Université de Montréal

Dévelopment de jeux vidéo en Scheme

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:

Dévelopment de jeux vidéo en Scheme

présenté par:

David St-Hilaire

a été évalué par un jury composé des personnes suivantes:

Mostapha Aboulhamid
président-rapporteur

Marc Feeley
directeur de recherche

Yann-Gaël Guéhéneuc
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 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 la réalisation de celui-ci. Plusieurs facteurs influencent ces différences. Par exemple, un langage à typage statique résultera en du code qui se doit d'être bien structuré, mais qui sera beaucoup moins malléable par la suite. Ainsi, afin d'optimiser les efforts développement pour un projet, une bonne analyse des besoins exprimés par le projet est de rigeur pour pouvoir faire le choix d'un langage répondant le mieux possible à ceux-ci.

En se basant sur les besoins et les contraintes énoncées 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 afin de permettre un développement efficace de jeux vidéo.

Ce chapitre donne une historique du langage Scheme afin de mettre en situation le langage et donné un aperçu de son origine. Par la suite, une brève explications de concepts de bases du langage est donnée. Plus d'informations sur les bases de Scheme peuvent êtres obtenues dans la littérature [6] [14]. Finalement, les particularités du langages sont mise en lumière dans le but de faire comprendre au lecteur quelles sont l'ampleur de celles-ci dans le but de développer des jeux vidéo.

[TODO: *d'autres idées ?*]

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 les racines commune à Scheme est les 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. Son histoire débute au MIT (*Massachusetts Institute of Technology*), dans le début des années 1960, lorsque John McCarthy publia un article décrivant ce langage [15]. 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 et le concept de ramasse miettes.

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. En 1984, Guy Steele publia un livre proposant d'unir ces dialectes dans le langage *Common LISP* [16]. Ce livre fut par la suite la fondation du standard créé pour le langage *Common Lisp* [17]. Il en résulte donc une spécification très complexe du langage qui s'est trouvé gonflé pour les multiples dialectes lui ayant donné naissance. Ainsi, Common Lisp possède beaucoup de bagage, mais il devient difficile pour un programmeur de pouvoir maîtriser tout les concepts de base de ce langage.

3.1.2 Avènement de Scheme

Peu avant la création de Common Lisp, Guy Steele et Gerald J. Sussman conçurent la spécification d'un nouveau dialecte du langage LISP appelé Scheme [18]. Ce dialecte était sensiblement différent des autres dialectes de LISP existant puisqu'au lieu d'ajouter des nouvelles composantes ou d'étendre le langage, il l'épura au maximum afin d'en extraire l'essence même. La spécification du langage étant écrite sur une trentaine de pages faisait contrastes aux spécifications des autres dialectes, notamment à Common Lisp qui vint peu après qui s'étalent sur plusieurs centaine de pages. Il en résultat que plusieurs implantations du langage apparurent grâce à la spécification beaucoup plus accessible.

Scheme devint ainsi populaire et est présentement l'un des deux dialecte les plus utilisé de LISP, en concurrence avec Common Lisp. On retrouve pas moins d'une cinquantaine d'implantation disponible, dont une douzaine sont considérées

comme étant des implantations majeures. Ces implantation majeures offrent non seulement plusieurs extensions du langage, mais aussi de très 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 apprécié car elle mène à utiliser ce qui est vraiment nécessaire et faire soi même ce qui n'est pas disponible, créant ainsi une bonne indépendance chez les programmeurs Scheme. De plus, le langage est accompagné d'une centaine librairies externes de traitement générique dénommées SRFI [19] (*Scheme Request For Implementation*).

3.1.3 Langages inspirés de Scheme

Plusieurs langages de programmation récent furent inspirés dans langages LISP. Pour la plupart, il s'agit de langages de script comme JavaScript [20], Ruby [21] ou Perl [22]. Ces languages vont notamment chercher l'aspect dynamique de LISP afin de pouvoir être interprétés. Ces langages utilisent aussi un ramasse miettes afin de simplifier leurs utilisation. Finalement, certains d'entre eux offrent des fonctions en tant que données de premières classe, permettant ainsi l'utilisation de la programmation fonctionnelle.

Ces langages demeurent par contre toujours des sous-ensemble, en terme de fonctionnalités, des langages LISP. Ils sont devenus populaires pour leurs 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 application très efficaces.

[TODO: *Java ? ?*]

3.2 Introduction au langage

Tout en bénéficiant du riche héritage des langages LISP, Scheme demeure un langage épuré et très simple à la base. Cette section vise à présenter les bases du langages 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 langages fournies par Gambit-C.

3.2.1 Syntaxe

La syntaxe de Scheme est une des caractéristique qui le distingue le plus des autres langages de programmation. En effet, la syntaxe utilisée est extrêmement simple. À la base, toute expression Scheme est une expression symbolique, nommée aussi *S expression*. Une S expression est une forme de donnée structurée récursive qui doit débuter par une parenthèse ouvrante, contient un nombre arbitraire de données qui sont soit des atomes ou des S expressions et qui doit se terminer par une parenthèse fermante. Les atomes sont des symboles, des nombres, des valeurs booléennes, etc.. La figure 3.1 donne un exemple très simple de S expression conte-

nant des informations sur les prix de la nourriture dans une épicerie.

((bananes 1.52)(bonbons 1/100))

FIGURE 3.1 – Exemple simple de S expression

En Scheme, ces S expressions sont considérées comme étant des listes chaînées où chacun des éléments de la liste chaînée contient la valeur associée à cet élément et un pointeur vers le prochain élément. Les listes sont ainsi implantées en utilisant des paires. Le point (.) est utilisé pour démarquer le premier élément d'une paire du deuxième. La figure 3.2 explicite la structure de liste de l'exemple de S expression provenant de la figure 3.1 en utilisant la notation de paires..

((bananes . (1.52 . ())) . ((bonbons . (1/100 . ())) . ()))

FIGURE 3.2 – Équivalence de la S expression de la figure 3.1 sous forme explicite de liste

Cette forme est strictement équivalente à celle donnée précédemment et correspond à la structure de donnée de base employé en Scheme, les listes. Toutes deux correspondent à des valeurs du langage, mais ne peuvent pas être données telles quel dans un programme car un programme Scheme est aussi une S expression qui correspond à *un appel de fonction* (ou à une forme spéciale du langage). Le premier élément de la liste doit être un symbole qui correspond à cette fonction ou cette forme spéciale et le restes des éléments sont les argument qui sont eux aussi des

programmes Scheme sous forme de S expressions. Conséquemment, une notation préfixe est utilisée en Scheme.

La distinction entre un appel de fonction et une forme spéciale est l'ordre d'évaluation des paramètres. Pour un appel de fonction, chacun des paramètres passés sont d'abord évalués et ensuite, la valeurs correspondante à leurs évaluation sont passées à la fonction appelée. Il s'agit donc d'un *passage par valeur*. En opposition, les formes spéciales possèdent des règles d'évaluation propres à elles. Les formes spéciales de bases de Scheme peuvent être étendu par des macros qui utilisent un passage de valeur par nom, où les paramètres sont passés directement à la macro comme étant des données Scheme. Le fonctionnement des macros Scheme est détaillé dans la section 3.4.

Il existe également quelques exception syntaxiques aux S expressions qui sont utilisées, entres autres, pour construire des données constantes. On utilise le caractère ' ou la forme spéciale *quote* pour signifier que la S expression subséquente est constante, *i.e.*qu'elle ne doit pas être considérée comme un programme Scheme, mais bien comme une valeur. L'utilisation du caractère è est strictement équivalente à l'utilisation de la forme spéciale correspondante. Une exemple d'utilisation est illustré dans la figure 3.3.

Après évaluation, ces deux morceaux de code retournent tous deux la valeur *((bananes 1.52)(bonbons 1/100))*.

```
'((bananes 1.52)(bonbons 1/100))
(quote ((bananes 1.52)(bonbons 1/100)))
```

FIGURE 3.3 – Exemples d'utilisation de la syntaxe *quote* et de la forme spéciale correspondante.

