'une manière très naturelle.

De plus, le système a été conçu afin de permettre d'être utilisé de manière récursive. Il est donc possible d'avoir une coroutine qui sera elle-même un système de coroutine, et ainsi de suite. Cette fonctionnalité a été implantée de manière à ce que le système soit le plus générique possible. De plus, l'idée de système récursifs est aussi très près du langage Scheme dans lequel la récursion de fonctions est très courante et commune grâce à l'implantation de l'optimisation d'appels terminaux.

Aussi, la notion de temps a été abstraite dans le système avec l'introduction de compteurs de temps (*timers*). Il est donc possible de choisir non seulement une granularité temporelle en spécifiant la fréquence de se compteur de temps, mais il est aussi possible de spécifier un facteur d'accélération, permettant de pouvoir accélérer la simulation en cours.

Le système se résume à un ordonnanceur de coroutine qui utilise une file de coroutines prêtes pour choisir la prochaine de celles-ci à prendre le contrôle. Aussi, une file de coroutine en attente sur le temps et sur de conditions sont disponible afin de permettre une bonne régulation du système. Lorsqu'une coroutine décide de passer la main à la coroutine suivante ou lorsqu'elle ne doit plus attendre après le temps ou une condition, elle se fait enfiler à la fait de la file d'attente de coroutines prêtes. La figure 5.1 illustre l'architecture globale du système.

Les sections suivantes décrivent un *API* permettant l'utilisation du système de coroutines créé.

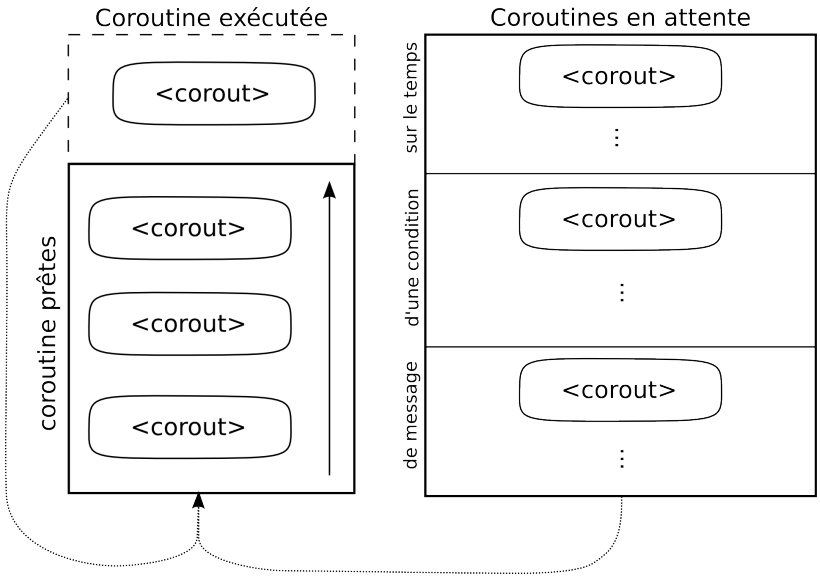

FIGURE 5.1 – Architecture globale du système de coroutine

5.1.1 Création de coroutines

Les coroutines sont des objets en eux même et peuvent être créés de manière externe au système de coroutines pour, par la suite, être intégrées à celui-ci. L'avantage de cette approche, utilisée aussi par le système de *threads* de Gambit-C, réside dans le fait que les objets correspondant aux *threads* peuvent être préparés à l'avance et conservés jusqu'au moment opportun de leurs intégration dans le système. Une nouvelle instance de coroutine s'effectue avec

```
(new-corout <corout-id> <thunk>)
```

où `<corout-id>` est un symbol permettant d'identifier la coroutine et le dernier argument est une fonction prenant aucun argument (appelée aussi *thunk*), qui contient le corps de l'exécution de cette coroutine.

L'objet créé pourra être par la suite intégré à un nouveau système par les fonctions d'initialisations `boot` et `simple-boot` ou encore intégrés à une coroutine d'une simulation existante avec la fonction `spawn-brother`. Ces fonction sont décrites plus bas.

Finalement, il est possible de rendre une coroutine prioritaire ou non prioritaire en utilisant les fonctions `(prioritize! <corout>)` et `(unprioritize! <corout>)`. Lors de leurs création, toute les coroutines sont considérées comme étant non prioritaire. Lorsqu'une coroutine devient prioritaire, elle sera automatiquement *la prochaine* a être exécutée lors d'un changement de contexte. Il est donc important que cette coroutine enlève sa priorité pour éviter d'obtenir des boucles infinies par la suite.

5.1.2 Démarrage du système

Le système de coroutines peut être démarrer en utilisant la fonction

```
(boot <timer> <return-val-handler> <corout1> ...)
```

dont le premier argument doit être un objet correspondant à un compteur de temps pour le système, le deuxième paramètre est une fonction permettant la gestion des valeurs de retours des coroutines et par la suite viennent les coroutines qui seront présentent au démarrage du système créé. L'ordre de ces coroutines est significatif car il indique dans quel ordre seront enfilées les coroutines dans la file d'attente de coroutines prêtes.

La création de *timers* se fait par un appel à la fonction

```
(start-timer! <period> [time-multiplier: <time-mult-value>]))
```

qui s'occupe de créer un objet faisant abstraction du temps avec une granularité associée à la période (en secondes) spécifiée. La fonction `(stop-timer! <timer>)` doit être utilisée afin d'arrêter le compteur de temps, lorsque son utilisation n'est plus nécessaire.

La fonction s'occupant de gérer les valeurs de retour des coroutines se doit de recevoir deux arguments, le premier étant le résultat accumulé des précédentes terminaisons de coroutines et le deuxième étant la valeur de retour de la dernière coroutine à avoir terminé son exécution. La valeur de retour de cette fonction sera alors utilisée comme la prochaine valeur accumulée lors de la terminaison subséquente d'une coroutine, comme c'est le cas avec la fonction `fold` (voir la figure 3.7).

La figure 5.2 présente un exemple de démarrage d'un système de coroutine où le temps sera augmenté toutes les secondes et où le résultat final de la simulation sera la somme des valeurs retournées par chacune des coroutines. Il est important de noter qu'il ne faut pas oublier d'arrêter le *timer* lorsqu'il n'est plus nécessaire.

Le processus de démarrage peut être simplifié avec l'utilisation de la fonction `(simple-boot <corout1> ...)` qui utilise un *timer* par défaut avec une période de 0.001 secondes et retourne la valeur de la dernière coroutine à être exécutée dans le système comme valeur de retour du système. La figure 5.3 illustre le démarrage

```
(let* ((c1 (new corout 'c1 (lambda () 1)))
       (c2 (new corout 'c2 (lambda () 2)))
       (c3 (new corout 'c3 (lambda () 3)))
       (timer (start-timer! 1.0))
       (result (boot timer + c1 c2 c3)))
    (stop-timer! timer)
    result)
```

6

FIGURE 5.2 – Exemple de démarrage d'un système de coroutines

d'un système de coroutines en utilisant `simple-boot`.

```
(let* ((c1 (new corout 'c1 (lambda () 1)))
         (c2 (new corout 'c2 (lambda () 2)))
         (c3 (new corout 'c3 (lambda () 3))))
    (simple-boot c1 c2 c3))
```

3

FIGURE 5.3 – Exemple de démarrage simple de système de coroutines

5.1.3 Manipulation du flot de contrôle

Puisque le contrôle du flot d'exécution entre les coroutines doit être explicité par l'utilisateur, une bonne diversité de fonctions permettent la manipulation de celui-ci. Il est important de noter que, sauf avec avis contraire, toutes les fonctions décrites dans cette section doivent être exécutée *par* une coroutine, *i.e.* dans leur corps.

La fonction la plus simple de manipulation du flot d'exécution est `(yield)`. Celle-ci arrête temporairement la coroutine actuelle et passe le contrôle à la *prochaine* coroutine disponible. La prochaine coroutine sera déterminée de manière

déterministe avec une simple file d'attente. La coroutine actuelle sera donc placée à la fin de cette file d'attente (à moins d'être prioritaire). Ainsi, le retour du contrôle à cette coroutine ne sera exécutée uniquement lorsque toutes les autres coroutines prêtes auront décidée de passer la main à la coroutine suivante. Il est également possible de choisir explicitement à quelle coroutine le contrôle sera passé avec la fonction `(yield-to <corout>)`.

Une coroutine peut aussi décider de se mettre en attente pour une certaine période de temps avec la fonction `(sleep-for <sec>)` un certain nombre de secondes. Il en résultera que la coroutine actuelle sera placée dans une file d'attente séparée et ne sera ré-enfilée dans la file de coroutines prêtes uniquement lorsque le délais prescrit sera dépassé. Cela n'implique pas que la coroutine sera exécutée à ce moment là, car si cette file contient déjà des coroutines en attentes, elle devra attendre sont tour pour pouvoir poursuivre son exécution.

Il est également possible pour une coroutine d'intégrer des nouvelles coroutines dans le système en utilisant les fonctions :

```
(spawn-brother <corout>)
(spawn-brother-thunk <corout-id> <thunk>)
```

La première intégrera simplement la coroutine spécifiée dans la file de coroutines prêtes. Cette coroutine *ne doit pas* déjà être présente dans le système. La deuxième fonction, `spawn-brother-thunk` créera une nouvelle coroutine ayant comme identificateur `<corout-id>` et la fonction spécifiée comme corps et sera, par la suite,

intégrée de même manière dans le système.

Une coroutine peut aussi altérer le fil de son exécution en modifiant la continuation de son calcul par celui d'une autre coroutine ou d'un *thunk* (fonction sans argument) de manière similaire à la forme spéciale `call/cc` de Scheme. Ainsi, un appel à la forme spéciale `(continue-with <corout>)` fera en sorte que la continuation de la coroutine devienne celle de cette autre coroutine. De manière similaire, `(continue-with-thunk! <thunk>)` utilisera la fonction passée comme continuation du calcul. La figure 5.4 présente un exemple utilisant ces deux formes spéciales. Lorsque la coroutine `c1` s'exécute, sa continue change pour celle de `c2` qui change à sont tour pour la fonction `t`. Ainsi, la fin du corps des coroutines `c1` et `c2` ne sera jamais exécutée puisque leurs continuation ont été altérées.

```
(let* ((t (lambda () (pp 'bonjour) 0))
       (c2 (new corout 'c2
                (lambda () (continue-with-thunk! t) (pp 'allo ) 2)))
       (c1 (new corout 'c1
                (lambda () (continue-with c2) (pp 'salut) 1))))
  (simple-boot c1))
```

bonjour
0

FIGURE 5.4 – Exemple de modification de la continuation d'une coroutine

Finalement, une coroutine peut changer sa continuation de manière plus drastique en forçant la terminaison du fil d'exécution de celle-ci en utilisant la fonction `(terminate-corout <return-val>)`. La valeur de retour de la coroutine sera alors la valeur spécifiée en paramètre. Il est aussi possible de faire terminer le système

actuel de coroutine, au complet, en faisant appel à `(kill-all! <return-val>)`. La valeur de retour finale de l'exécution du système sera alors la valeur spécifiée.

5.1.4 Environnement dynamique

Un environnement dynamique est disponible pour les coroutines, lors de leurs exécution. Cet environnement donne accès à de l'information sur ce dernier. il comprend les paramètres :

- `(current-corout)` : retourne l'instance de la coroutine actuellement exécutée.
- `(timer)` : retourne l'objet d'abstraction du temps associé avec le système courant. La valeur actuelle du temps peut être obtenue facilement en appellent plutôt la fonction `(current-sim-time)`.
- `(return-value)` : Valeur de retour accumulée des coroutines au moment de l'appel.

5.1.5 Système de communication inter coroutines

Le mécanisme principal de synchronisation des coroutines est basé sur le passage de messages, tel que fait dans le système de programmation distribué Termite [43]. Cette approche semble naturelle pour la programmation de jeux vidéo où les coroutines peuvent devenir des entités à part entière et pourraient ainsi communiquer avec d'autres entités par ces mécanismes d'une manière naturelle.

L'envoi de messages se fait par la fonction `(! <corout> <msg>)` qui s'occupe d'acheminer le message désiré à la coroutine spécifiée. La réception de messages, quant à elle, se fait avec deux fonctions distinctes et une forme spéciale :

- `(? [timeout: <sec>])` : Réception du premier message disponible avec attente bloquante. Un *timeout* peut être spécifié afin de limiter l'attente faite. Si plusieurs messages sont disponibles, le premier arrivé sera alors retourné, afin de respecter une politique équitable de réception de messages.

- `(?? <predicat> [timeout: <secs>])` : Réception du premier message qui retourne vrai selon le prédicat spécifié. Le prédicat doit prendre un seul argument, un message reçu. Comme pour la fonction `?`, l'attente bloquante peut être interrompue après un délais de temps, si ce dernier est spécifié.

- `(recv (<pattern> <body>) ...)` : Cette forme spéciale permet la réception sélective de messages selon des patrons de filtrage par motifs (*pattern matching*). Ces patrons permettent de pouvoir facilement exprimer la forme attendue du message et de pouvoir lier des variables locales à des parties du message reçu afin de les utiliser dans le corps du patron. Lorsque plusieurs messages reçus peuvent correspondre aux motifs donnés, la sélection du message se fait en choisissant dans un premier temps le premier motifs possible, et par la suite le premier message répondant à ce motif. Ainsi l'ordre de spécification des motifs est important et significatif.

Les valeurs de délais d'attente maximal spécifié pour la réception de messages implique qu'il est possible qu'une réception échoue. Dans un tel cas, une exception de type `mailbox-timeout-exception` est lancée par la fonction de réception utilisée.