On retrouve une forme altéré du *quote* nommée *quasiquote* qui permet des évaluations partielles à l'intérieur d'une forme *quote* en utilisant le caractère `,` équivalent à la forme spéciale `unquote`. La figure 3.4 illustre un exemple d'utilisation.

```
`((bananes ,(+ 1.0 52/100)) (bonbons 1/100))
(quasiquote ((bananes (unquote (+ 1.0 52/100))) (bonbons 1/100)))
```

FIGURE 3.4 – Exemple d'utilisation des formes spéciales `quasiquote` et `unquote`

L'utilisation de la forme `unquote` permet donc ici l'évaluation de la S expression effectuant l'addition pour donner comme résultat la liste donnée dans la figure 3.1.

Malgré le fait que, pour certains, le parenthèsage des programmes Scheme peut sembler imposant, il en résulte un code *structuré* et *sans aucune ambiguïté*. Cette structure permet aussi de pouvoir utiliser très efficacement des outils de d'édition de S expressions, accélérant ainsi l'écriture de programmes.

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 bases requise 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 bases du langages sont tous disjoints entre eux (aucun n'objet ne peut appartenir a plus d'un de ces types, mais doit appartenir à un seul). Ces types sont vérifiés par les prédicats `boolean?`, `pair?`, `symbol?`, `number?`, `char?`, `string?`, `vector?`, `port?`, `procedure?`. Chacun de ces types sont documentés et possèdent des fonctions standards de création et d'utilisation.

Les valeurs booléennes sont dénotées par `#t` et `#f`. Par contre, le langage tire profit du dynamisme du typage et considère que toute valeurs différentes à `#f` est vraie.

Les paires, comme mentionné précédemment, sont les structures de données de bases en Scheme. La fonction de création d'une paire se nomme `cons` et les accesseurs aux premier et deuxième valeurs sont nommés respectivement `car` et `cdr`. La figure 3.5 illustre la création programmatique de la S expression de la figure 3.1.

Les symboles se sont spécifié uniquement par du texte dans le programme et sont utilisé comme données, mais surtout pour faire des liaisons dans le programmes.

```
(cons (cons 'bananes (cons 1.52 '()))
      (cons (cons 'bonbons (cons 1/100 '()))
            '()))
```

((bananes 1.52)(bonbons 1/100))

FIGURE 3.5 – Exemple de création programmatique de listes

En effet, les variables sont représentées par des symboles. Ainsi, pour utiliser un symbole comme une donnée (donc un symbol constant), la forme spéciale `quote` doit être utilisée. Aussi, plusieurs formes spéciales de liaisons sont spécifiée dans le standard. Les plus fondamentales étant `define` et `let`. La forme spéciale `define` permet la déclaration de variables globales ou locales permettant la récursion. La forme spéciale `let` permet la création de liaisons locales non-récursives. La figure 3.6 illustres de création de liaisons globales, locales et d'utilisation de symboles comme données. Le morceau de code présenté définit une fonction globale qui vérifie si tout les éléments d'une liste sont strictement inférieurs à 10. Ainsi l'appel `(drole-de-fonction (list 1 11 4))` s'évalue à la valeur *non*.

On y retrouve ainsi une spécification pour le calcul numérique qui offre la possibilité de faire du calcul sur nombres flottants, entiers, réels et même complexes.

L'utilisation caractères et les chaînes de caractères est très simple. Les caractères en tant que donné sont précédés par `#\` et les chaînes de caractères sont facilement convertibles en listes de caractères pour une manipulation plus « Schemiène »ou l'on peut accéder directement et modifier des caractères dans l'objet de type `string`.

```
(define (drole-de-fonction x)
  (let ((premier (car x))
        (deuxieme (cdr x)))
    (if (< premier 10)
        (if (null? x)
            'oui
            (drole-de-fonction deuxieme))
        'non)))
```

FIGURE 3.6 – Exemple de créations de liaisons et d'utilisation de symboles comme données

[TODO: *vectors*]

[TODO: *ports*]

[TODO: *procedures, apply, appels terminaux*]

3.2.3 Extensions présentées par Gambit-C

3.3 Programmation fonctionnelle

Le langage Scheme est à la base un langage fonctionnel. Ces derniers sont caractérisés 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 facile à comprendre, ce qui attire un grand nombre de chercheurs à contribuer au développement et l'utilisation

de tels langages dans le milieu commercial.

Une caractéristique importante des langages de programmation fonctionnelle est la présence de fonctions de premier ordre, *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 lorsqu'elles sont créés, elle conserve l'environnement d'exécution dans lequel elle se trouvait lors de sa création.

Ce concept ouvre la porte à la création d'abstractions très intéressantes : les fonctions d'ordre supérieures. Celles ci reçoivent des fonctions en paramètre les manipulent en souvent en retournent de nouvelles. La figure 3.7 illustre un exemple d'écriture et d'utilisation de fonctions d'ordre supérieurs. La premières fonctions 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 créer une nouvelle à partir de celle donnée où l'on aura retiré 1 à chaque élément.

L'utilisation de fonctions d'ordre supérieure permet l'abstraction d'algorithmes de manière très générique. Une autre fonction d'ordre supérieur très utilisée est la

```
(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 3.7 – Exemple de création et d'utilisation de fonctions d'ordres supérieures.

fonction `foldl` qui permet d'utiliser chacun des éléments d'une liste pour calculer un résultat unique. La figure 3.8 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 3.8 – Fonction d'ordre supérieure `foldl`.

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 la création des premiers ordinateurs. C'est dans les années 1936 que nul autre qu'Alonzo Church a introduit le formalisme mathématique de définition de fonctions mathématique dénommé lambda calcul non typé [23]. Ce formalisme introduit des notations afin de définir des fonctions anonymes qui sert de base aux langages de programmation fonctionnels. La figure 3.9 illustre la définition de la fonction `flip` définie en Scheme dans la figure 3.7.

$$\lambda f \rightarrow \lambda y \rightarrow \lambda x \rightarrow fxy$$

FIGURE 3.9 – 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.

Comme déjà mentionné dans la section 3.1, le lambda calcul a donné naissance une famille de langages dont la racine est le langage LISP. Parmi cette famille, les langages ML [24] et Haskell [25] sont parmi les plus connu 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 par-

tage 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.

Ainsi, les l'utilisation de fonctions en tant que données de premières classes permet de pouvoir faire de puissantes abstractions algorithmiques. De plus, l'utilisation des paradigme 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 ensembles, 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 sources. 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 autre dans le langage C. Ce type de macro, quoi que très simple, permet de faire déjà beaucoup d'abstractions allant à des abstractions d'optimisations à des expansions conditionnelles de code permettant de développer des librairies sur

plusieurs plates-formes. La figure 3.10 illustre des exemples de macros C. Dans cet exemple, on défini la macro `swap` qui interchange la valeur de deux variables entre elles. La fonction main retournera 1 si et seulement si le symbole `__WEIRD_OS__` est défini, et retournera 0 sinon. On peut imaginer un système d'exploitation fictif qui s'attend à une valeur de statut de terminaison de processus qui doit être égale à 1 pour signaler la terminaison normale d'un processus. Puisque normalement, le statut de terminaison normal d'un processus est 0, on utilise une expansion conditionnelle à la présence d'un symbole qui ne devrait qu'être présent que lorsque l'on se trouve dans ce système fictif.

```
#define swap(x,y) do { int tmp; tmp = x; x = y; y = tmp; } while(0)
int main(){
  int x = 0;
  int y = 1;
#ifdef __WEIRD_OS__
  swap(x,y);
#endif
  return x;
}
```

FIGURE 3.10 – 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 sources 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 difficile à trouver car ils ne surviennent que le programme est compilé, bien après l'expansion des macros. Il en résulte que le code compilé ne contient aucune trace des 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.

La syntaxe préfixe extrêmement simple de Scheme donne la chance au langage de posséder la même syntaxe que les structures de données de base de ce dernier. Ceci ouvre la porte à implanter un système de macro très puissant, un système de macro procédural.

Plusieurs systèmes d'expansions de macro sont disponible en Scheme. Le plus simple d'entre eux est celui qui provient directement de LISP, 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 mais convaincant de macro Scheme est illustré dans la figure 3.11. Cette macro sert à définir des variables constantes correspondant à des

nombres de la série de Fibonacci. Le nombre `n` passé à la macro est utilisé pour calculer le n^{e}nombre de Fibonacci *durant l'expansion de la macro* pour générer le code présenté dans la figure 3.12. Ainsi, le code résultant de l'expansion est par la suite évalué durant l'exécution du programme, et il est possible de faire référence à la variable `fib10`.

```
(define-macro (define-fib n)
    (letrec ((fib (lambda (n) (if (< n 3) 1 (+ (fib (- n 1)) (fib (- n 2)))))))
      `(define ,(string->symbol (string-append "fib" (number->string n)))
         ,(fib n))))
(define-fib 10)
```

FIGURE 3.11 – Exemple simple de macro Scheme.

```
(define fib10 55)
```

FIGURE 3.12 – Résultat de l'expansion de la macro présentée dans la figure 3.11 avec 10 comme paramètre.

Cet exemple très simple n'a pas utilisé le passage de paramètre par nom dont disposent les macros Scheme, *i.e.*que la macro de la figure 3.11 n'a pas utiliser le fait que le code passé en paramètre est une donnée Scheme, puisqu'un nombre s'évalue en lui même en Scheme. Il est intéressant de noter qu'un appel à cette macro comme `define-fib (+ 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 l'évaluation de l'addition ne sera fait que durant l'exécution du

programme.

Un exemple simple de macro tirant profit du fait que l'argument est passé par nom est illustré dans la figure 3.13.

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

2

FIGURE 3.13 – Exemple simple de macro utilisant le passage par nom à profit.

Cette macro très simple utilise le passage par nom pour permettre d'incrémenter la valeur d'une variable. Un exemple plus complexe de macro utilisant le passage par nom est donné dans la figure 3.14. Cette macro permet d'augmenter le langage Scheme d'une nouvelle forme spéciale qui correspond à une forme simple d'itération.

```
(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 3.14 – Exemple de macro ajoutant une nouvelle forme spéciale au langage.

Cette macro comporte, par contre 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 Variables*. 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 à la 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 expansées de macro assurant l'hygiène des macros, dont entre autres la forme spéciale `syntax-rules` [26]. La réécriture de la macro `for` en utilisant un système de macro hygiénique est donnée dans la figure 3.15. 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 d'elle dans des conditions potentiellement problématiques.

Malgré le 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, par exemple, être intentionnelle. Aussi, le problème est facilement résolu avec l'utilisation de symboles uniques.

```
(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 3.15 – Macro `for` hygiénique

Ainsi, en utilisant les macros de Scheme, il est possible d'ajouter des nouvelles formes spéciales au langages. On peut ainsi créer des langages spécifiques au domaine (*Domain specific languages*), en se créant un langage complet ou partiel au dessus de Scheme à l'aide des macros. Ces langages sont conçus dans le but de résoudre un problème précis et l'utilisation de macros permet pouvoir exprimer ce langage en Scheme, sans aucun coûts additionnels.

3.5 Continuations

L'état d'un calcul en cours d'exécution est souvent implanté ou illustré par une pile d'exécution. C'est le modèle d'implantation utilisé en C et dans la plupart des langages de programmation. Ainsi, l'exécution d'un appel de fonction empile l'adresses de retour et utilise le dessus de la pile pour les variables locales à la fonction. Lorsque la fonction termine, l'adresse de retour est dépilée et le contrôle

revient à l'endroit de l'appel de fonction. L'environnement d'exécution de cet appel est contenu partiellement dans l'état de la pile et devient donc perdu lorsque l'appel se termine.

En Scheme, le modèle d'exécution, quoi que similaire, est sensiblement différent. L'évolution de l'exécution d'un programme est implanté par des continuations. Ces dernières sont des fermetures qui représentent le reste du calcul à faire à un moment du calcul donné. La continuation initiale d'un programme est la terminaison de ce dernier, puisque après avoir exécuter le programme, il ne restera plus rien à faire.

Les figures 3.16 et 3.17 illustrent la continuation d'un appel de fonction dans un calcul très simple. Ainsi, après avoir exécuté l'appel de la fonction `f`, le reste du calcul à effectuer peut être représenté par une fermeture attendant le résultat de l'appel de fonction en entrée.

```
(+ a (- [f x y] 2))
```

FIGURE 3.16 – Exemple simple de calcul en Scheme

```
(lambda (res-f) (+ a (- res-f 2)))
```

FIGURE 3.17 – Continuation de l'appel de la fonction `f` présenté dans la figure 3.16

Le langage Scheme offre la possibilité de réifier la continuation d'un calcul donné par l'entremise de la forme spéciale `call-with-current-continuation` souvent disponible aussi via `call/cc`. La figure 3.18 donne un exemple de la réification

d'un calcul similaire à celui présenté dans la figure 3.16. 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.

```
(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 3.18 – Réification d'une continuation à l'aide `call/cc`

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

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 3.19 illustre une utilisation de la fonction d'ordre supérieur `foldl` présentée dans la figure 3.8 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 3.19 – 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 3.20 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 3.20 – Utilisation d'une continuation pour échapper au flot de contrôle.

3.5.1.2 Recherche par retour sur trace

Un autre exemple d'utilisation intéressant de continuations est dans la création d'un système de recherche par retour sur trace (*backtracking*). La figure 3.21 illustre un exemple simple de recherche d'un triplet de nombres entiers tel que $x^2 = y^2 + z^2$.

```
(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 3.21 – Exemple de recherche par retour sur trace

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`.

3.5.2 Forme d'écriture de code en CPS

En Scheme, on distingue deux types d'appels de fonction, les appels terminaux et les appels non-terminaux. Un appel terminal est un appel dont la continuation est la même que la continuation de la fonction dans laquelle il se trouve, sinon, il s'agit d'un appel non-terminal. La figure 3.22 illustre l'implantation traditionnelle de la fonction factoriel en Scheme. On constate que cette fonction contient un seul

appel terminal : l'appel à la fonction `*`. Cet appel possède la même continuation que celle de la fonction `fact` puisque le calcul qui reste après cet appel est en fait la suite du calcul à l'appel original de `fact`.

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

3628800

FIGURE 3.22 – Exemple de fonction contenant des appels terminaux et non-terminaux.

Les appels terminaux sont sujet à une optimisation intéressante. L'optimisation d'appels terminaux consiste à changer la continuation d'un appel terminal directement par la continuation de la fonction dans laquelle se trouve cet appel. Dans le cadre de l'exemple de la fonction `fact` la continuation de l'appel à `*` dans la figure 3.22 est la même que celle de la fonction factoriel. Supposons que nous appelons cette continuation `k`. Ainsi, la continuation de `*` pourrait être représentée par la fonction `(lambda (r) (k r))`. L'optimisation d'appels terminaux revient à utiliser directement `k` comme continuation à la multiplication. Ainsi, le flot de contrôle du calcul n'a plus besoin de revenir dans `fact` lorsque la multiplication est terminée, mais peut continuer directement le calcul. Cette optimisation est requise pour toute implantation de Scheme par la spécification du langage.

Il existe un style d'écriture de code appelé *Continuation Passing Style* ou style

en forme CPS qui consiste à expliciter toutes les continuations dans un programme. Une propriété intéressante de ce style d'écriture est que tous les appels sont des appels terminaux. La figure 3.23 contient la réécriture de la fonction factoriel présentée dans la figure 3.22. On peut constater que cette réécriture a appliqué l'optimisation d'appels terminaux.

```
(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 3.23 – Forme CPS de la fonction factoriel

Cette transformation est souvent utilisée par les implantations de Scheme car elle facilite l'implantation de l'optimisation d'appels terminaux en rendant tous les appels comme terminaux. Ce style de programmation permet aussi d'utiliser de manière élégantes la puissance d'abstraction des langages fonctionnels en permettant de diviser le calcul en petites parties facilement interchangeable et modulaires.

3.6 Gestion mémoire automatique

L'écriture de code effectuant gestion de la mémoire d'un programme à 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 reliées à une mauvaise gestion de la mémoire sont classifiées 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 accessible par le programme, mais dont l'espace mémoire correspondant a été récupéré par le système d'exploitation.