Les patrons utilisés dans le filtrage de motifs de la forme spéciale `recv` diffèrent de ceux dans Termite. Ici, le patron doit être une donnée Scheme. Lorsque, dans cette donnée, un symbole est précédé d'une virgule (*unquote*), le symbole est *lié* à la valeur trouvée à cet endroit du motif dans le message reçu. Les formats de motifs reconnus sont les symboles, les mots clés, les caractères, les valeurs booléennes, les nombres, les chaînes de caractères, les listes et les vecteurs. Aussi, un patron spécial permettant d'effectuer des attentes bornées dans le temps est disponible sous la forme `(after <sec> <body> ...)`. Ce patron spécial doit être spécifié *en dernier*, s'il est présent. La figure 5.5 illustre plusieurs motifs différents qui peuvent être utilisés.

```
(recv (salut 'got-salut)                    ; symbol match
      ("bonjour" 'bonjour)                  ; strng match
      (1011 '11-or-1011?)                   ; number match
      ((a ,b c) (string b #\ k))            ; list match
      (#(a b ,c) c)                         ; vector match
      ((tata 1 #(toto ,x) "titi") (+ x 1)) ; complex match
      (,anything anything)                  ; can match anything
      (after 2 'timeout!))                  ; timeout
```

FIGURE 5.5 – Exemple de patrons pouvant être utilisés dans le filtrage par motifs de la forme spécial `recv`

Une particularité intéressante de ce filtrage par motif est reliée au fait qu'il est possible d'utiliser des patrons de manière dynamique, *i.e.* qui n'apparaissent pas dans la forme spéciale `recv`, mais plutôt qui ont été spécifiés grâce à une autre forme spéciale, `(with-dynamic-handlers ((<pattern> <handler-body>) ...) <body>)`. Tous les appels à `recv` se trouvant dans le corps (`<body>`) de `with-dynamic-handlers` se trouveront augmentés de ces nouveaux patrons. Il est important de mentionné que ces patrons dynamiques seront toutefois considérés en dernier lieu. Il est donc possible de factoriser des patrons communs et de les appliquer à tous les appels de la forme spéciale de réception de messages par filtrage de motifs.

Aussi, un mécanisme de liste de diffusion a été inclus. Ce mécanisme simple permet d'enregistrer des coroutine dans une liste de diffusion et, par la suite, d'envoyer à toutes les coroutines inscrites des messages de manière simultanée. Quoi que très primitif, ce système permet d'émuler la base d'un système de programmation réactive [44]. Les fonctions de gestion et d'utilisation de listes de diffusions sont :

- `(subscribe <list-id> <corout>)` : Inscription de la coroutine spécifiée à la liste de diffusion identifiée par le symbole choisi. Si la liste n'existait pas, elle sera créée.
- `(unsubscribe <list-id> <corout>` : Désinscription d'une coroutine à une liste de diffusion.

- `(broadcast <list-id> <msg>)` : Envoi d'un message à une liste de diffusion.

5.1.6 Autres mécanismes de synchronisation

En plus du mécanisme de messagerie sophistiqué du système de coroutine, les coroutines disposent d'un système de sémaphores afin de pouvoir exécuter des synchronisation de manière plus traditionnelle. Ces dernières sont manipulés par les fonctions :

- `(new-semaphore <init-value>)` : Création d'une nouvelle sémaphore ayant comme valeur initiale `<init-value>`.
- `(new-mutex)` : Création d'une nouvelle sémaphore ayant comme valeur initiale 1.
- `(sem-locked? <sem>)` : Permet de vérifier s'il reste des ressources disponibles dans une sémaphore. S'il en reste, la fonction retournera vrai, ou faux sinon.
- `(sem-lock! <sem>)` : Prend une ressource de la sémaphore spécifiée. Si aucune ressource n'est disponible, la coroutine se met en étant d'attente bloquante jusqu'à ce qu'une ressource soit libérée.
- `(sem-unlock! <sem>)` : Libère une ressource de la sémaphore spécifiée. Aucune vérification n'est faite qu'en a s'assurer que la coroutine courante possède

réellement cette ressource. Si des coroutines sont en attentes de ressources, la première à s'être mise en attente est alors réveillée.

L'utilisation de sémaphores n'est pas nécessaire, puisque le système de messagerie permet d'effectuer n'importe qu'elle synchronisation. Par contre, les sémaphores permettent de pouvoir exprimer d'une manières différente des synchronisations et donc apportent plus de liberté aux utilisateurs du système.

5.1.7 Systèmes en cascades

Le système de coroutine a été conçu de manière à pouvoir permettre de créer des systèmes de coroutines de manière cascadée, *i.e.* de créer des coroutines qui sont elles-mêmes des système de coroutines. La procédure de création de ces derniers est simple, il suffit de démarrer un nouveau système de coroutine à l'intérieur d'une coroutine existante. Par contre, un problème se pose : il devient alors impossible de pouvoir retourner le contrôle aux coroutines appartenant au système primordial, puisque `(yield)` ne fera que changer de contexte que les coroutines du sous-système.

Ce problème est résolu par l'utilisation de la fonction `(super-yield)` qui effectue un changement de contexte pour le système de coroutine courant. Si le système courant ne se retrouve pas dans un système cascadé, alors rien ne se produira. Une fonction similaire, `(super-kill-all! <return-value>)` terminera *tous* les systèmes de coroutines se retrouvant dans l'arborescence de système de coroutine

présentement active. Un exemple de système de coroutines en cascade est donné dans la figure 5.6. On constate que les appels à `super-yield` ont bien effectué les changements de contextes de des coroutines hôtes des sous-systèmes de coroutines (`s1` et `s2`).

```
(let* ((ret (lambda (x) (pp `(now returning: ,x)) x))
       (c1 (new corout 'c1 (lambda () (ret 1))))
       (c2 (new corout 'c2 (lambda () (super-yield) (ret 2))))
       (c3 (new corout 'c3 (lambda () (ret 3))))
       (c4 (new corout 'c4 (lambda () (ret 8))))
       (c5 (new corout 'c5 (lambda () (super-yield) (ret 4))))
       (c6 (new corout 'c6 (lambda () (ret 2))))
       (timer (start-timer! 1.0))
       (s1 (new corout 's1 (lambda () (boot timer + c1 c2 c3))))
       (s2 (new corout 's2 (lambda () (boot timer - c4 c5 c6)))))
  (boot timer * s1 s2))
```

(now returning : 1)
(now returning : 8)
(now returning : 2)
(now returning : 3)
(now returning : 4)
(now returning : 2)
12

FIGURE 5.6 – Exemple de démarrage de système de coroutines en cascade

5.2 Implantation

L'implantation de ce système de coroutine est centralisée sur l'utilisation de la forme spéciale `call/cc` afin de permettre la conservation de l'état courant d'une coroutine.

Un net avantage lorsque le système est directement implanté par l'utilisateur

est qu'il répond sur mesure aux demandes de ce dernier et donc il peut s'exprimer dans une syntaxe simple, tout en conservant un contrôle fin sur le comportement du système.

Cette section explique les mécaniques internes du système de coroutines développé, en expliquant non seulement les structures de données utilisées, mais aussi les algorithmes utilisés.

5.2.1 Implantation des coroutines

La structure de donnée des coroutine est implantée en utilisant le système d'objets fournit dans le système Gambit-C. Une version orientée objet est aussi disponible. La structure employée est illustrée dans la figure 5.7. Elle contient toutes les informations essentielles au fonctionnement de celle-ci, dont entre autre sa continuation (`kont`), sa boîte de messagerie, une sauvegarde de l'environnement d'un sous-système de coroutine (`state-env`), etc...

La continuation primordiale d'une coroutine est sa terminaison de manière « propre », *i.e.* en utilisant la fonction de terminaison de coroutines. C'est ce qui permet à une coroutine ayant comme corps uniquement `(lambda () 1)` de terminer correctement avec la valeur de retour 1.

La variable d'état `sleeping?` permet d'indiquer à l'ordonnanceur de savoir si la coroutine qui vient de céder le contrôle a été mise en veuille ou non, afin de savoir si cette dernière doit retourner dans la file d'attente des coroutine prêtes.

```
(define-type corout id kont mailbox state-env
                    prioritize? sleeping? delta-t msg-lists)
(define (new-corout id thunk)
  (let ((kont (lambda (dummy) (terminate-corout (thunk))))
        (mailbox (new-queue))
        (state-env #f)
        (prioritize? #f)
        (sleeping? #f)
        (delta-t #f)
        (msg-lists (empty-set)))
   (make-corout id kont mailbox state-env
                prioritize? sleeping? delta-t msg-lists)))
```

FIGURE 5.7 – Structure de donnée représentant une coroutine

5.2.2 Timers

Les timers sont aussi implantés comme des structures *define-type*. Leurs rôle est de fournir une abstraction temporelle pour le déroulement du système de coroutines, qui peut être perçu comme une simulation. Cette classe très simple contient des champs afin de garder compte du temps courant de la simulation, de la période du timer, etc...

Les timers doivent être rafraîchis régulièrement selon une période fixe, ainsi il doivent être exécutés complètement à l'extérieur du système de coroutine pour y arriver et donc, sont implantés en utilisant le système de *threads* de Gambit-C. Il en résulte donc que ces derniers sont rafraîchis régulièrement de manière concurrente avec le système de coroutines.

5.2.3 Ordonnancement

L'ordonnancement est le coeur du système de coroutine. Cet ordonnancement évolue dans un environnement dynamique contenant l'état du système de coroutines actif. Cet environnement dynamique comprend les paramètres :

- `current-corout` : Paramètre contenant la coroutine actuellement exécutée. Lorsque celle-ci termine sont exécution, sa valeur de retour doit être placée dans ce paramètre pour signaler à l'ordonnanceur la terminaison de la coroutine.
- `q` : File d'attente des coroutines prêtes à être exécutée (*ready queue*).
- `timer` : *Timer* utilisé pour la simulation.
- `time-sleep-q` : File prioritaire implantée avec un arbre rouge-noire qui contient les coroutines en attentes sur le temps.
- `root-k` : Continuation primordiale, *i.e.* continuation de du système de coroutine courant.
- `return-value-handler` : Fonction utilisée pour accumuler les résultats de terminaison des coroutines.
- `return-value` : Accumulateur de valeurs de retour des coroutines terminées.

- `parent-state` : Sauvegarde de l'état du système de coroutine *parent* au système actuel, pour des systèmes cascadés. Cet état est décrit par les paramètres présentés ici.

- `return-to-sched` : Continuation de l'ordonnanceur

- `dynamic-handlers` : Liste de patrons dynamiques utilisés avec la forme spéciale `recv`.

- `sleeping-coroutines` : Nombres de coroutines en attentes. Cette variable est nécessaire parce que l'accès aux files `q` et `time-sleep-q` n'est pas suffisante. En effet, les coroutines peuvent être en attente sur des mutex ou sur eux-même (reception d'un message), ainsi ce paramètre permet à l'ordonnanceur de savoir s'il existe toujours au moins une coroutine en attente.

La récursion de système fonctionne grâce à un système de sauvegarde de cet état dans le paramètre `parent-state` dans un sens ou dans le champs `state-env` d'un objet de coroutine dans l'autre.

L'algorithme d'ordonnancement, en lui même, est très simple. Ce dernier est présenté intégralement dans la figure 5.8. Dans un premier temps, la valeur de retour de la coroutine précédente, si cette dernière a terminée, est traitée ou dans le cas échéant, elle est automatiquement remise dans la file d'attente des coroutines prêtes. Par la suite, la file prioritaire de coroutines en attente sur le temps est regardée afin réveiller toutes coroutines ayant dépassé leurs délais de sommeil.

Finalement, la première coroutine disponible dans la file de coroutines prêtes est choisie comme prochaine coroutine et sont travail est poursuivi par un appel à `resume-coroutine`. Si toutefois aucune coroutine ne se trouvait dans la file `q`, alors une vérification des coroutines en attente sur le temps est faite afin de déterminer s'il reste du travail à faire. S'il y a des coroutine en attente sur le temps, alors l'ordonnanceur se met en veille pour le délais d'attente restant. Cette particularité se distingue nettement des simulations à événements discrets qui auraient plutôt incrémentés leur horloge interne directement de ce délais, comportement qui est indésirable pour un jeu vidéo. Par contre, si aucune coroutine n'est prête ou ne dort sur le temps, alors le système ne peut plus rien faire. S'il existe d'autres coroutines en veille (par exemple sur l'attente d'un message), alors le système est en position d'interblocage. Sinon, le travail est terminé et donc, l'état du système parent est rétabli et la continuation primordiale est invoquée.