Dans les deux cas, ces erreurs sont très problématiques. C'est probablement 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 automatique, appelles aussi ramasses miettes, apparurent avec le langages 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 se temps dépend grandement de la technique de gestion utilisée. Aussi, les algorithmes de parcours de mémoire se doivent d'être

conservateurs afin d'assurer de ne pas causer de pointeurs fous. 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 des fuites de mémoires lorsqu'une gestion manuelle de mémoire est utilisée.

3.6.1 Survol des techniques

Plusieurs techniques de gestion de mémoire automatique 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 [27] [28] [29].

Une des techniques les plus simples 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, tous 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.

Une autre technique classique se nomme *Stop And Copy* [30]. De manière similaire au *Mark And Sweep*, cette technique parcours 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 nouvelles zone mémoire, résultant en une désallocation implicite des espaces inutilisées 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.

Une amélioration notable de la technique du *Mark And Sweep* a donné naissance aux ramasses miettes générationnels [31]. 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 récupérations et lorsque celle-ci devient remplie, elle déclenche une récupération dans la génération au dessus d'elle. Cette algorithme est beaucoup plus efficace, car les objets récemment alloués ont une petite durée de vie moyenne, contrairement aux objets déjà retenus qui eux ont une meilleur chance d'être toujours 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.

Afin de remédier au problème d'arrêt complet de l'exécution du programme, deux nouveaux types de ramasses miettes ont été développées : les ramasses miettes incrémentaux (ou temps-réels) [32]. Ceux-ci utilisent un algorithme de marquage à trois couleurs de la mémoire afin de permettre de faire partiellement le travail de récupération et de poursuivre l'application dans le but de continuer plus tard. Il en résulte donc que les pauses dues à la récupérations peuvent être contrôlées et de temps constant.

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 tout les nouveaux langages de programmations.

Le fait que l'exécution du programme puisse être arrêtée 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 continues. Ainsi, une attention particulière devra être apportée sur l'utilisation de la mémoire d'avoir de trop grandes pauses de récupération.

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, la principale différence entre les deux approches est le moment où les erreurs de programmation apparaissent.

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 vérification de types. Une particularité des systèmes typés statiquement consiste à avoir des structures de code plus rigides, demandant possiblement beaucoup d'efforts pour être modifié.

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 de légères pertes de performances puisque les vérifications de effectuées pendant l'exécution du programme. Par contre, des erreurs apparaissant durant l'exécution peuvent être beaucoup plus faciles à corriger, si le système est muni de bon système de déboggage. Aussi, un typage dynamique permet au langage de pouvoir être interprété en plus de pouvoir être compilé. 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. Cela permet de pouvoir tester facilement les modules écrits, de manière externe au système dans lequel il se trouve. Les cas extrêmes peuvent ainsi facilement être vérifiés et donc donner une bonne assurance sur le fonctionnement 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'environnements de fermetures et la possibilité de rétablir ces environnement et de pouvoir exécuter du code arbitraire dans celui-ci afin 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 le déboggage. Par exemple, James Long a utilisé ce système afin de pouvoir débugger des processus Scheme exécuté 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 james long et tcpip ?*]

3.8 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 fonctionnel du langage et de sont puissant système de macro permettrait de faciliter les abstractions nécessaires à une bonne modularisation du code source et par la possibilité de création de langages spécific au 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 ramasse miettes évite la présence de beaucoup d'erreurs potentielles de gestion de mémoire, mais pourrait causer des problèmes de ralentissement lors de l'exécution de tels 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 leurs permet de se bâtir des outils personnalisés pour mieux répondre leurs besoins.

Un paradigme absent du langage Scheme est celui de la programmation orientée objet. Ce paradigme implique l'utilisation de 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 demeurées 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'états qui évoluent 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 neuvième SRFI [33] (*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 de manière hiérarchique simple. L'exemple de la figure 4.1 illustre l'étendu de ce qui peut être tiré de ce système. On arrive donc à utiliser le polymorphisme pour l'accès aux membres des objets issus de la même hiérarchie.

```
(define-type point x y)
(define-type point x y extender: define-point-type)
(define-point-type 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 #135 x : 4 y : 6>

FIGURE 4.1 – Exemple de création d'objets hiérarchiques et de polymorphisme simple dand Gambit-C

Ainsi, le système Gambit-C nous permet d'utiliser des concepts de la programmation orientée objet, mais d'une manière très limitée. Cette approche ne fournie aucune possibilité de créer des méthodes qui sont au coeur du réel avantage qu'apporte le polymorphisme.

L'approche plus traditionnelle quant à la déclaration de méthodes de classes

est celle utilisée par le langage de programmation Java [34]. Cette approche lie les méthodes à une et une seule classe. Les appels de ces méthodes possèdent ainsi un argument caché (souvent nommé *self*) 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 plusieurs objets différents. L'exemple de la figure 4.2 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 manuellement. L'exemple illustré demeure encore très simple puisqu'il s'agit de *dispatch* double sur deux types, mais pourrait devenir très complexe en s'il s'agissait de faire une discrimination sur plus d'objets.

Le langage Common Lisp [16] possède un système orienté objet, le Common Lisp Object System [35] (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 [36]. 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 dont celles-ci possèdent des spécifications optionnelles de types pour ses arguments. Lorsqu'un appel de fonction est fait sur une

```
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 4.2 – Exemple d'utilisation de méthodes à la manière traditionnelle (en Java)

fonction générique, le système décide alors de l'instance de la fonction générique la meilleure à utiliser en fonction des arguments actuels utilisés dans l'appel, des types de ces arguments et de leur nombre. On obtient donc un système très dynamique effectuant du *dispatch* multiple. La figure 4.3 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.

```
(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))
        (print (add p1 p2)))
```

2
3

FIGURE 4.3 – Exemple de *dispatch* multiple écrit en CLOS.

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.

Il serait donc très intéressant de pouvoir utiliser la programmation orientée objet afin de pouvoir écrire des jeux vidéo en Scheme. Un tel système permettrait de facilement abstraire les comportements communs à plusieurs types d'objets tout en gérant de manière générique leurs états. Ces problèmes sont très fréquents dans un jeu vidéo où il existe souvent une grande variété d'objets similaires ayant un comportement commun. L'utilisation de fonctions génériques permettrait ainsi de pouvoir facilement modulariser ces comportements.

Plusieurs système objets sont déjà disponible pour des système Scheme, dont

la plupart sont inspirés de la puissance expressive de CLOS. Pour Gambit-C, on retrouve entre autre le système orienté objet Meroon [37]. Ce système semble offrir de bonnes performances. Par contre, il implique plusieurs limitations, dont le fait que la hiérarchie de classe est limitée à un héritage simple. Aussi, il n'est pas possible d'allouer des attributs de classes, communs à toutes les instances. OOPS [38] est un autre système objet disponible pour Gambit-C. Celui-ci est beaucoup plus générique que Meroon, mais après essaie à démontré des sérieux problèmes de performances, ce qui ne peut pas être acceptables dans un jeu vidéo.

Ainsi, un système d'objet a été écrit afin 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 système et les motivations derrière les choix fait pour le développer sont décrits 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 [35] et possède la plupart des propriétés intéressantes de ce dernier. On retrouve entre autre un 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 apporté aux performances, tout spécialement à l'accès aux membres d'instances, puisque ces opérations sont très fréquentes dans un jeu vidéo.

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 4.4.

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

FIGURE 4.4 – 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 4.5 et 4.6 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 avec `define-type`.

```
(define-type point x y extender: define-point-type)
(define-point-type circle radius)
(let ((p1 (make-point 1 2)) (p2 (make-circle 3 4 5)))
  (sqrt (+ (expt (- (point-x p2) (point-x p1)) 2)
           (expt (- (point-y p2) (point-y p1)) 2))))
```

2.8284271247461903

FIGURE 4.5 – Utilisation d'objets créés par la forme spéciale `define-type` de Gambit-C.

```
(define-class point () (slot: x) (slot: y))
(define-class circle (point) (slot: radius))
(let ((p1 (make-point 1 2)) (p2 (make-circle 3 4 5)))
  (sqrt (+ (expt (- (point-x p2) (point-x p1)) 2)
           (expt (- (point-y p2) (point-y p1)) 2))))
```

2.8284271247461903

FIGURE 4.6 – Utilisation simple du système objets pour une utilisation similaire à celle pouvant être fait avec la forme `define-type`.