Le changement de contexte et le retour au contexte d'une coroutine sont fait, à la base, par les fonctions `yield` et `resume-coroutine`, illustrée respectivement dans les figure 5.9 et 5.10. La figure 5.9 illustre aussi la version permettant le changement de contexte du système de coroutine. Ces fonctions illustre la base même du système de coroutine, où la réification de continuations permet la sauvegarde de l'état présent du calcul pour une utilisation future. Comme l'indique la figure 5.10, l'état de ces calculs sont poursuivis simplement par l'appel de ces continuations sauvegardées dans la structure de donnée des coroutine. Il est important de préciser

```
(define (corout-scheduler)
    (manage-return-value)
    (wake-up-sleepers)
    (current-corout (dequeue! (q)))
    (cond
     ((corout? (current-corout))
      (resume-coroutine)
      (corout-scheduler))
     ((not (time-sleep-q-empty? (time-sleep-q)))
      (let* ((next-wake-time
              (time-sleep-q-el-wake-time
               (time-sleep-q-peek? (time-sleep-q)))))
        (thread-sleep! (/ (- next-wake-time (current-sim-time))
                          (timer-time-multiplier (timer)))))
      (current-corout ___scheduler-is-speeping___)
      (corout-scheduler))
     (else
      (let ((finish-scheduling (root-k))
            (ret-val (return-value)))
        (if (> (sleeping-coroutines) 0)
            (error "Deadlock detected in coroutine system..."))
        (restore-state (parent-state))
        (continuation-return finish-scheduling ret-val)))))
```

FIGURE 5.8 – Algorithme d'ordonnancement

que l'ordre dans lequel les états des systèmes de continuations (pour des systèmes récursifs) sont sauvegardés et restaurés est critique afin du bon fonctionnement de ces appels. Par exemple, lors d'un appel à `super-yield`, l'état du système actuel doit absolument être sauvegardé avant de restauré l'état du système parent pour éviter d'être perdu.

```
(define (yield)
  (continuation-capture
   (lambda (k)
     (corout-kont-set! (current-corout) k)
     (resume-scheduling))))
(define (super-yield)
  (continuation-capture
   (lambda (k)
     (if (parent-state)
         (let ((state (save-state)))
           (restore-state (parent-state))
           (corout-state-env-set! (current-corout) state)
           (corout-kont-set! (current-corout) k)
           (resume-scheduling))))))
```

FIGURE 5.9 – Fonctions de changement de contexte explicites

5.2.4 Système de messagerie

La fonctionnalité de messagerie du système de coroutines est à la base très simple. Comme l'illustre la figure 5.7, chaque objet coroutine possède une boîte de réception de messages. Cette boîte est implantée simplement comme une file d'attente, de manière similaire à la file d'attente des coroutines prêtes utilisée par l'ordonnanceur.

L'envoi de message n'est qu'alors l'ajout d'un nouveau message dans cette file d'attente. Après cet ajout, le système vérifie si la coroutine réceptrice était en attente d'un message et si c'est le cas, alors cette dernière est remise en action dans l'ordonnanceur. La figure 5.11 illustre ce procédé d'acheminement de message. Puisqu'il est possible de spécifier une valeur d'attente maximale de messages,

```
(define (resume-coroutine)
  (continuation-capture
   (lambda (k)
     (return-to-sched k)
     (let ((kontinuation (corout-kont (current-corout))))
       (if (corout-state-env (current-corout))
           (let ((state (save-state)))
             (restore-state (corout-state-env (current-corout)))
             (parent-state state)))
       (continuation-return kontinuation 'go)))))
```

FIGURE 5.10 – Implantation du retour au contexte d'une coroutine

la coroutine en attente pourrait se retrouver dans la file d'attente sur le temps des coroutines. Afin de pouvoir distinguer une telle attente bornée d'un appel à `sleep-for`, l'état `interruptible?` est utilisé.

```
(define (! dest-corout msg)
  (enqueue! (corout-mailbox dest-corout) msg)
  (cond ((sleeping-on-msg? dest-corout)
         (corout-set-sleeping-mode! dest-corout #f)
         (corout-enqueue! (q) dest-corout))
        ((and (sleeping-over-time? dest-corout)
              (interruptible? dest-corout))
         (time-sleep-q-remove! (sleeping-over-time?->node dest-corout))
         (corout-set-sleeping-mode! dest-corout #f)
         (corout-enqueue! (q) dest-corout))))
```

FIGURE 5.11 – Envoi de messages entre coroutines

La réception de messages se marie bien avec l'envoi. La coroutine vérifie si un message est disponible et si c'est le cas retourne le retourne. Sinon, alors deux cas sont possibles : soit la coroutine attendra jusqu'à la réception d'un nouveau message, soit cette attente sera bornée dans le temps. Pour une attente bornée, la

coroutine est mise en veille pour la durée spécifiée, en spécifiant qu'elle peut être interrompue par la réception d'un message. Sinon, alors l'état de la coroutine est conservée et elle est considérée comme « dormante en attente de message ». Si le délais d'attente est dépassé, alors une exception est lancée à l'utilisateur. La figure 5.12 illustre l'implantation de la fonction `?`, la plus simple pour la réception de message. Elle permet par contre de donner une bonne idée du procédé employé.

```
(define (? #!key (timeout 'infinity))
  (define mailbox (corout-mailbox (current-corout)))
  (if (empty-queue? mailbox)
      (if (number? timeout)
          (sleep-for timeout interruptible?: #t)
          (continuation-capture
           (lambda (k)
             (let ((corout (current-corout)))
               (corout-kont-set! corout k)
               (corout-set-sleeping-mode! corout (sleeping-on-msg))
               (resume-scheduling))))))
  (if (empty-queue? mailbox)
      (raise mailbox-timeout-exception)
      (dequeue! mailbox)))
```

FIGURE 5.12 – Réception de messages avec la fonction `?`

L'implantation de la forme spéciale de réception de messages `recv` est complexe et ne sera pas détaillée. Par contre, un exemple d'expansion de cette macro est donné dans la figure 5.13. On constate que le patron donné est vérifié en premier lieu. Par la suite, si aucun messages ne correspondent à ce patron, une vérification est faite parmi les patrons dynamique afin de trouver un message pouvant être utilisé. Si aucun message n'est trouvé, alors la coroutine est mise en veuille et

recommencera la vérification après avoir reçu un nouveau message.

```
(let ((#:mailbox3058 (corout-mailbox (current-corout))))
  (let #:loop3057 ()
    (cond ((queue-find-and-remove!
            (lambda (#:msg3059) (match #:msg3059 (#(allo ,x) #t) (,_ #f)))
            #:mailbox3058)
           =>
           (lambda (#:msg3060) (match #:msg3060 (#(allo ,x) x) (,_ #f))))
          ((find-value (lambda (pred) (pred)) (dynamic-handlers))
           =>
           (lambda (res) (unbox res) (#:loop3057)))
          (else
           (begin
             (continuation-capture
              (lambda (k)
                (let ((corout (current-corout)))
                  (corout-kont-set! corout k)
                  (corout-set-sleeping-mode! corout (sleeping-on-msg))
                  (resume-scheduling))))
             (#:loop3057))))))
```

FIGURE 5.13 – Expansion macro de `(recv (#(allo ,x) x))`

5.3 Conclusion

Ainsi, un système de coroutine a été implanté de manière à pouvoir fournir aux programmeurs de jeux vidéo une interface à un système permettant de pouvoir facilement exprimer des problèmes de changements de contextes (notamment dans le cadre de jeux multi-joueurs), sans avoir à se soucier de synchroniser les coroutines dans le but d'éviter les problèmes de sections critiques.

Ce système offre ainsi une interface similaire à celle offerte par le système Ter-

mite [43], orientée sur un calcul en série au lieu de distribué. Il est ainsi possible de concevoir des coroutines comme des entités évoluant dans un même environnement de manière successive (comme s'est souvent le cas dans un jeu vidéo).

Le système a également été généralisé de manière à permettre une utilisation en cascade de systèmes de coroutines. Ainsi, l'utilisation du système ne se limite pas à un usage monolithique, mais permet de séparer de tâches en sous-systèmes de coroutines.

L'utilisation de synchronisation de coroutines par envoi et réception de messages est très naturelle et donc facile à utiliser. Lorsqu'elle est combinée avec des listes de diffusion de message, on obtient un style de programmation se rapprochant beaucoup des système de programmation réactive [44].

Il serait maintenant intéressant d'ajouter un mécanisme de profilage de coroutine au système développé. Un tel mécanisme permettrait d'avoir, entre autre, une meilleur idée du temps moyen que prend une coroutine donnée avant de céder le contrôle dans le but de mieux balancer ou d'optimiser ces dernières.

[TODO: *Ajouter en annexe le code ?*]

CHAPITRE 6

ÉVALUATION ET EXPÉRIENCES

Afin de pouvoir déterminer les forces et les faiblesses du développement de jeux vidéo en Scheme, des jeux doivent être développés en augmentant graduellement la complexité de ceux-ci pour permettre de trouver et résoudre de manière itérative les problèmes reliés à leur développement. Un jeu simple comprend les problèmes les plus fondamentaux qu'un jeu puisse avoir : détection de collisions, animations, concept de niveaux, etc... Ainsi, ces problèmes peuvent être adressés dans un premier temps, puis de nouveaux problèmes peuvent être entrepris par la suite en développant un jeu plus complexe. Aussi, cette approche permet de bâtir une infrastructure de développement de jeux vidéo qui réduit la difficulté du développement de jeux plus complexes.

Le jeu choisi pour la première phase de développement est *Space Invaders* (1978). Ce dernier date de la période des jeux d'arcades et, tout en étant très simple, adresse plusieurs problèmes fondamentaux qu'impliquent les jeux vidéo modernes : interactions avec usager rapide, des niveaux, de la détection de collision et même des parties multi-joueurs. Puisque le graphisme de ce jeux est très rudimentaire, il consiste en un très bon choix pour un premier jeu, car il permet de concentrer le développement sur le moteur de celui-ci. Le développement de ce jeu est traité dans la section 6.1. Le développement de ce dernier a permis de faire face à plusieurs

problèmes reliés au développement de jeu dont le rendu graphique, la détection et résolution de collision et sur le flot de contrôle du jeu. Son développement a permis la création du système orienté objet décrit dans le chapitre 4 et du système de coroutine présenté dans le chapitre 5.

Par la suite, le jeu *Lode Runner* (1983) a été développé. Ce dernier date de l'époque des premiers ordinateurs personnels et est plus élaboré. Par contre, c'est la version arcade du jeu (1984) qui a été reprise. Tout en conservant les problèmes fondamentaux traités dans *Space Invaders* , ce jeu étend le concept de niveaux déjà traités, introduit la notion d'intelligence artificielle et possède des entités possédant des états plus complexes. De plus, le développement de *Lode Runner* a pour but de consolider les techniques développées pour la création du premier jeu. Le développement de ce jeu est traité dans la section 6.2. Les nouveaux problèmes rencontrés furent principalement reliés à une gestion des états et du rendu plus complexe des objets et à de la plus grande quantité d'objets présent dans le jeu. Les outils et techniques déjà développés ont toute fois permis d'aisément résoudre ces derniers.

Ainsi, ce chapitre a pour but d'expliquer le cheminement du travail nécessaire à l'écriture de ces deux jeux afin de pouvoir répondre à la problématique de base traité par ce document.

6.1 Développement de *Space Invaders*

Ce jeu consiste à « sauver la galaxie »en éliminant une armée d'envahisseurs extra-terrestre grâce à un vaisseau spatial équipé de lasers et bénéficiant de la présence de trois boucliers. Le joueur ne peut se déplacer que de gauche à droite et tirer des lasers. Les ennemis sont placés en formations et se déplacent aussi latéralement jusqu'à ce qu'ils frappent un côté de l'écran. Ils descendent alors d'une rangée, se rapprochant ainsi du joueur. La vitesse de déplacement des ennemis est inversement proportionnelle à leurs nombre. Le niveau se termine lorsque tous les ennemis sont détruits. Alors, le prochain niveau (qui est exactement le même que le précédant) débute. La figure 6.1 illustre une capture d'écran du jeu développé.

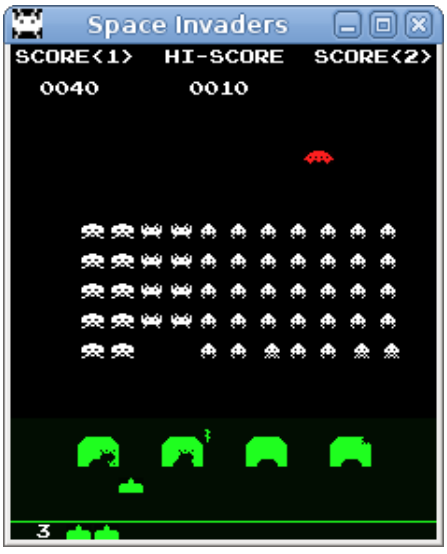

FIGURE 6.1 – Capture d'écran du jeu *Space Invaders*

L'objectif visé en développant *Space Invaders* est simple : écrire un premier jeu qui permet d'exposer les problèmes fondamentaux reliés au développement de jeux

vidéo et résoudre ces problème en tentant de tirer profit de la puissance expressive du langage Scheme. Pour y arriver, une approche de développement itératif (en spirale) a été utilisée.

Une première version a été développée dans le but d'avoir l'infrastructure de base du jeu et d'identifier les problèmes rencontrés au cours du développement. Le développement de cette première version est décrit dans la section 6.1.1.

Par la suite, une deuxième version fut écrite afin de répondre à certaines lacunes que présentait la première version du jeu. Pour ce faire, le développement du système objet présenté dans le chapitre 4 fut entrepris. Cette démarche d'amélioration du jeu est décrite dans la section 6.1.2.

Finalement, une version expérimentale fut développée. Cette version tentait de s'éloigner de l'architecture classique de jeu vidéo où une boucle principale (*main loop*) gère l'état du système en entier. L'idée était de tenter de fusionner le système objet et le système de coroutine afin que tous les objets du jeu soient eux-mêmes des coroutines, en espérant ainsi simplifier l'écriture d'entités du jeu. En expérimentant ainsi sur le contrôle de flot du jeu, il est possible de démontrer si l'utilisation de Scheme permet de faciliter de telles expérimentations. Cette dernière version du jeu est décrite dans la section 6.1.3

6.1.1 Version initiale

Dans un premier temps, l'analyse de la version arcade du jeu disponible sur l'émulateur MAME [45]. Ce dernier est un système émulant plusieurs architectures de machines arcades et permet de pouvoir jouer sur PC aux versions originales de ces jeux. Plusieurs problèmes potentiels ont alors été identifiés :

- Rendu graphique du jeu
- Flot du contrôle logique du jeu
- Animations
- Structures de données
- Détection et résolution de collisions
- Parties multi-joueurs

6.1.1.1 Rendu graphique

Afin de pouvoir effectuer le rendu graphique du jeu, une librairie de lecture et chargement d'images a dû être écrite. Plutôt que d'utiliser une librairie C déjà existante pour faire ce travail, le choix de l'écriture de cette librairie visait à limiter les dépendances externes et d'avoir autant que possible de code en Scheme. Par contre, les librairie SDL [46] et *OpenGL* [47] ont été utilisées afin de pouvoir simplifier la gestion d'entrées/sorties, du fenêtrage et du rendu de l'application.

L'utilisation des interfaces aux fonctions étrangères (*Foreign Function Interface*) du système Gambit-C ont fait en sorte que la tâche d'accès et d'utilisation de ces librairies fut triviale.

Cette librairie de lecture et de chargement d'images s'occupe de lire des images en format textuel *ppm*, qui est l'un des formats les plus simples pour décrire une image *bitmap*. Le système s'attend à lire des images sous forme de « fontes », *i.e.* des images contenant plusieurs sous-images. Ce choix semble raisonnable car une entité de jeu vidéo est souvent représentée par plusieurs images. Ceci permet donc de garder ces dernières dans une même image. Un fichier descriptif de la fonte, sous forme de S-expressions, doit être présent afin de permettre une interprétation correcte de la fonte lue. Par exemple, l'image fournie dans la figure 6.2 doit être accompagnée d'un fichier de même nom, portant une extension *.scm* conteant la description `((colors: (white green red)) (chars: (0 1)))`.

Cette fonte peut alors être lue et chargée en mémoire vidéo grâce à la forme spéciale :

```
(define-uniform-font <font-name> <width> <height>
                     [static] [loop-x] [loop-y])
```

Cette dernière effectue la génération de code C permettant la lecture et le chargement de la fonte spécifiée, à condition que celle-ci soit uniforme, *i.e.* que toutes les sous images soient de même taille. L'option `static` permet d'effectuer la lecture durant l'expansion macro et d'inclure l'image dans le code source généré. Les

FIGURE 6.2 – Exemple de fontes utilisée dans *Space Invaders* .

options `loop-x` et `loop-y` permettent de spécifier le comportement à entreprendre lorsque l'image est agrandie. Par défaut, l'image est lue dynamiquement et est étirée lorsqu'elle est agrandie, mais elle peut être lue dynamiquement et/ou répétée lorsque agrandie dans la direction de l'axe des X ou des Y.

Cette librairie est très simple puisqu'elle ne traite que des fontes aux sous-images de taille uniformes. Ce choix fut effectué puisqu'il était bien adapté aux jeux développés et dans le but de ne pas trop complexifier la tâche de la création des jeux.

6.1.1.2 Logique du jeu

Le flot de contrôle de la logique de ce jeu fut expérimental dès le départ. Dans la première version développée, une simulation par événements discrets a été utilisée.

L'idée derrière ce choix provenait du fait que dans un jeu vidéo, la progression peut être considérée comme une série d'événements discrets. Par exemple, le laser avance, le joueur se déplace à droite, le vaisseau ennemi explose, etc... Ainsi, un petit système de simulation par événements discrets a été développé et utilisé. Ce dernier fut implanté de manière très simple en utilisant un monceau triant les événements ordonnancés de manière croissante en fonction du temps de leurs arrivé dans la simulation. Ces évènements étaient des *thunk*, fermetures Scheme sans arguments, effectuant le travail à accomplir par l'évènement. Lorsqu'un évènement se termine, l'ordonnanceur choisi alors le prochain et appelle le corps de celui-ci. Il est également possible pour un évènement d'ordonnancer un autre évènement grâce à la forme spéciale `(in <secs> <thunk>)` qui enregistre cette nouvelle fermeture dans le système de manière à ce qu'elle se fasse appelée dans /scheme¡secs¿ secondes. Contrairement aux simulation à évènements discrets habituelles, les attentes du systèmes entre les évènements sont effectuées par l'ordonnanceur de manière à respecter les délais demandés par les évènements.

La figure 6.3 illustre des événements qui sont ordonnancés au tout début d'une partie. Ainsi le premier événement se chargera de faire un *dispatch* d'autres événements débutant la partie. Les deux derniers événements sont des événements de gestion d'affichage et d'entrées/sorties du jeu.

Ces événements doivent donc s'occuper de poursuivre le calcul du jeu leur étant assigné en réordonnant de nouveaux événements ultérieurement dans le jeu. Par

```
(schedule-event! sim 0
  (lambda () (new-player! level)
             (in 0 (create-init-invader-move-event level))
             (in 1 (create-invader-laser-event level))
             (in (mothership-random-delay)
                 (create-new-mothership-event level))))

(schedule-event! sim 0 (create-main-manager-event level))
(schedule-event! sim 0 (create-redraw-event level))
```

FIGURE 6.3 – Exemple de code de simulation par événements discrets

exemple, la figure 6.4 illustre l'évènement principal implantant le comportement désiré pour le vaisseau mère des ennemis.

```
(define (create-mothership-event level)
 (define mothership-event
   (synchronized-event-thunk level
     (let ((mothership (level-mothership level)))
       (if mothership
           (let ((collision-occured? (move-object! level mothership)))
             (if (or (not collision-occured?)
                     (is-explosion? collision-occured?)
                     (eq? collision-occured? 'message))
                 (in mothership-update-interval mothership-event)))))))
  mothership-event)
```

FIGURE 6.4 – Évènement représentant le vaisseau ennemie de type *mothership*.

Cet exemple illustre bien deux désavantages provenant de l'utilisation d'évènements discrets pour implanter le comportement d'entités dans un jeu. Le premier désavantage est que la plupart des évènements sont récursifs, et donc ils se réordonnancent par eux-mêmes un peu plus tard dans le temps. Ceci résulte en une écriture moins intuitive de ce comportement. Aussi, lorsqu'un évènement est déjà prévu, il est

possible que l'état du jeu change entre temps rendant cet évènement désuet. Pour le vaisseau mère, il est possible que son prochain déplacement ait été déjà prévu, mais que ce dernier a explosé suite à la collision avec un laser du joueur. De tels événements sont donc prompts à donner des erreurs dues à des inconsistances entre l'état du jeu attendu et l'état réel lorsque se produit un événement.

Aussi, il est important de faire bien attention à doser le calcul effectué par un événement de manière à ne pas monopoliser le temps du processeur utilisé. Ainsi, les évènements sont souvent réordonnancés avec un intervalle de temps de zéro, de manière à laisser la chance aux autres évènements de s'exécuter, tout en forçant l'évènement courant de s'exécuter le plus tôt possible.

Toutefois, cette approche semble avoir été très bénéfique pour la création d'animations. En effet, en combinant des événements discrets avec un style d'écriture en CPS, il est possible de bien modulariser des animations dans le jeu. Par exemple, l'animation du début de jeu peut être écrite sous forme CPS de manière à ce que la suite de cette animation puisse être n'importe quel autre événement. La figure 6.5 illustre une partie de l'implantation de l'animation de début de partie. Celle-ci reçoit une continuation en paramètre. Elle termine son animation en utilisant une animation CPS s'occupant de faire clignoter le score du joueur. Puisqu'il s'agit de la fin de l'animation de début de partie, la continuation est passée directement en paramètre à cette animation de clignotement (optimisation d'appel terminal). On a pu ainsi facilement utiliser l'animation de clignotement, et ce de manière

très modulaire, dans notre animation de début de partie. Cette figure illustre un exemple d'utilisation de cette animation en réécrivant de manière plus complète l'événement de début de partie de la figure 6.3.

```
(define (start-of-game-animation-event level continuation)
  (lambda ()
    ...
    (in 0 (create-text-flash-animation-event level
            (level-get level score-msg-obj)
            animation-duration new-cont))))
(schedule-event! sim 0
  (start-of-game-animation-event level
    (generate-invaders-event level
      (lambda ()
        (new-player! level)
        (in 0 (create-init-invader-move-event level))
        (in 1 (create-invader-laser-event level))
        (in (mothership-random-delay)
            (create-new-mothership-event level))))))
```

FIGURE 6.5 – Exemple d'animation combinant les événements discrets et une programmation CPS

6.1.1.3 Structures de donnés

Les structures de données utilisées dans *Space Invaders* furent définies par la forme spéciale `define-type` de Gambit-C. Elles ont utilisé l'héritage simple que permet cette forme de manière à tirer profit du maximum de polymorphisme et de modularisation de code possible. La figure 6.6 donne certaines définitions des structures employées.

L'avantage principal de l'utilisation de ces structures est qu'elles sont très per-

```
(define-type game-object id type pos state color speed
  extender: define-type-of-game-object)
(define-type-of-game-object invader-ship row col)
(define-type-of-game-object player-ship)
(define-type-of-game-object message-obj text)
...
```

FIGURE 6.6 – Structures de données utilisées dans la première version de *Space Invaders*

formantes (voir la table 4.3). Par contre, elles ne permettent pas d'avoir des champs communs à toutes les instances d'un type donné. Il en résulte que l'expression du fait que toutes les instances du type `player-ship` doivent être associés à une boîte englobante (*bounding box*) de 13 par 8 pixels est difficile à exprimer. Ces informations pourraient être ajoutées dans chaque instances, mais ce serait un gaspillage important d'espace mémoire puisque ces données sont communes à toutes les instances de ce type. Ainsi, pour cette version de *Space Invaders*, un système manuel de type *très rudimentaire* a été utilisé. Une structure de donnée décrivant un type est créée. Cette dernière contient les données communes aux instances de ce type. Un pointeur vers le bon type est par la suite ajouté dans le champs `game-object-type` de chacune des instances du jeu. L'utilisation de ce petit système de type demande ainsi une écriture de code fastidieuse qui ne devrait pas être nécessaire.

6.1.1.4 Détection et résolution de collisions

La détection de collisions sur ces objets a pu être faite de manière efficace et modulaire grâce aux types qui contiennent de l'information sur les boîtes englobantes des objets. Puisqu'il n'y a jamais beaucoup d'objets présents dans le jeu en même temps, une détection très rudimentaire de collisions fut utilisée. Cette dernière est présentée dans la figure 6.7.

```
(define (detect-collision? obj level)
  (or (exists (lambda (collision-obj)
                (if (shield? collision-obj)
                    (obj-shield-collision? obj collision-obj)
                    (obj-obj-collision? obj collision-obj)))
              (level-all-objects level))
      (obj-wall-collision? obj (game-level-walls level))))
```

FIGURE 6.7 – Détection de collision dans *Space Invaders*

Par contre, Un problème majeur relié à l'utilisation de ces structures de données est apparu lorsque la *résolution de collisions* a due être implantée. En effet, cette résolution dépend du types des deux objets entrant mutuellement en collision. La figure 6.8 illustre bien le problème rencontré. On constate que les types des objets doivent être manuellement analysés de manière à utiliser la bonne fonction de résolution qui, elle aussi, se doit d'analyser le type de l'objet reçu de manière a choisir la bonne résolution à adopter. Il en résulte en du code très lourd, où il est très facile d'introduire des erreurs.

```
(define (resolve-collision! level obj coll-obj)
  (cond
   ((player-ship? obj) (resolve-player-collision! level obj coll-obj))
   ((laser-obj? obj) (resolve-laser-collision! level obj coll-obj))
   ...))
(define (resolve-laser-collision! level laser-obj collision-obj)
  (cond ((invader-ship? collision-obj) ...)
        ((laser-obj? collision-obj) ...)
        ((player-ship? collision-obj) ...)
        ...))
```

FIGURE 6.8 – Résolution de collisions dans la première version de *Space Invaders*

6.1.1.5 Parties multi-joueurs

Les parties multi-joueurs de *Space Invaders* sont très simples. Chaque joueur joue en alternance jusqu'à se que son vaisseau se fasse détruire. Ainsi, cela implique de pouvoir conserver deux parties du jeu en *parallèle*, où une seule des deux avance à la fois. Avec un flot de contrôle sous forme de simulation par événements discrets, cela implique d'avoir deux simulations exécutées en parallèle. Puisque le système de simulation ne permet pas de pouvoir facilement arrêter une simulation et d'en avoir une deuxième se produisant en même temps, une autre approche a été utilisée. Un système de coroutines dans lequel les simulations sont exécutées à l'intérieur de coroutines différentes a été développé et utilisé. Ainsi, le changement de contexte peut être manuellement utilisé lorsqu'un joueur meurt afin de réactiver la partie du prochain joueur.

C'est de cette idée qu'est né le système de coroutine présenté dans le chapitre 5. Il fut initialement beaucoup plus simple que celui présenté dans ce chapitre, mais les

composantes de base restent les mêmes. La figure 6.9 résume la procédure utilisée lors de la mort d'un joueur. Après avoir effectué la gestion des objets dans le niveau, un mutex est bloqué afin de permettre l'arrêt des animations du jeu, chose requise pour l'animation de mort d'un joueur. Ce mutex provient du système d'événement discret qui fut, par la suite, intégré au système de coroutine. Par la suite, l'animation de mort d'un joueur est lancée, avec comme continuation `continuation`. Cette dernière, dans le cas où le joueur n'a pas terminé sa partie, s'occupe d'effectuer le changement de coroutines, après avoir informé l'autre coroutine de certaines informations au préalable. Il est intéressant de noter que juste avant le changement de contexte, un événement très prioritaire doit être ordonnancé de manière à se produire immédiatement au retour du contexte de la coroutine. Il s'agit de l'animation ré-introduisant la partie du joueur actuel.