Les fonctions utilitaires générées par la création 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.
- `(point-x <instance>)` : Assesseur 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>)` : Fonctions de changement de l'état 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 allouer des attributs d'instances, il est possible d'allouer 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 classes fera en sorte que chaque sous-classe possède sa propre instance de l'attribut, permettant ainsi des personnalisations des instances de cette classe face à cet attribut.

Puisque ces attributs sont communs, ils ne nécessitent pas d'instance pour être accédés ou modifiés, mais ces fonctions d'accès ou de modification de valeur d'attributs de classes permettent de prendre tout de même une instance pour pouvoir utiliser ces fonctions de manière dynamique, ou encore, polymorphique. La figure 4.7 illustre ce processus.

En plus des créateurs d'instances à la `define-type`, le système objet permet la

```
(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 4.7 – Exemple d'utilisation dynamique d'attributs de classes.

définition d'un ou de plusieurs constructeurs pour la création de nouvelles instances simplifiée. 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 4.8 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))))
  (pp (toto-t obj1))
  (toto-t obj2))
```

```
1
2
```

FIGURE 4.8 – Utilisation générique de constructeurs

Finalement, l'utilisation de crochets (*hooks*) sur les fonctions assesseur et de modifications d'états d'attributs est possible en ajoutant les des mots clés (*keywords*) à l'intérieur des attributs désirés. Il est possible de définir des crochets en lecture ou en écriture avec respectivement les mots clés `read-hooks:` et `write-hooks:`. Ces mots clés doivent être le premier élément d'une liste qui contiendra tous les fonctions de crochets désirées. Des crochets d'attributs d'instance doivent prendre en paramètre une référence vers l'instance en question et la valeur (ou nouvelle valeur) de l'attribut. Les crochets d'attributs de classes ne doivent pas prendre d'instances en paramètres.

[TODO: *Encore un exemple, ou bien est-ce suffisant ?*]

4.1.2 Définition de fonctions génériques

Toujours avec une forte inspiration du modèle éprouvé du système objets de Common Lisp [35], 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 souhaitions que notre système d'objets puisse offrir le plus de puissance expressive que possible aux programmeurs de jeux vidéo, mais surtout puisque plusieurs problèmes se retrouvant dans le développement jeux vidéo peuvent être exprimés de manières très concise avec des fonctions génériques.

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'ins-

tance 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 4.9 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)))
    (pp (add 10 11))
    (pp (add 10 11 12))
    (pp (point-x (add p p)))
    (pp (add p l)))
```

21
33
2
6

FIGURE 4.9 – 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.

- `(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>`.

- `(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>` peuvent être des clauses `match-value` ou `match-member-value`.

- `(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>` peuvent être des clauses `match-value` ou `match-member-value`.

Un exemple simple de discrimination par valeur de paramètre est donné dans la figure 4.10. 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.

```
(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)))
  'Error!)
(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))
(pp (point-x (div (new point 1 2) 2)))
(pp (div (new point 1 2) 0))
(pp (eq? (div (new point 0 0) 5) (point-zero)))
```

1/2
Error !
#t

FIGURE 4.10 – Exemple de discrimination d'instance de fonctions génériques par la valeur des arguments.

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.

Puisque le système d'objet supporte un 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 4.11. 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 utilisable pour une instance de la classe

`colored-circle`. L'algorithme discrimine alors de la manière suivante : 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 super 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.

```
(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)) (pp 'colored))
(define-method (test (x circle)) (pp 'circle))
(test (new colored-circle 1 2 3 'red))
```

circle

FIGURE 4.11 – Exemple illustrant l'algorithme de sélection d'instances de fonctions génériques.

Finalement, la fonction `call-next-method` est disponible dans le contexte du corps d'une instance de fonction générique pour appeler la prochaine instance la plus appropriée pour les arguments actuels passés. Si une instance précise doit être appelée, il est également possible d'utiliser un *cast* pour effectuer l'appel spécifiquement désiré. 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 de ces deux propriétés est illustré dans la figure 4.12. La première création d'instance de la classe `blue-circle` n'utilise que 2 arguments. L'appel au super constructeur avec *cast* est direct. Par contre, puisque la super classe `circle` de la classe `blue-circle` est plus spécifique, elle est considérée en premier pour l'appel à `call-next-method`. Cette classe possède bien un constructeur à deux paramètre de type `*`, et donc c'est cette instance qui se trouve à être appelée par cet appel. Cette instance utilise elle même un appel à `call-next-method`. Puisque la classe `circle` ne possède qu'une seule super classe et que les arguments du constructeur de celle-ci correspondent au constructeur de `circle` actuellement utilisé, c'est alors ce super constructeur qui est utilisé. Pour la deuxième construction d'objet, trois arguments sont utilisés, mais aucun constructeur de la classe `blue-circle` ne possède 3 arguments. Ainsi, c'est le constructeur le plus spécifique qui est utilisé, soit celui de la super classe `circle`. Il en résulte donc que le champs `color` de ce deuxième objet ne sera pas initialisé par cette instantiation.

4.1.3 Fonctions et formes spéciales utilitaires

Plusieurs fonctions et formes spéciales utilitaires sont disponible 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 :

```
(define-class point () (slot: x) (slot: y)
  (constructor: (lambda (self x y)
                  (pp 'point-constructor)
                  (point-x-set! self x) (point-y-set! self y))))
(define-class circle (point) (slot: radius)
  (constructor: (lambda (self x y)
                  (pp 'circle-constructor-1)
                  (circle-radius-set! self 1.0)
                  (call-next-method)))
  (constructor: (lambda (self x y r)
                  (pp 'circle-constructor-2)
                  (circle-radius-set! self r)
                  (init! cast: '(point * *) self x y))))
(define-class colored () (slot: color)
  (constructor: (lambda (self color)
                  (pp 'colored-constructor)
                  (colored-color-set! self color))))
(define-class blue-circle (circle colored)
  (constructor: (lambda (self x y)
                  (pp 'blue-cicrle-constructor)
                  (init! cast: '(colored *) self 'blue)
                  (call-next-method))))
(new blue-circle 1 2)
```

blue-cicrle-constructor
colored-constructor
circle-constructor-1
point-constructor
<blue-circle-obj x : 1 y : 2 radius : 3 color : blue >