6.1.1.6 Conclusion

Ainsi, l'écriture de cette première version de *Space Invaders* a permis d'identifier plusieurs problèmes reliés au développement de jeux vidéo (détection et résolution de collisions, parties multi-joueurs, etc...). En plus de permettre de pouvoir facilement expérimenter sur le type de flot de contrôle utilisé pour développer ce jeu, Scheme a permis de faire rapidement plusieurs outils non triviaux pour l'implantation du jeu, dont un système de fontes et un petit système de coroutines.

Par contre, certains problèmes ne possèdent pas des solutions satisfaisantes. En

```
(define (explode-player! level player)
  (level-loose-1-life! level)
  (level-add-object! level player-expl-obj)
  (level-remove-object! level player)
  (let ((continuation
         (if (<= (game-level-lives level) 0)
             (begin ...)
             (lambda ()
               (if (2p-game-level? level)
                   (begin
                     (send-update-msg-to-other level #f)
                     (in NOW! (start-of-game-animation-event
                               level (return-to-player-event level)))
                     (yield-corout))
                   (begin ...))))))
    (in 0 (lambda ()
            (sem-lock! (level-mutex level))
            (in 0 (player-explosion-animation-event
                   level expl-obj animation-duration continuation))))))
```

FIGURE 6.9 – Événement de mort d'un joueur

effet, la résolution de collision est très fastidieuse et aurait besoin d'être modularisée et améliorée. Aussi, la technique de flot de contrôle expérimentale utilisée (simulation par événements discrets), fonctionne bien pour certains aspects, mais ne semble pas naturelle pour d'autres. Ainsi, le flot de contrôle pourrait aussi être amélioré de manière à pouvoir exprimer de manière plus naturelle le comportement des entités du jeu.

6.1.2 Version orientée objet

Une nouvelle version de *Space Invaders* fut développée afin de pallier à une lacune existante dans la première version écrite : le problème relié aux types de structures de données qui engendrait des cascades fastidieuses de vérifications de types pour, entre autre, la résolution de collisions (voir la figure 6.8).

Initialement, plusieurs systèmes objets existant pour Gambit-C furent mis à l'essai pour pallier à ce problème, mais aucun ne répondait bien aux besoins présents. Certains offraient beaucoup de puissance expressive, mais avec un coût de performance trop élevé tandis que d'autre, plus performants, étaient trop restrictifs dans l'interface offerte. Pour ces raisons, un système de programmation orientée objet fut développé de manière à pouvoir bien répondre aux besoins dans *Space Invaders*, sans apporter un coût d'utilisation trop élevé. Ce système fait l'objet du chapitre 4.

Ainsi, les objets sont maintenant déclarés comme des classes appartenant à une hiérarchie relativement simple et tirant profit de la possibilité de concevoir des hiérarchies multiples afin d'apposer des propriétés aux objets du jeux. La figure 6.10 illustre un échantillon de la hiérarchie de classes utilisée dans le jeu. On constate que les champs communs à toutes les instances ont été ajoutés comme membres de classe. Une classe abstraite, `sprite-obj`, a aussi été ajoutée afin de donner la propriété de *sprite* aux objets dont le rendu dépend d'images dans une fonte. Par la suite, la classe `invader-ship` devient enfant de ces deux classes, tout en ajoutant

les deux champs communs à tous les types d'*invader*. Cette dernière est sous-classée par les trois types d'ennemis existant (à l'exception du vaisseau mère qui est traité à part). Puisque chaque type d'*invader* possède sa propre classe, on peut alors spécifier les valeurs des champs de classes pour chacune de celles-ci de manière simple et claire. L'utilisation de ces champs pourra être, par la suite, faite de manière complètement uniforme et transparente. La macro `setup-static-fields` a été écrite afin de permettre de facilement assigner les champs de classes pour les classes enfants de `game-object`.

```
(define-class game-object ()
  (slot: id) (slot: pos) (slot: state) (slot: color) (slot: speed)
  (class-slot: sprite-id) (class-slot: bbox)
  (class-slot: state-num) (class-slot: score-value))
(define-class sprite-obj ())
(define-class invader-ship (game-object sprite-obj) (slot: row) (slot: col))
(define-class easy-invader   (invader-ship))
(define-class medium-invader (invader-ship))
(define-class hard-invader   (invader-ship))
(setup-static-fields! easy-invader 'easy (make-rect 0 0 12 8) 2 10)
(setup-static-fields! medium-invader 'medium (make-rect 0 0 12 8) 2 20)
(setup-static-fields! hard-invader 'hard (make-rect 0 0 12 8) 2 30)
```

FIGURE 6.10 – Aperçu de la hiérarchie de classe du jeu

L'attrait principal du développement et de l'intégration du système objet pour la création du jeu réside dans l'utilisation des fonctions génériques. L'utilisation de celles-ci permet d'utiliser des opérations spécifiques aux objets de manière complètement transparente, rendant ainsi l'écriture de ces parties du code plus élégante et solides face aux changements éventuels des structures de donnés ou de

l'architecture du jeu.

Trois fonctions génériques ont été utilisées pour améliorer *Space Invaders* : `detect-collision?`, `resolve-collision!` et `render` qui effectuent respectivement la détection de collisions, la résolution de collisions et le rendu graphique des objets. La figure 6.11 donne un aperçu de l'utilisation faite de la fonction générique `resolve-collision!`. Des instances de cette dernière sont définies pour les paires de collisions possibles dans le jeu et le système s'occupe de faire automatiquement le choix dynamique de la bonne instance en fonction des objets passés en paramètre. Le code résultant est beaucoup plus modulaire et clair que celui de la version précédente du jeu.

La deuxième version du jeu apporte beaucoup d'améliorations face au développement de jeu vidéo en Scheme grâce à l'utilisation de la programmation orientée objet. La modularité et l'abstraction du code de la première version a été améliorée de grâce à l'utilisation d'une hiérarchie de classes annotée de propriétés et comportant des membres de classe et des fonctions génériques pour effectuer de manière transparente les actions de base sur les instances de ces classes. Par contre, un problème n'a toujours pas été adressé dans cette nouvelle version. En effet, le flot de contrôle implanté par une simulation à événement discret n'a pas été modifié afin de mieux l'adapter aux comportements des entités du jeu.

```
(define-method (resolve-collision! level (laser laser-obj) (inv invader-ship))
  (invader-ship? inv)
  (level-increase-score! level inv)
  (destroy-laser! level laser)
  (explode-invader! level inv))

(define-method (resolve-collision! level (laser1 laser-obj) (laser2 laser-obj))
  (let ((inv-laser (if (player_laser? laser1) laser2 laser1)))
    (explode-laser! level inv-laser)
    (destroy-laser! level inv-laser)))
...
(define (move-object! level obj)
  (move-object-raw! obj)
  (let ((collision-obj (detect-collision? obj level)))
    (if collision-obj
        (begin (resolve-collision! level obj collision-obj)
               collision-obj)
        #f)))
```

FIGURE 6.11 – Exemple de définition et d'utilisation d'une fonction générique dans *Space Invaders*

6.1.3 Version avec flot de contrôle sous forme de coroutines

Ainsi, après deux version du jeu, *Space Invaders* possède toujours certains problèmes dont le flot de contrôle utilisé qui n'est très bien adapté aux objets du jeu. Afin d'adresser ce problème, le système de coroutines utilisé pour l'implatation de parties multi-joueurs a été étendu de manière à ce que le flot de contrôle des entités du jeu soit exprimé plutôt sous forme de *coroutines*. Ce système est présenté dans le chapitre 5. Cette approche sur le contrôle de flot est aussi expérimentale, mais l'utilisation d'une simulation à événements discrets a laissée croire que la description du comportent des entités sous forme de *threads* serait plus naturelle. En

effet, en utilisant des événements discrets, le corps des événements devait faire en sorte que l'événement se poursuive en se réordonnançant un peu plus tard, donnant ainsi la chance aux autres événements de pouvoir s'exécuter. Ce phénomène correspond très bien à la fonction `yield` du système de coroutine qui interrompt de manière temporaire l'exécution d'une coroutine afin de laisser la chance aux autres coroutines de pouvoir continuer leur travail.

Ainsi, dans cette nouvelle version du jeu, *tous* les objets sont maintenant des coroutines exécutant leurs comportement de manière indépendante. Pour ce, une version orientée objet du système de coroutine fut produite. La figure 6.12 illustre l'intégration du système de coroutine aux objets du jeu. L'utilisation de constructeurs permet l'appel du constructeur de base des coroutines afin de spécifier le comportement que devra entreprendre l'objet. Ce corps est déterminé de manière dynamique avec la fonction générique `behaviour` et sera exécuté dans un environnement contenant des gestionnaires de messages dynamiques, communs à tous les objets, permettant de pauser le jeu ou de détruire ces derniers.

Malgré l'idéologie prometteuse d'avoir des objets indépendant régis chacun par leurs propres comportements, l'implantation du comportement des entités du jeu est devenu très rapidement *cauchemardesque*. En effet, puisque toutes les entités possèdent leurs fils d'exécution et que ceux-ci sont tous équivalents, la *synchronisation* entre ces dernière est primordiale et implique des protocoles de communication complexes entre les objets. Ces protocoles doivent régir tout le comportent du jeu

```
(define-class game-object (corout)
  (slot: pos)
  (slot: state)
  (slot: color)
  (slot: speed)
  (class-slot: sprite-id)
  (class-slot: bbox)
  (class-slot: state-num)
  (class-slot: score-value)
  (constructor: (lambda (obj id pos state color speed level)
                  (init! cast: '(corout * *) obj id
                         (lambda ()(with-dynamic-handlers
                                     ((pause (pause obj level))
                                      (die   (die   obj level)))
                                     ((behaviour obj level)))))
                  (set-fields! obj game-object
                    ((pos pos)     (state state)
                     (color color) (speed speed))))))
```

FIGURE 6.12 – Classe de base des objets en tant que coroutine

et, en bout de ligne, le flot de contrôle devient soudainement très difficile à exprimer. Ainsi, des méta-objets ont dû être introduits afin de pouvoir effectuer la synchronisation entre les objets de base, comme les ennemis. La figure 6.24 donne le comportement du méta-objet régissant l'activité d'une rangée d'*invader*. Ce dernier utilise des listes de diffusions pour envoyer des messages aux *invader* de la rangée qu'il coordonne. Il possède deux états : l'état initial qui attend la réception d'un message lui indiquant que c'est le tour de sa rangée de se déplacer et un deuxième état consistant à une barrière de synchronisation effectuant l'attente de la réception du message `moved` de la part de tous les *invader* de la rangée. Puis, il vérifie si une collision s'est produite avec un mur durant se déplacement et effectue la gestion de

cette collision si c'est le cas en avertissant les autres contrôleurs de la situation.

De plus, l'utilisation intensive (voir abusive) du système de coroutine couplé avec le système d'objets a résulté en une chute dramatique des performances du jeu. Cette chute de performance a permis d'améliorer le système de coroutine en implantant un système de profilage des coroutine afin de pouvoir visualiser quelles coroutines monopolisaient le contrôle de l'ordonnanceur pour une durée trop longue. Il était ainsi possible d'optimiser certaines coroutines afin d'améliorer les performances globales du système. Toutefois, même après effectuer ces optimisations, les performances du jeu demeuraient très mauvaises et donc, la troisième itération de *Space Invaders* fut abandonné en cours de route. La majeur partie de la logique du jeu fut implanté sous forme de coroutines. Cette version a permis de réaliser que l'approche plus tradionnelle pour le flot de contrôle serait plus appropriée que les approches utilisées pour implanter *Space Invaders* . Par contre, cette expérimentation sur la technique de flot de contrôle a permis de démontrer la facilité avec laquelle il a été possible de modifier le coeur du jeu pour essayer de nouvelles options quant à l'architecture du programme et de son flot de contrôle.

6.1.4 Conclusion

Ainsi, le développement de *Space Invaders* en Scheme fut très riche en expériences. Il a permis de trouver plusieurs problèmes reliés au développement de jeux vidéo. Ces derniers ont été résolus en utilisant le mieux possible les avantages qu'offrent un

langage de haut niveau tel que Scheme en utilisant les fonctions d'ordres supérieures, un style de programmation CPS et en développant un système de coroutines.

La création d'un système de programmation orienté objet a grandement permis d'améliorer la qualité du code écrit lors du développement de la première version du jeu grâce à l'utilisation des fonctions génériques permettant de faire un choix dynamique de procédures en fonctions des objets passés en paramètres. Notamment, la résolution de collision , qui dépend du types des deux objets en question, a pu être écrite de manière très élégante et modulaire grâce aux fonction génériques.

L'expérimentation effectuée sur le flot de contrôle du jeu a permis de démontrer que l'architecture du moteur peut être facilement modifier afin de mettre à l'essai de nouvelles techniques de gestion du flot de contrôle pour jeu. L'utilisation du système de coroutines permis d'implanter efficacement une version multi-joueurs du jeu, mais fut inappropriée pour effectuer la gestion complète du flot de contrôle du jeu.

Puisque *Space Invaders* est un jeu très simple, la gestion mémoire automatique n'a jamais constituée un obstacle au développement puisque trop peu d'objets étaient utilisés et les pauses dues à la récupération mémoire utilisaient environ 2 millisecondes, soit environ 1% du temps processeur requis pour le rendu d'une image.

Il serait maintenant intéressant d'implanter un jeu plus complexe afin de valider si les techniques utilisées pour *Space Invaders* sont toujours applicables et de

voir si elles s'étendent bien aux nouveaux défis de programmations résultant de l'implantation d'un tel jeu.

6.2 Développement de *Lode Runner*

Après avoir développé *Space Invaders*, un deuxième jeu a dû être développé afin de pouvoir consolider les techniques développées. Le jeu *Lode Runner* a été choisi dans le but de remplir cette tâche pour plusieurs raisons. Principalement, ce jeu contient de nouveaux éléments qui n'étaient pas présent dans *Space Invaders* comme le concept d'intelligence artificielle et étendait d'autre concepts qui y étaient très primitifs comme le concept de niveaux et d'états d'entités.

Ce jeu consiste à tenter de capturer tout l'or dispersé dans le niveau en évitant de rentrer en contact avec les robots ennemis. Pour y arriver, le joueur doit grimper des échelles, passer sur des cordes et utiliser son pistolet laser qui permet de faire des trous à la gauche ou à la droite du joueur. Ces trous se referment après un délais prescrits et donc les joueurs doivent se dépêcher afin d'éviter de rester pris au piège à l'intérieur d'un trou. Lorsque tout l'or est en possession du joueur, une échelle de sortie apparaît et mène le joueur au niveau suivant. La joueur dispose de trois vies pour se rendre le plus loin possible dans le jeu et obtenir le meilleur score possible. La figure 6.13 illustre une capture d'écran du jeu développé.

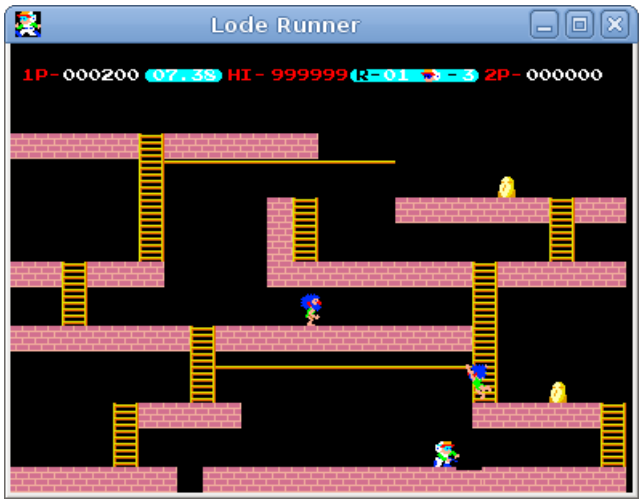

FIGURE 6.13 – Capture d'écran du jeu *Lode Runner* développé

6.2.1 Logique du jeu

Afin de pouvoir développer ce jeu, les meilleurs techniques utilisées pour le développement de *Space Invaders* ont été réutilisées et raffinées pour les besoins plus grands de *Lode Runner* . Par contre, le flot de contrôle de la logique du jeu est régi de manière beaucoup plus conservatrice en utilisant une boucle principale effectuant le travail à faire pour chaque image du jeu (*Game Loop*). Il en résulte qu'une fonction générique `animate` est appelée avant le rendu de chaque image de manière à effectuer le travail que doit accomplir chaque objet avant ce rendu. La figure 6.14 illustre une partie de la boucle principale du jeu et l'instance de la fonction générique `advance-frame!` pour un niveau de jeu. Cette figure illustre la simplicité du flot de contrôle du jeu résultante grâce à l'utilisation de fonction génériques et de fonctions d'ordre supérieures.

```
(define-method (advance-frame! (level level) keys-down keys-up)
  (cond ((level-get 'player level) =>
         (lambda (player)
           (player-velocity-set! player point-zero))))
  (cond
   ((level-paused? level) 'do-nothing)
   ...
   (else
    (update! level level current-time (lambda (t) (+ t (fl/ 1. (FPS)))))
    (if (> (level-current-time level) (level-time-limit level))
        (level-game-over! level)
        (begin
          (for-each (flip process-key level) keys-down)
          (for-each (flip animate level) (level-objects level)))))))

;; Game loop
(let loop ((level (current-level)))
  (if exit-requested? (quit))
  (poll-SDL-events)
  (advance-frame! level (key-down-table-keys) (key-up-table-keys))
  (render-scene screen level)
  (loop (current-level)))
```

FIGURE 6.14 – Boucle principale du jeu *Lode Runner*

Par contre, l'utilisation d'une telle boucle fait en sorte que la vitesse du jeu dépend directement du taux de rafraîchissement de celui-ci.

6.2.2 Gestion de l'état des objets

Les entités du jeu *Lode Runner* sont sensiblement plus complexes que celles présente dans le jeu précédant. En effet, ces dernières possèdent sont animés et réagissent différemment en fonction du contexte de celle-ci. Encore une fois, ce sujet est traité de manière traditionnelle en utilisant des états d'objets et des machines à

états afin de régir leurs comportements et les animations de ceux-ci. La figure 6.15 donne le contenu de la classe `human-like` qui sert de base aux classes `player` et `robot` qui implantent respectivement les instances du joueurs et des robots ennemis. Elle démontre bien la complexité de l'état de ces entités.

```
(define-class human-like (game-object moving statefull)
  (slot: can-climb-up?)
  (slot: can-go-down?)
  (slot: can-use-rope?)
  (slot: can-walk?)
  (slot: droped-rope?)
  (slot: stuck-in-hole?)
  (slot: facing-direction)
  (slot: walk-cycle-state)
  (slot: shooting?)
  (slot: escaping?)
  (constructor: (lambda (self x0 y0 initial-velocity id) ...)))
```

FIGURE 6.15 – Définition de la classe `human-like` dans le jeu *Lode Runner*

La fonction générique `change-state!` utilise l'état des instances de ces classes pour déterminer l'animation qui doit être utilisée afin d'obtenir un rendu cohérent de l'entité. La figure 6.16 contient la gestion principale de l'état des entités appartenant à la hiérarchie de la classe `human-like`. Ici, avant le rendu de chaque image, l'état de chaque objet est vérifié et mis-à-jour. Puisque les états possibles d'une instance de la class `human-like` ne sont pas orthogonaux, *i.e.* que plusieurs états correspondant à des animations différentes peuvent être présent en même temps, l'ordre de la modification de l'état est significative. Si ces derniers avaient été orthogonaux, la gestion de ceux-ci aurait pu être effectuée en utilisant la discrimination

de méthodes sur les valeurs de ces états. Le code résultant aurait donc été très limpide.

```
(define-method (change-state! (hl human-like) level)
  (let* ((v (moving-velocity hl)))
    (cond
     ((eq? (human-like-escaping? hl) 'escaping) (ascend-cycle! hl))
     ((human-like-shooting? hl)                 (shoot-cycle hl))
     ((human-like-stuck-in-hole? hl)            (dying-cycle! hl))
     ((and (not (zero? (point-x v)))
           (human-like-can-walk? hl))
                                                (walk-cycle! hl))
     ((human-like-can-use-rope? hl)
       (if (or (not (zero? (point-x v)))
               (not (memq (human-like-state hl) rope-states)))
           (rope-cycle! hl)
           'keep-same-state^_^))
     ((not (human-like-can-go-forward? hl))     (fall-cycle! hl))
     ((not (zero? (point-y v)))                 (ascend-cycle! hl))
     (else (reset-walk-cycle! hl)))))
```

FIGURE 6.16 – Gestion de l'état d'une entité appartenant à la hiérarchie de type `human-like` dans *Lode Runner*

Les fonctions d'animation des entités (`walk-cycle`, `rope-cycle!`, ...) produisent des animations se répétant de manière cyclique. Puisque ce genre d'animation est fréquente dans le jeu, elle fut abstraite dans une forme spéciale afin de simplifier la définition de celles-ci. La figure 6.17 présente cette dernière qui construit une définition de fonction qui met-à-jour les champs `cycle-member` et `state-member` afin de produire correctement le rendu de l'animation. Le champs `cycle-member` de l'instance reçue contiendra un nombre correspondant à l'état actuel de l'animation. Cet état est conservé dans l'attribut `state-member` et sera utilisé par

l'algorithme de rendu graphique du jeu. Le paramètre `cycle-delta` représente le nombre d'images qui seront rendues avant le changement de l'état actuel de l'entité.

```
(define-macro (define-cyclic-animation name
                #!key base-class cycle-delta cycle-member
                      state-member states other-actions-fun)
  (let ((obj (gensym 'obj))
        (cycling-delta (gensym 'cycling-delta))
        ...)
    `(define (,name ,obj)
       (let* ((,cycle-length (* ,cycle-delta ,(length states))))
         (update! ,obj ,base-class ,cycle-member
                  (lambda (s) (modulo (+ s 1) ,cycle-length)))
         (let ((,next-state
                (case (quotient (,cycle-member-getter ,obj) ,cycle-delta)
                  ,@(map-with-index (lambda (i x) `((,i) ',x)) states)
                  (else (error ...)))))
           (,state-member-setter ,obj ,next-state)
           (,other-actions-fun ,obj))))))
```

FIGURE 6.17 – Forme spéciale utilisée afin de simplifier la définition d'animations cycliques d'entité du jeu *Lode Runner*

Ainsi les animations du jeu peuvent être facilement définies en utilisant cette forme spéciale. La figure 6.18 illustre la définition de l'animation de marche utilisée par les sous-classes de la classe `human-like`. On peut donc ainsi tirer profit du polymorphisme du système objet en plus du petit langage de définition d'animations afin de définir de manière claire et concise une animation cyclique.

```
(define-cyclic-animation walk-cycle!
  base-class: human-like
  cycle-delta: 5
  cycle-member: walk-cycle-state state-member: state
  states: (standing-up standing-left standing-up standing-right)
  other-actions-fun:
    (lambda (obj) (human-like-facing-direction-set!
                   obj (human-like-get-direction obj))))
```

FIGURE 6.18 – Définition d'animation cyclique dans le jeu *Lode Runner*

6.2.3 Détection et résolution de collisions

Une nette amélioration apportée au jeu, en comparaison celui développé précédemment, est l'utilisation d'une grille pour accélérer la détection de collisions. Le principe est simple : une grille bidimensionnelle contenant un nombre de subdivisions variable est créée. Lorsqu'un objet est ajouté au jeu, il est ajouté dans chacune des cases de la grilles auxquelles il entre en contacte et, réciproquement, cet objet possède des pointeurs vers chacune de ces cases de la grille. Si la grille est assez fine, alors la détection de collisions devient triviale : il ne suffit que de regarder dans cases de la grille qui touchent l'objet en question afin de voir si d'autres objets s'y trouvent et si c'est le cas, alors une collision est détectée. Bien sûr, la taille de la grille influence sur le travail nécessaire lors de chaque ajout et déplacement d'objets, mais puisque dans *Lode Runner* la plupart des objets sont statiques, cela ne pose pas de problèmes réels. La figure 6.19 illustre l'algorithme de détection de collisions utilisé.

La résolution de collisions ainsi que le rendu graphique sont effectués tous deux

```
(define (detect-collisions obj level)
  (filter (lambda (x) (not (eq? x obj)))
          (fold-l (curry2* set-union eq?)
                  '()
                  (map (curry2 grid-get (level-grid level))
                       (game-object-grid-cells obj)))))
```

FIGURE 6.19 – Détection de collisions dans *Lode Runner*

de manière très similaire à ce qui a été utilisé dans *Space Invaders* . Des fonctions génériques `resolve-collision` et `render` s'occupent de choisir dynamiquement la bonne instance de ces fonctions génériques en fonction des objets passés.

6.2.4 Rendu graphique

Il est intéressant de noter que le rendu graphique est légèrement plus complexe que pour le jeu *Space Invaders* , car il est possible d'avoir des objets qui se superposent. Afin de résoudre les problèmes de visibilités qui en résultent, l'algorithme du peintre a été utilisé. Ce dernier est implanté en utilisant des couches sur lesquelles doivent apparaître les objets afin de donner des priorités de rendu. Comme l'indique la figure 6.20, le joueur se trouve sur la couche `human-like-layer` qui est prioritaire aux couches des objets du jeu, résultant en l'affichage du joueur par dessus ces derniers. Les objets sont triés lorsque des nouveaux objets sont ajoutés au niveau afin d'éviter à faire ce tri lors du rendu de chaque images.

```
(enum background-layer stage-layer foreground-layer human-like-layer top-layer)
(define-method (get-layer (h human-like))    human-like-layer)
(define-method (get-layer (h hole))          foreground-layer)
(define-method (get-layer (s stage))         stage-layer)
(define-method (get-layer (obj game-object)) foreground-layer)
```

FIGURE 6.20 – Implantation des couches de rendu dans *Lode Runner*

6.2.5 Intelligence Artificielle

L'intelligence artificielle du jeu fut implantée de manière directe en utilisant des fermeture afin d'abstraire les différents algorithmes potentiels (proposant différentes difficultés au joueurs). Seuls deux algorithmes furent toutefois implantés pour le jeu. Le premier, complètement trivial, fait en sorte que les robots demeurent immobiles. Le deuxième utilise un environnement totalement observable et reproduit un comportement *similaire* à celui des ennemis de la version originale. Ces algorithmes sont présentés dans les figures 6.21 et 6.22. Les fermetures contenant ces algorithmes sont utilisées de manière dynamique en fonction du niveau de difficulté choisi par le joueur. Il est ainsi possible de pouvoir modifier durant la partie la difficulté. Aussi, l'utilisation de fermeture donne la chance de pouvoir créer de nouvelles fermetures contenant des informations plus spécialisées pour l'état actuel du jeu. L'implantation du choix dynamique de l'algorithme d'intelligence artificielle est présenté dans la figure 6.23.

```
(define (immobile-ai robot level)
  (robot-velocity-set! robot point-zero))
```

FIGURE 6.21 – Algorithme trivial d'intelligence artificielle des robots

```
(define (seeker-ai robot level)
  (let ((player (level-get 'player level)))
    (if player
        (let* ((dir (point-sub player robot))
               (y-axis? (let ((x (point-x dir)) (y (point-y dir)))
                          (and (not (zero? y))
                               (can-go-up/down? robot x y))))
               (velo (if y-axis?
                         (new point 0 robot-movement-speed)
                         (new point robot-movement-speed 0)))
               (factor (if (< (if y-axis? (point-y dir) (point-x dir)) 0)
                           -1
                           1)))
          (robot-velocity-set! robot (point-scalar-mult velo factor))))))
```

FIGURE 6.22 – Algorithme d'intelligence artificiel reproduisant un comportement *similaire* au jeu original

6.2.6 Conclusion

L'implantation du jeu *Lode Runner* a permis de consolider certaines des techniques d'implantation de jeux développés pour *Space Invaders* . En effet, l'utilisation de fonctions génériques a permis d'exprimer simplement et de manière très *modulaire* le comportement des objets dans le jeu.