```
    (new blue-cicrle 1 2 3)
```

circle-constructor-2
point-constructor
<blue-circle-obj x : 1 y : 2 radius : 3 color : undefined ! >

FIGURE 4.12 – Exemple d'utilisation de *cast* et de `call-next-method` avec des fonctions génériques.

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.

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.

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 de constructeurs 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 4.13 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 4.13 – Exemple d'utilisation des formes spéciales fournies par le système objet.

Ainsi, l'introspection demeure très limitée en comparaison à celle rendue disponible par le protocole de méta objet de CLOS, mais elle demeure tout de même fonctionnelle.

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 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.

[TODO: *Mettre en annexe le code source et le lie ici ?*]

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 les 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 macro.

En effet, le polymorphisme d'attributs est possible grâce à une indirection pour l'accès aux attributs. Malgré le fait qu'une telle indirection ralentie légèrement l'accès aux attributs, cette dernière permet aussi d'avoir des instances de taille optimales, *i.e.*qu'il n'y a pas de cases vides dans les instances d'objets.

La conséquence de l'implantation du polymorphisme d'accès aux attributs par une indirection est que la taille des descripteurs de classe augmente linéairement en fonction du nombre d'attributs des classes déjà définies lors de la création de cette nouvelle classe. Par contre, puisqu'il y a généralement beaucoup plus d'instances de classes que de classes en elles même, cette approche semble un bon compromis

entre une optimisation de l'utilisation de la mémoire et du temps requis pour l'accès aux attributs. La figure 4.14 illustre les structures utilisées pour effectuer cette indirection sur l'accès aux données durant l'exécution.

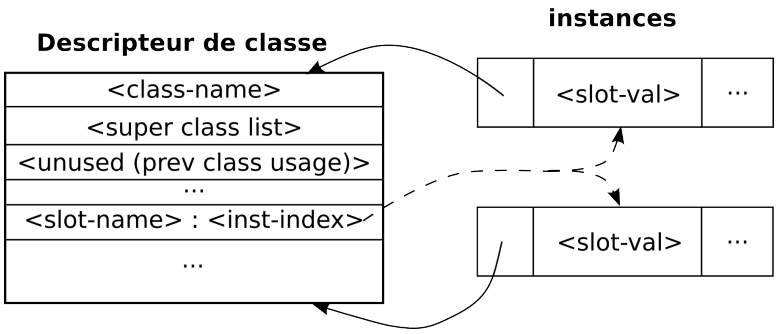


FIGURE 4.14 – Illustration des structures de données utilisées pour implanter l'exécution des classes et de leurs instances.

Ainsi, l'endroit pour trouver l'index d'instance d'un champ donné *se trouve toujours au même endroit* dans tous les descripteurs de classes. C'est ce qui permet de pouvoir utiliser les fonction d'accès à un membre même avec une instance d'une sous-classe. Les descripteurs de classes, tout comme les instances, sont implantés avec des vecteurs Scheme afin d'avoir un accès rapide aux champs de ceux-ci.

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 décrite. Cette information 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 *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 indexes associés à leurs attributs. Ces indexes correspondent à l'endroit dans les descripteurs de classes où l'on trouvera les indexes d'indirection vers les instances durant l'exécution.

Lorsque les nouveaux attributs (attributs non-hérités) sont inclus dans descripteurs de classe de l'exécution, un compteur global (expansion macro) est utilisé pour obtenir l'index du nouveau champs dans le vecteur correspondant à ce descripteur. Un deuxième index, local celui-ci, sera utilisé pour trouvé l'index de la position de l'attribut dans l'instance. Ainsi, l'index global pointe vers l'endroit où placer l'index d'instance dans le descripteur de classe de l'exécution. La création du descripteur de classe pour l'expansion macro est donnée dans la figure 4.15. Ce même objet sera simplifié et utilisé comme descripteur de classe durant l'exécution.

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 illustrant les structures utilisées lors de l'exécution du système. On constate que le premier élément du vecteur correspondant à l'instance créée est aussi un vecteur, qui lui, correspond au descripteur de la classe `toto`. Le premier élément de ce descripteur est le nom de la classe, le deuxième est la liste des super

```
(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 de l'expansion macro.

classes. Par la suite viennent les indexes des attributs dans l'instance. Par contre, puisque la classe `blub` a été préalablement définie, le premier conteneur d'indirection a été reservé pour le champs `b` de la classe `blub`. Ainsi, l'indirection vers le champs `titi` se trouve dans le quatrième éléments ($index = 3$) du descripteur. L'affichage de la fonction d'accès au champs `titi` confirme bien ce fait et montre que la valeur obtenue, en l'occurance 1, est utilisée comme index pour obtenir la valeur du champs dans l'instance.

Deux constructeurs d'instances sont finalement générés avec les accesseurs. Le premier constructeur (`make-<class-name>`) construit et initialise *tous* les champs

```
(define-class blub () (slot: b))
(define-class toto () (slot: titi) (class-slot: tutu))
(toto-tutu-set! 'allo!)
(new toto 12)
```

#(#(toto () unknown-slot 1 allo !) 12)

```
(pp toto-titi)


(lambda (#:obj445)
  (vector-ref #:obj445
              (vector-ref (instance-class-descriptor #:obj445)
                          3)))
```

FIGURE 4.16 – Exemple concret de structure d'instance, descripteur de classe et de fonction d'accès à un attribut par indirection.

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 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`.

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 macro 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.

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*

```
(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!`

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 réalisé ce *dispatch* polymorphique.

Finalement, la fonction `call-next-method` fonctionne grâce au mécanisme de variables à portée dynamique fourni par Gambit-C. Une telle variable, (`___call-next-method`), est utilisée pour contenir un appel à la fonction générique courante avec un *cast* spécial des arguments ou chaque argument est une liste des super classes des types de l'instance actuellement utilisée. Ainsi, lorsque cette appel est fait, l'on cherchera l'instance polymorphique la plus spécifique pour ces types en fonction des

```
(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

arguments spécifié lors de l'appel courant. Ainsi, comme désiré, l'instance de cette fonction générique la plus spécifique et différente de l'instance courante sera appelée.

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 Conclusion

Ainsi un système objet complet a été développé dans le but d'étendre le langage Scheme et d'y inclure les 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 gardant 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'objets sont de tailles optimales, et ne requièrent qu'une indirection supplémentaire afin de pouvoir accéder aux champs de ceux-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 objets 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 [39] (*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 [40] 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.

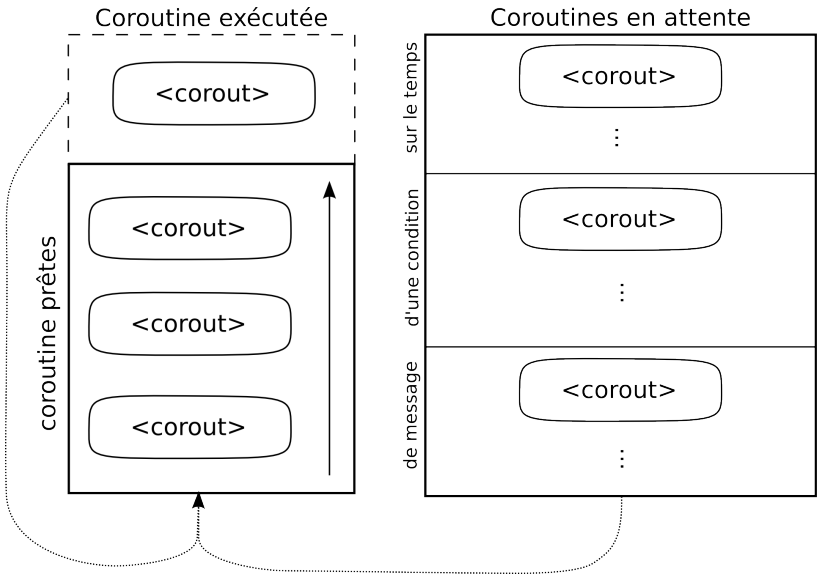


FIGURE 5.1 – Architecture globale du système de coroutine

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 3.8).

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 [40]. 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 [41]. 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 ne suffit que 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 [40], 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 [41].

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 de manière à résoudre les problèmes reliés à leur développement de manière itérative. 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 abaisse la difficulté du développement de jeux plus complexes.

Le premier jeu choisi pour la résolution des problèmes de bases 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 usagers 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.

Par la suite, le jeu *Lode Runner* (1983) a été développé. Ce dernier date de

l'époque des premiers ordinateurs personnels et est relativement 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 traité, introduit la notion d'intelligence artificielle et possède des entités possédant des états beaucoup 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.

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, dans un tel cas, ils descendent 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.

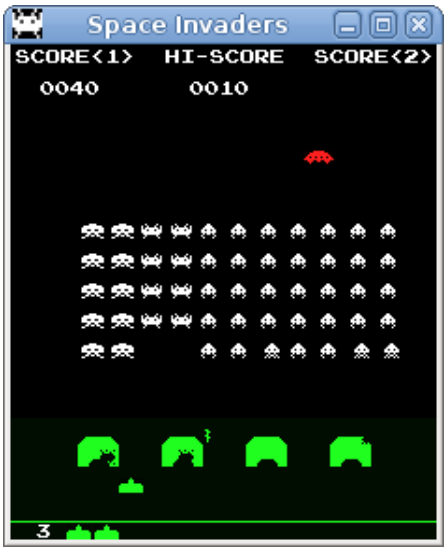


FIGURE 6.1 – Capture d'écran du jeu *Space Invaders*

L'objectif visé en développant *Space Invaders* est très 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 répondre à certaines lacunes que présentait la première version du jeu. Pour ce, le développement du système objet 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 a été de tenter de fusionner le système objet et le système de coroutine ensemble afin que tous les objets du jeu soient eux même 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

Une première version du jeu *Space Invaders* a été écrite dans le but de rencontrer les problèmes reliés au développement de jeu vidéo et les résoudre en utilisant le langage Scheme. Ainsi, après l'analyse de la version arcade du jeu disponible sur l'émulateur MAME [42], plusieurs problèmes potentiels ont été rencontrés :

- Rendu du jeu
- Flot du contrôle du jeu
- Structures de données
- Détection et résolutions de collisions
- Animations

- Parties multi-joueurs

Afin de pouvoir effectuer le rendu du jeu, une librairie de lecture et chargement d'image a dû être écrite. Le choix de l'écriture de cette librairie, plutôt que d'utiliser une librairie C déjà existante pour faire ce travail réside dans le but de demeurer indépendant d'autant de dépendances externes que possibles et d'avoir autant de code Scheme impliqué dans le jeu que possible. Par contre, la librairie SDL [43] a été utilisé comme système de gestion d'entrés/sorties et de fenêtrage. Le système de rendu utilisé est *OpenGL* [44] afin de pouvoir profiter d'un bonne portabilité et de permettre facilement d'ultérieurement utiliser les travaux effectués pour faire des jeux vidéo produisant rendu tri dimensionnel.

Ainsi, la librairie de lecture et de chargement d'images s'occupe de lire des images en format textuel *ppm*, qui est l'un des format les plus simple pour d'écrire 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 toutes les garder de manière cohésive dans une même image. Un fichier descriptif de la fonte, sous formes de S expressions doit être présent afin de permettre une correcte interprétation 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* contant la description `((colors: (white green red)) (chars: (0 1)))`.

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

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

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

Cette dernière effectue la génération de code permettant la lecture et le chargement de la font spécifiée, à condition que cette fonte soit symétrique, *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 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.

Le flot de contrôle d'exécution dans ce jeu fut expérimental dès le départ. Dans cette première version, 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

considéré comme une série d'événements discrets. Par exemple, le laser avance, le joueur se déplace à droite, le vaisseau ennemie explose, etc... Ainsi, un petit système de simulation par événements discrets à été développé et utilisé. La figure 6.3 illustre des événements qui sont ordonnancés au tout début d'une partie. Un événement est une fonction ne prenant aucun argument (*thunk*). Ainsi le premier événement se chargera de faire un *dispatch* de d'autres événements débutant la partie. Les second et dernier événements sont des événements de gestion d'affichage et d'entrées/sorties du jeu.

```
(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

Ces événements doivent donc s'occuper de poursuivre le calcul du jeu leur étant assigné en ré-ordonnacant de nouveaux événements ultérieurement dans le jeu. Par exemple, la figure 6.4 illustre l'évènement principal implantant le comportement désiré pour le vaisseau mère des ennemies.

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

```
(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*.

est que la plupart des évènements se trouvent à être récursifs, et donc ils se ré-ordonnancent par eux même un peu plus tard dans le temps, ce qui donne 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 aie été déjà prévu, mais que ce dernier a explosé suite à la collision avec un laser du joueur. Donc de tels événements sont prompts à donner des erreurs dues à des inconsistance entre l'état du jeu attendu et l'état réel lorsque ce produit un événement.

Aussi, il est important de faire bien attention a doser le calcul effectuer par un événement de manière à ne pas éterniser le temps du processeur utilisé. Ainsi, les évènements sont souvent ré-ordonnancés avec une intervalle de temps de 0 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 pouvoir très 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 reçoit une continuation en paramètre. Elle termine son animation en utilisant une animation CPS s'occupant de faire clignoter le score du joueur, et 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 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.

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

L'avantage principal de l'utilisation de ces structures est qu'elles sont très performantes. Par contre, elles ne permettent pas d'avoir facilement des champs com-

```
(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

muns à toutes les instances d'un type donné. Par exemple, l'expression du fait que toutes les instances du type `player-ship` possèdent doivent être associés à une boîte englobante (*bounding box*) de 13 par 8 pixels et qu'ils sont rendus par la fonte nommée `player`. Ces informations pourraient être ajoutées dans chaque instances, mais ce serait un gaspillage important d'espace mémoire puisque ces données sont

```
(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*

commune à toutes les instances de ce type. Ainsi, pour cette version de *Space Invaders* , un système 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 l'écriture de code qui est fastidieuse et qui ne devrait pas être nécessaire.

La détection de collisions sur ces objets a pu être fait 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 présent dans le jeu en même temps, une détection très rudimentaire de collisions est effectuée. 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*

Un problème majeur relié à l'utilisation de ces structures de données est par contre 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 a 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 a 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 collision dans la première version de *Space Invaders*

Les parties multi-joueurs de *Space Invaders* sont très simple. Chaque joueur joue en alternance jusqu'à se que son vaisseau se fasse détruire. Ainsi, cela implique de pouvoir conserver deux parties du jeu de manière *parallèle*, où une seule des deux avance à la fois. Avec le flot de contrôle sous forme de simulation par événements discrets utilisé, cela implique d'avoir deux simulations en parallèle. Puisque le système de simulation ne permet pas de pouvoir facilement arrêter une simulation et d'avoir une deuxième simulation se produisant en même temps, une autre approche a été utilisée. Un système de coroutines dans lequel les simulations seront exécutés dans des 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 tel qu'il est présenté dans ce chapitre, mais les composantes de bases restent les mêmes. La figure 6.9 résume la procédure utilisée lors de la mort d'un joueur. Après avoir effectuer 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. Ensuite, 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.

Ainsi, l'écriture de cette première version de *Space Invaders* a permise 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

```
(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

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 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'autre. 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

Ainsi, 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 de manière manuelle 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 donc maintenant déclarés comme des classes appartenant à une hiérarchie relativement simple et tirant profit de la possibilité de concevoir des hiérarchies multiples afin de donner 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 membre 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 ses deux classes, tout en ajoutant les deux champs communs à tous les types d'*invader*. Cette classe est sous-classée par les trois types d'ennemis existant (à l'exeption 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 ainsi d'utiliser des opérations spécifiques aux objets de manière complètement transparente, ce qui rend l'écriture de ces parties du jeu beaucoup

plus élégante et solide face aux changements éventuels des structures de donnés ou de l'architecture du jeu.

Ainsi, 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'amélioration 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 beaucoup 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 d'utilisation de 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 coroutine présenté dans le chapitre 5 a été étendu de manière à ce que le flot de contrôle des entités du jeu soit exprimé plutôt sous forme de *coroutine*. 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 beaucoup plus naturelle. En effet, en utilisant des événements discrets, le corps des événements devait faire en sorte que l'événement se poursuive en se ré-

ordonnancant 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 autre coroutine 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 entités est primordiale et implique des protocoles de communication complexes entre les objets. Ces protocoles doivent régir tout le comportent du jeu et, en bout de ligne, le flot de contrôle devient soudainement très difficile à ex-

```
(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

primer. Ainsi, des méta-objets ont dû être introduits afin de pouvoir effectuer la synchronisation entre les objets de base, comme les ennemis. La figure 6.20 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'objet 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.

Par contre, même après avoir optimisé certaines coroutine, les performances du jeu demeuraient très mauvaises et donc, la troisième itération de *Space Invaders* fut abandonné en cours de route, après avoir implanté la majeur partie de la logique du jeu 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, entre autre, de trouver plusieurs problèmes généraux reliés au développement de jeux vidéo comme des problèmes de rendus graphiques, de flot de contrôle, de

résolution de collisions, etc... Ces problèmes 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.

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

Ce jeu consiste à tenter de capturer tout l'or dispersée 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, un 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.

[TODO: *Ajouter un screen du jeu avec des robots*]

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

Afin de pouvoir développé ce jeu, les meilleurs techniques utilisées pour le développement de *Space Invaders* ont été réutilisées et raffinées pour les besoins plus grands de ce jeu. Par contre, le flot du contrôle est régi de manière beaucoup plus conservatrice en utilisant une boucle principale effectuant le travail à faire pour chaque image du jeu. 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.