Le flot de contrôle du jeu a été implanté de manière beaucoup plus traditionnelle que ce fut le cas pour *Space Invaders* . Il en résulte que le jeu dépend beaucoup des notions d'états des objets. Puisque le langage Scheme offre le meilleur des deux

```
(define (get-ai-fun difficulty)
  (case difficulty
    ((easy) immobile-ai)
    ((hard) seeker-ai)
    (else immobile-ai)))

(define (run-ai robot level)
  ((get-ai-fun (level-difficulty level)) robot level))
```

FIGURE 6.23 – Utilisation dynamique des algorithmes d'intelligence artificielle

mondes, *i.e.* qu'il offre un langage fonctionnel permettant des mutations, ce type de contrôle de flot a permis de tirer profit des avantages qu'offrent la programmation fonctionnelle (fonctions d'ordre supérieures, fermetures, etc...) tout en utilisant un modèle éprouvé de flot de contrôle et de modularisation.

La gestion des états plus complexes des objets du jeu a pu être écrite de manière très élégante grâce à la conception d'un langage spécifique au domaine minimaliste. Il fut ainsi possible de pouvoir décrire le « quoi »des animations tout en cachant la mécanique commune aux animations des entités du jeu.

L'ajout d'une grille pour accélérer la détection de collisions semble peut-être peu significative puisque *Lode Runner* ne contient pas une très grande quantité d'objet, mais ce système a été ajouté dans le but de pouvoir facilement et rapidement implanter des jeux qui seront plus complexes et qui nécessiteraient de tels optimisations.

Tout comme ce fut le cas pour *Space Invaders* , la gestion mémoire automatique n'a pas été un problème pour le développement de ce jeu. En effet, malgré un

plus grand nombre d'objets présents, ceux-ci demeurent trop peu pour éprouver la gestion mémoire sur des ordinateurs récents.

6.3 Conclusion

Ainsi, il a été possible de développer deux jeux vidéo de complexité croissante en utilisant le langage Scheme. Le développement du premier jeu, *Space Invaders* , a permis d'identifier et de répondre aux besoins fondamentaux de jeux vidéo : rendu graphique, détection de collisions, etc... Pour ce faire, un système de programmation orienté objet et un système de coroutines ont été ajouté au langage Scheme. Aussi, le premier jeu a permis de vérifier la difficulté inhérente au changement de l'architecture du flot de contrôle d'un jeu en passant d'un flot contrôlé par une simulation par événements discrets à une simulation par agents. La conversion s'est faite sans trop d'efforts, grâce aux abstractions utilisées dans le jeu.

Un deuxième jeu a, par la suite, été développé afin de permettre de vérifier si les techniques utilisées pour *Space Invaders* s'adaptaient à un autre jeu plus complexe. Ainsi, *Lode Runner* fut développé en utilisant les abstractions de haut niveau que permettent les fonctions de première classe combinées avec un système d'objets très polyvalent et la conception de nouvelles formes spéciales. En utilisant, cette fois ci, une approche conservatrice sur le flot de contrôle. Le résultat fut très clair : le jeu a été développé rapidement et sans problèmes importants.

Ainsi, il serait maintenant intéressant de voir quel serait l'impact de l'implanta-

tion en Scheme d'un jeu de qualité commerciale afin de tester plus profondément les limites que pourrait imposer le langage sur des jeux plus exigeant tant en utilisation processeur qu'en utilisation mémoire.

```
(define (invader-controller)
  (define (init-state)
    (recv
     (init
      (subscribe instant-components (self))
      (broadcast `(invader-row ,(Inv-Controller-row (self)))
                 'move)
      (wait-state))))
  (define-wait-state wait-state moved (inv-nb)
    (recv
     (go-down-warning
      (let ((row-nb (Inv-Controller-row (self))))
        (broadcast `(invader-row ,row-nb) 'wall-collision)
        (for i 0 (< i invader-row-number)
             (if (not (= i row-nb))
                 (if (not (zero? (msg-list-size `(invader-row ,i))))
                     (let ((move-back? (< i row-nb)))
                       (broadcast `(invader-row ,i)
                                  `(wall-collision ,move-back? #f))))
                 (broadcast `(invader-row ,i)
                            `(wall-collision  #t ,(self))))))
      (wait-state))
     (after 0
            (wait-for-next-instant)
            (broadcast `(row-controller
                         ,(next-row (Inv-Controller-row (self))))
                       'init)
            (unsubscribe instant-components (self))
            (init-state))))
  (init-state))
```

FIGURE 6.24 – Comportement du méta-objet `invader-controller` qui régi le comportement d'une rangée d'*invaders*

CHAPITRE 7

TRAVAUX RELIÉS

Dans un premier temps, il serait intéressant de comparer les différents langages utilisés dans l'industrie du jeu vidéo avec Scheme afin d'en faire ressortir les différences. Ces différences mènent directement à une méthode de développement qui seront nécessairement différentes. Ainsi, un portrait des techniques de programmation utilisées dans l'industrie est dressé dans le but de situer où se retrouvent les travaux présentés dans ce mémoire.

Les langages de programmation de la famille *Lisp* ont déjà été utilisés pour produire des jeux vidéo commerciaux de très bonne qualité. Certains seront cités et une revue de l'expérience acquise par les développeurs sera exposée dans le but de pouvoir comparer leurs expériences avec celle acquise pour le développement de *Space Invaders* et de *Lode Runner* .

7.1 Comparaison de langages

L'industrie du jeu vidéo est déjà bien établie et donc, certains langages semblent être favorisés quant au développement de jeux vidéo par les compagnies oeuvrant dans le domaine. Cette section présente ces langages dans le but de comparer ces langages à Scheme afin de mettre en lumière la possibilité de les remplacer par ce dernier.

7.1.1 C++

Langage principal utilisé pour le développement de jeu vidéo est sans aucun doute le langage C++. Ce dernier est une version orientée objet du langage C qui est aussi très répandu et apprécié pour être un langage suffisamment de bas niveau pour permettre d'écrire du code très efficace. Ainsi, C++ apporte les paradigmes de la programmation orientée objet en introduisant des classes à héritage multiples et des fonctions virtuelles permettant un *dispatch* simple. C++ possède également un système de gestion d'exceptions et de *templates*.

Ainsi, le langage C++ semble être grandement apprécié pour son mélange de concepts de haut niveau comme la programmation orientée objet et de bas niveau, comme la gestion de mémoire manuelle et son modèle d'exécution très près du matériel. Il diffère grandement d'un langage de haut niveau tel que Scheme. En effet, il ne possède pas de fermetures, de continuations et de système de macros évolué. La programmation en C++ est dite impérative parce que le résultat du programme dépend intrinsèquement de l'état de celui-ci. Puisqu'il s'agit d'un langage typé statiquement, il peut difficilement être interprèté tout en conservant ses avantages de performances.

Il en découle donc que C++ met à la disposition beaucoup moins de concepts de haut niveaux, réduisant ainsi la facilité d'abstractions et de programmation en générale pour donner accès plus facilement à de bonnes performances. Ainsi, C++ semble un bon choix de langage pour des plateformes possédant un matériel

limité, tant en mémoire que puissance du processeur. Par contre, sur des machines plus puissantes, Scheme semble plus approprié parce qu'il apporte beaucoup plus de puissance d'abstraction aux programmeurs. Une utilisation de C++ pour implanter les parties critiques du moteur de jeu pourrait tout en utilisant un langage comme Scheme pour développer la partie *gameplay* du jeu semble aussi une bonne approche sur des plateformes performantes afin d'optimiser les performances du jeu.

Il est également important de souligner le facteur de la main d'oeuvre pour l'adoption de Scheme pour remplacer C++. En effet, la quantité de programmeurs maîtrisant le langage Scheme est très restreintes en comparaison aux nombres de programmeurs C++ et donc, même si Scheme pourrait se révéler un meilleur choix technologique pour le développement d'un jeu, il est possible que l'embauche de suffisamment de programmeurs Scheme soit trop difficile et coûteux vis-à-vis l'embauche de programmeurs C++ beaucoup plus nombreux. Rendant ainsi C++ plus attrayant et moins risqué pour une entreprise.

7.1.2 Lua

Un langage Lua [48] est fréquemment utilisé non pas pour implanter le moteur d'un jeu vidéo, mais plutôt pour effectuer le *scripting* de celui-ci, *i.e.* le développement dynamique de niveaux, d'environnements, etc... Lua a été utilisé notamment dans le jeu *World Of Warcraft* afin de permettre aux joueurs de pouvoir ajouter des éléments au jeu de manière dynamique.

Ce langage est reconnu pour avoir un environnement d'exécution très léger, environ 150 kilo-octets, ce qui fait en sorte que l'ajout de cet environnement se fait très bien, surtout lorsque cette intégration est faite sur des plateformes au matériel plus limité comme des consoles portables. Lua contient certains éléments de la programmation dite de haut niveau dont des fermetures, une gestion de mémoire automatique, l'optimisation d'appels terminaux et un système de coroutines. Il fourni aussi un système d'objet très limité à la Javascript [22], basé sur le prototypage d'objets.

Ainsi, malgré le fait que ce dernier contient plusieurs aspects de la programmation de haut niveau, Lua possède plusieurs différence au langage Scheme. Il ne supporte pas la réification de continuation et ne possède pas de système de macros. Il s'agit d'un langage généralement uniquement interprété, ce qui ne le rend pas approprié pour écrire le moteur d'un jeu vidéo.

Pour les mêmes raison que Lua, d'autre langages de programmations sont également utilisés afin d'effectuer le *scrpting* d'un jeu. Parmi ceux-ci, on retrouve principalement Python [49] et Javascript [22] qui sont très similaire à Lua et donc constituent eux aussi un sous ensemble de Scheme.

7.2 Jeux en *Lisp*

Malgré la domination du langage C++ dans l'industrie du jeu vidéo, certains jeux ont déjà été reconnu pour utiliser Scheme à différents degrés. Ceux-ci sont mis

en lumière dans cette section afin de permettre de comparer l'expérience acquise par ces développeurs à celle acquise pour la rédaction de ce mémoire.

7.2.1 Naughty Dog

La compagnie Naughty Dog [50] est bien connue pour utiliser des dialectes de *Lisp* dans le développement de jeux vidéo. En effet, cette compagnie a écrits plusieurs jeux sur la *PlayStation 2* en utilisant le compilateur *GOAL*(*Game Oriented Assembly Lisp*), un système maison uniquement compilé vers du code machine pour le *PlayStation 2* qui comprenait entre autre un système d'objet. Ce dernier fut utilisé pour développer entièrement de nombreux jeux sur la console dont les jeux *Crash Bandicoot* et *Jak and Dexter*.

Scott Shumaker, programmeur chef chez Naughty Dog, a commenté leur utilisation de GOAL dans un article sur le site Gamasutra [51]. Dans ce dernier, il mentionne que le développement de leur propre compilateur de *Lisp* leur a permis d'utiliser plusieurs techniques qui n'auraient pas pu être utilisées avec d'autres langages. Entre autre, leur système, malgré le fait qu'il ne comprenait pas d'interpète, permettait la compilation de code et le chargement de celui-ci directement dans le jeu qui était exécuté sur une console *PlayStation 2* donnant accès ainsi à de la programmation très dynamique basée sur du prototypage. Il mentionne également que GOAL comprenait un système de coroutine. Un exemple de code simplifié pour GOAL est donné dans la figure 7.1.

```
(dotimes (ii (num-frames idle))
  (set! frame-num ii)
  (suspend))
```

FIGURE 7.1 – Exemple de code pour GOAL utilisant le système de coroutines

Il mentionne également dans cet article que l'utilisation de leur système GOAL a été la source de plusieurs problèmes principalement dus au fait que GOAL fut développé par un seul programmeur *Lisp* et que lui seul en comprenait parfaitement l'étendue. Il mentionne aussi qu'il fut difficile de trouver de la main d'oeuvre maîtrisant le langage *Lisp* et que même ceux-ci devaient tout de même s'adapter au système utilisé.

Ils ont par la suite créé les jeux *Uncharted : Drake's fortune* et *Uncharted 2 : Among Thieves* qui sont rapidement devenus très populaire pour l'action et le rendu graphique exceptionnel contenu dans ces titres. Dan Liebgold de Naughty Dog a donné une conférence sur le développement du jeu *Uncharted : Drake's fortune* [1] et mentionna qu'ils ont utilisé le langage Scheme afin de pouvoir exprimer les données du jeu de manière dynamique, mais que le moteur du jeu est écrit en C++. Ils profitent ainsi des performances qu'offrent C++ pour les zones critiques du jeu et utilisent un langage de plus haut niveau pour la description des données et utilisent un schéma de compilation dynamique similaire à celui utilisé par GOAL. La figure 7.2 illustre la constructions de données utilisées dans ce jeu.

```
(deftype vec4 (:align 16)
  ((x float) (y float)
   (z float) (w float :default 0)))
(deftype quaternion (:parent vec4) ())
(define (axis-angle->quat axis angle)
 (let ((sin-angle/2 (sin (* 0.5 angle))))
  (new quaternion
       :x (* (-> axis x) sin-angle/2)
       :y (* (-> axis y) sin-angle/2)
       :z (* (-> axis z) sin-angle/2)
       :w (cos (* 0.5 angle)))))
```

FIGURE 7.2 – Code de données utilisées dans le jeu *Uncharted : Drake's fortune* [1]

7.2.2 QuantZ

Récemment, le jeu QuantZ [52] fut développé presque exclusivement en utilisant le langage Scheme. Ce dernier constitue un jeu de puzzle où le but est d'agencer des billes de même couleurs sur un cube.

[TODO: *Trouver comment dire que j'ai travaille sur le jeu... O_O*]

L'utilisation de Scheme dans QuantZ a apporté plusieurs avantages technologiques. En effet, le dynamisme du langage a su profiter aux développeurs en leur permettant de pouvoir débugger le jeu lors de son exécution résultant permettant ainsi la résolution rapides de problèmes.

Aussi, la syntaxe sous formes de S expressions de a été utilisée a profit en créant un système d'analyse de code permettant de pouvoir extraire toutes les chaînes de caractères présente de le jeu afin de pouvoir créer facilement des outils d'internationalisations. Ces outils permettent entre autre la création automatiques

de fontes optimisées par langues, qui contiennent uniquement les caractères utilisées dans celles-ci.

Les fermetures disponible en Scheme ont été utilisées comme base d'un système d'interface graphique complet permettant la création de fenêtres, de boutons, etc... En utilisant ces fermetures pour conserver l'état de ces entités, on obtient ainsi un résultat similaire l'utilisation d'un système de programmation orienté objet, où l'état est conservé dans ces fermetures, mais ne fournissant pas de fonctions génériques.

Afin d'éviter d'avoir des problèmes de pauses dues à la gestion de mémoire automatique, plusieurs techniques ont été utilisées afin de cacher des données statiques au gestionnaire automatique de mémoire. En effet, l'utilisation de la sérialisation de données disponible dans Gambit-C a été utilisée afin de transformer de grosses structures de données statiques lors de l'exécution du jeu et ainsi éviter que le ramasse miettes n'aies à la parcourir pour chaque images du jeu générées.

L'état du jeu a été implanté en utilisant un mécanisme de programmation fonctionnelle réactive basée sur Scheme permettant l'écriture élégante de niveaux grâce à la propagation de signaux correspondant à l'état du jeu.

Ainsi, le jeu QuantZ représente un exemple concret d'un jeu commercial ayant utilisé Scheme comme langage principal de développement. Leur utilisation du langage a su tirer profit de la plupart des particularité de Scheme dont notamment l'utilisation de fonctions de premier ordre et du dynamisme du système Gambit-C.

7.3 Conclusion

Ainsi, plusieurs langages sont utilisés dans l'industrie du jeu vidéo dont, notamment les langages Lua et C++. Après une comparaison au langage Scheme, on constate que ces derniers répondent à des besoins précis présents dans les jeux vidéo. Lua propose un dynamisme permettant un développement dynamique tandis que C++ permet d'obtenir de très bonnes performances au coût d'avoir moins de puissance d'abstractions. Le choix du langage le plus approprié pour le développement d'un jeu devrait donc être fait en prenant en considération le matériel utilisé afin d'assurer que de pouvoir utiliser le plus de puissance expressive que possible, sans toutefois avoir un coût relié à ces abstractions trop élevé.

Aussi, *Lisp* et Scheme ont déjà été utilisés dans l'industrie du jeu vidéo pour développer des jeux de grandes qualités tels *Jak and Dexter*, *Uncharted : Drake's Fortune* et *QuantZ*. L'expérience acquise par les développeurs de ces jeux est très positive. On constate de manière unanime ces langages leurs ont fourni un environnement de développement dynamique et l'extensibilité du langage a jouer à l'avantage de ceux-ci. On constate ainsi, que malgré la faible adoption de Scheme pour l'écriture de jeux vidéo commerciaux, les équipes l'ayant utilisé semblent l'avoir beaucoup apprécier et surtout, l'utilise toujours dans le développement de leurs jeux.

CHAPITRE 8

CONCLUSION

L'expérience acquise lors du développement des jeux *Space Invaders* et *Lode Runner* permet donc de faire le point sur les forces et les faiblesses de l'utilisation d'un langage de programmation fonctionnelle et dynamique tel que Scheme pour le développement de jeux vidéo.

L'expressivité du langage Scheme, notamment grâce aux fonctions de premier ordre et aux macros, a permis de pouvoir développer un système de programmation orientée objet et l'intégrer directement au langage. Aussi, la réification de continuations a rendu possible le développement d'un système de coroutines élaboré. L'utilisation de ces systèmes dans la programmation de jeu vidéo fut très bénéfique quant à l'amélioration du code produit grâce à l'introduction d'une bonne modularité fournie par les fonctions génériques du système objets et une grande simplicité pour l'implantation de parties multi-joueurs grâce à l'utilisation de coroutines pour encapsuler les parties se déroulant en parallèle. Le coût de développement de tels modules, en utilisant des langages donnant accès à moins de puissance d'abstraction, aurait certainement rendu le développement de ceux-ci impossible et donc, Scheme s'est révélé un langage très puissant pour le développement de ces outils indispensables.

Le dynamisme du langage a également jouer un rôle implicite très important

pour le développement de ces jeux. En effet, la présence d'un déboggeur très dynamique a apporté une grande flexibilité au développement et permis de développer efficacement ces deux jeux. En effet, malgré le fait que les erreurs de programmation sont signalées durant l'exécution du jeu, la résolution de celles-ci sont font très rapidement grâce à la possibilité d'introspection de l'état du programme et d'évaluation dynamique de code.

Aussi, le langage Scheme a permis de facilement pouvoir expérimenter sur le contrôle de flot des jeux en permettant de développer facilement des systèmes non-triviaux de simulations à événements discrets et de simulation par agents (coroutines). La modification du flot de contrôle du premier jeu a pu être fait sont trop d'efforts grâce aux abstractions faites entre autre par le système d'objets.

Bien sûr, la modularité de programmes apporte généralement des coût en performances pour ceux-ci. Ainsi, une difficulté d'utiliser un langage comme Scheme face à la programmation de jeux vidéo réside entre trouver le bon équilibre entre le niveau d'abstraction et la spécification de code (optimisations). Bien sûr cette balance varie grandement en fonction du matériel sur lequel le jeu sera porté. S'il s'agit d'une plateforme au matériel très limité, alors le coût d'abstraction pourrait peut-être s'avérer trop grand. Dans un tel cas, l'utilisation de langages de plus bas niveau, comme C++, semble plus appropriée afin de bâtir le moteur du jeu, mais Scheme pourrait alors très bien être utilisé comme langage de script afin de permettre de développer dynamiquement le *gameplay* du jeu. Par contre, s'il s'agit de

plateforme performantes, comme c'est le cas pour les jeux développés dans le cadre de ce mémoire, alors la possibilité de modulariser le code est beaucoup plus grande et les performances deviennent moins critiques, face à la réutilisabilité potentielle du code écrit. Scheme devient alors un candidat idéal.

Un problème potentiel qui n'a pas été adressé dans ce mémoire est relié à l'allocation et la gestion de mémoire automatique utilisée par le langage Scheme. En effet, puisque Scheme est un langage fonctionnel, ce dernier effectue généralement beaucoup d'allocations mémoire. Ainsi, il est possible que sur des systèmes où le coût d'allocation mémoire est élevé, l'utilisation d'un tel langage devienne plus grand. De même, la présence du gestionnaire automatique de mémoire pourrait potentiellement engendrer des pauses du jeu qui sont très indésirables, si les pause dues à la gestion automatiques deviennent trop grandes. Ces phénomènes n'ont par contre pas été observés dans les jeux développés pour ce mémoire, puisqu'ils demeuraient trop simples.

Ainsi, malgré le coût que peut apporter l'utilisation d'abstractions, nous croyons que le bénéfice résultant est bien supérieur à ce dernier, surtout sur des plateformes performantes. Les techniques de programmations orientées objets développées nous ont permis de rapidement pouvoir développer un nouveau jeu en réutilisant plusieurs modules, dont notamment les modules de rendus graphiques et de résolution de collisions.

BIBLIOGRAPHIE

[1] Dan Liebgold. Adventures in Data Compilation : Uncharted : Drake's Fortune. Game Developers Conference, 2008.

[2] NPD Group. 2007 U.S. Video Game And PC Game Sales Exceed $18.8 Billion Marking Third Consecutive Year Of Record-Breaking Sales. `http://www.npd.com/press/releases/press_080131b.html`, 2008.

[3] NPD Group. U.S. Retail Sales of Non-Games Software Experience near Double-Digit Decline in 2008. `http://www.npd.com/press/releases/press_090217.html`, 2009.

[4] Chris Morris. Special to CNBC.com — 12 Jun 2009 — 05 :12 PM ET. `http://www.cnbc.com/id/31331241`, 2009.

[5] ECMA. C# Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, 2006.

[6] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9) :26–76, 1998.

[7] Marc Feeley. Gambit-C. `http://www.iro.umontreal.ca/~gambit`.

[8] Marc Feeley. Gambit-C vs. Bigloo vs. Chicken vs. MzScheme vs. Scheme48. `http://www.iro.umontreal.ca/~gambit/bench.html`, September 3, 2009.

[9] Leonard Herman, Jer Horwitz, Steve Kent, Skyler Miller. The History Of Video Games. `http://www.gamespot.com/gamespot/features/video/hov/index.html`, September 3, 2009.

[10] Johnny L. Wilson Rusel DeMaria. *High score ! : the illustrated history of electronic games*. McGraw Hill, 2004.

[11] Thomas T. Goldsmith Jr. Cathode Ray Tube Amusement Device. U.S. Patent #2455992, 1948.

[12] Dennis Ritchie and Ken Thompson. Unix past. http ://www.unix.org/what_is_unix/history_timeline.html.

[13] Joshua Walrath. 30 frames per second vs. 60 frames per second. http ://www.daniele.ch/school/30vs60/30vs60_1.html, 1999.

[14] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass., 1996.

[15] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4) :184–195, 1960.

[16] Guy L. Steele. *Common Lisp the Language*. Digital Press, 2nd edition edition, 1984.

[17] Paul Graham. *ANSI Common Lisp*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

[18] Gerald Jay Sussman. Scheme : An interpreter for extended lambda calculus. In *Memo 349, MIT AI Lab*, 1975.

[19] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. 1973.

[20] Gerald Jay Sussman and Guy L. Steele, Jr. The first report on scheme revisited. *Higher Order Symbol. Comput.*, 11(4) :399–404, 1998.

[21] Scheme request for implementation.
`http://srfi.schemers.org/`.

[22] E. C. M. A. International. *ECMA-262 : ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[23] Ruby programming language.
`http://www.ruby-lang.org/`.

[24] Perl language.
`http://www.perl.org/`.

[25] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2) :345–363, 1936.

[26] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[27] SP Jones. *Haskell 98 language and libraries : the revised report*. Cambridge Universiry Press, Cambridge, MA, USA, 2003.

[28] E E Kohlbecker. *Syntactic extensions in the programming language LISP*. PhD thesis, Bloomington, IN, USA, 1986.

[29] Paul R. Wilson. Uniprocessor garbage collection techniques. In *IWMM '92 : Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.

[30] Norbert Podhorszki. Garbage collection techniques. Technical report.

[31] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques, 1995.

[32] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11) :611–612, 1969.

[33] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6) :419–429, 1983.

[34] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[35] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection : an exercise in cooperation. *Commun. ACM*, 21(11) :966–975, 1978.

[36] Richard Kelsey. Scheme Request For Implementation 9 : Defining Record Types.

`http://srfi.schemers.org/srfi-9/srfi-9.html`, 1999.

[37] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[38] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification x2ji3 document 88-002r. *SIGPLAN Notices*, 23(Special Issue) :1.1–2.94, 1988.

[39] Gregor Kiczales, Jim D. Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, July 1991.

[40] Christian Queinnec and Lip Inriarocquencourt. Meroon v3 : A small, efficient and enhanced object system.

[41] Kenneth Dickey. Thinking scheme. Workshop on Scheme and Functional Programming., september 2008.

[42] Marc Feeley. SRFI 18 : Multithreading support. `http://srfi.schemers.org/srfi-18/srfi-18.html`.

[43] Marc Feeley Guillaume Germain and Stefan Monnier. Termite : a Lisp for Distributed Computing. 2nd European LISP and Scheme Workshop. `http://lisp-ecoop05.bknr.net/submission/19654.html`, July 2005.

[44] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5) :242–252, 2000.

[45] Multiple arcade machine emulator (mame). `http://mamedev.org/`.

[46] Simple directmedia layer (sdl). `http://www.libsdl.org/`.

[47] Open graphic library (opengl). `http://www.opengl.org/`.

[48] Langage de programmation lua. `http://www.lua.org/`.

[49] Langage de programmation python. `http://docs.python.org/reference/`.

[50] Naughty dog.

http://www.naughtydog.com/.

[51] Postmortem : Naughty dog's jak and daxter : the precursor legacy.

http://www.gamasutra.com/view/feature/2985/postmortem_naughty_dogs_jak_and_.php.

[52] Quantz.

http://www.quantzgame.com/.