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.

Cette approche vis-à-vis le flot de contrôle et le fait que le jeu *Lode Runner* contient des entités plus complexes impliquent que le comportement des objets de ce jeu sont dépends de la situation. Encore une fois, ce sujet est traité de manière plus traditionnelle en utilisant des états d'objets et des machines à états afin de régir leurs animations. La figure 6.15 donne le contenu de la classe `human-like` qui sert de bases aux classes `player` et `robot` qui servent à implanter respectivement les instances du joueurs et des robots ennemis. Elle démontre bien la complexité de l'état de ces entités.

La fonction générique `change-state!` utilise l'état des instances de ces classes

```
(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)))))))
(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*

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`.

[TODO: *Si jai le temps, remplacer ca un DSL de machines a etats*]

Une nette amélioration apportée au jeu, en comparaison celui développée 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

```
(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*

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 6.17 illustre l'algorithme de détection de collisions utilisé.

La résolution de collisions ainsi que le rendu graphique sont effectués tous deux 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. Il est intéressant de noter que le rendu graphique est légèrement plus complexe que pour

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

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

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 6.18, le joueur se trouve sur la couche `human-like-layer` qui est prioritaire

```
(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 6.17 – Détection de collisions dans *Lode Runner*

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 6.18 – Implantation des couches de rendu dans *Lode Runner*

[TODO: *Intelligence Artificielle*]

6.2.1 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 s'est avérée à permettre 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, ce qui vient à l'encontre des paradigmes de la programmation fonctionnelle. Par contre, puisque le langage Scheme offre le meilleur

des deux 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.

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.

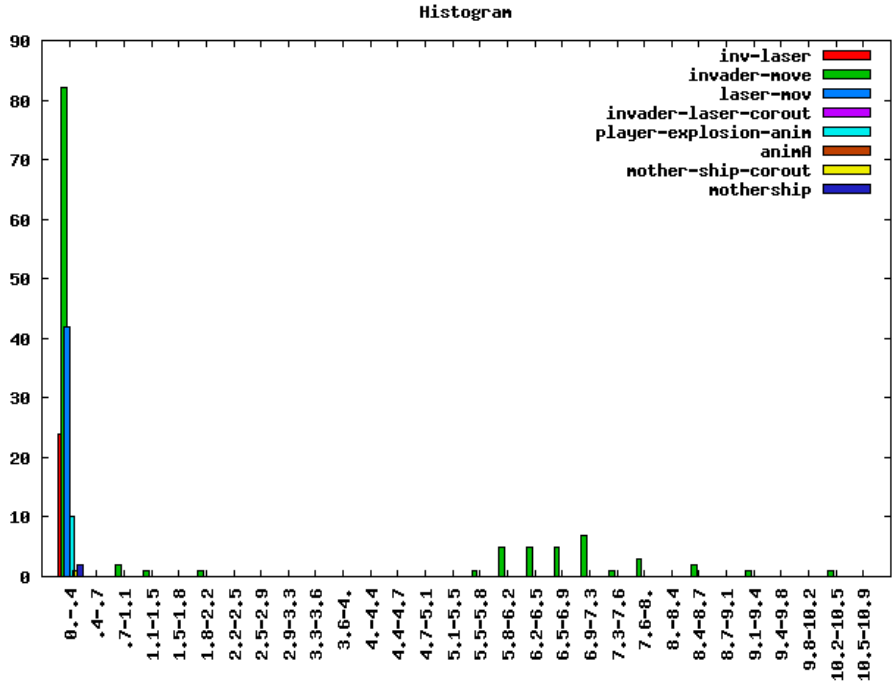
6.3 Conclusion

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 change-

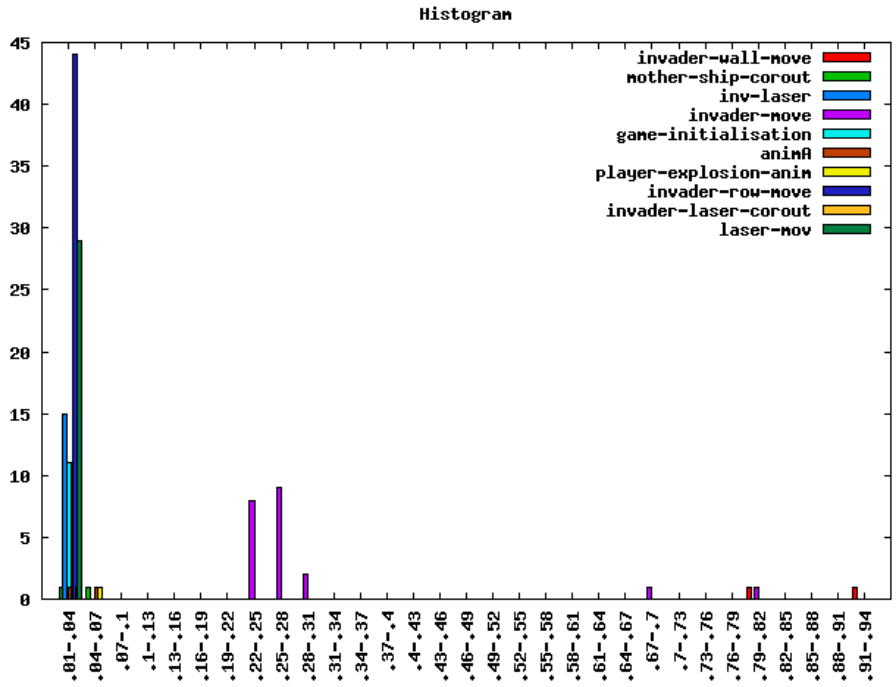
ment de l'architecture du flot de contrôle d'un jeu en passant d'un flot contrôlé par une simulation par événements discrets à une simulation par agents. La conversion s'est faite sans trop d'efforts, grâce aux abstractions utilisées dans le jeu.

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 premiers ordre combinées avec un système d'objets très polyvalent. En utilisant, cette fois-ci, une approche conservatrice sur le flot de contrôle, le résultat fut très clair : le jeu a été fait rapidement et sans aucun problèmes.

Ainsi, il serait maintenant intéressant de voir quel serait l'impact de l'implantation 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.



(a) Avant optimisation de la coroutine `invader-move`



(b) Après optimisation de la coroutine `invader-move`

FIGURE 6.19 – Profilage de coroutines du jeux *Space Invaders*

```
(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 6.20 – 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.

Le langage de programmation Scheme a déjà été utilisé 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.

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. 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 ne peut pas être interprété, même si les *templates* permettent une émulation simple de typage dynamique.

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.

7.1.2 Lua

Un langage Lua [45] 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, entre autre, de permettre aux joueurs de pouvoir ajouté 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 au-

tomatique, l'optimisation d'appels terminaux et un système de coroutines. Il fourni aussi un système d'objet très limité à la Javascript [20], 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 [46] et Javascript [20] 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.

7.2.1 Naughty Dog

La compagnie Naughty Dog [47] 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 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 [48]. 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 parfaite-

ment 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 créer 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, 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 [49] 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 équivalent l'utilisation d'un système de programmation orienté objet, où l'état est conservé dans ces fermetures.

Afin d'éviter d'avoir des problèmes de pauses dues à la gestion de mémoire auto-

matique, plusieurs techniques ont été utilisées afin de cacher des données statiques au ramasse miettes. En effet, l'utilisation de la sérialisation de données disponible dans Gambit-C a été utilisé afin de transformer de grosses structures de données statiques lors 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.

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 que Scheme leurs a fourni un environnement de développement dynamique et l'extensibilité du langage a jouer à l'avantage de ceux-ci.

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 sur l'utilisation d'un langage de programmation fonctionnelle 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.

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 approprié 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.

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, l'utilisation de ramasse miettes 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] Scheme request for implementation.
`http://srfi.schemers.org/`.

[20] 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.

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

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

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

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

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

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

[27] 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.

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

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

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

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

[32] 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.

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

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

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

[35] 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.

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

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

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

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

[40] 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.

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

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

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

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

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

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

[47] Naughty dog. http://www.naughtydog.com/.

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

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

[49] Quantz.

http://www.quantzgame.com/.