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 [1]. La place occupée par l'industrie du jeu vidéo durant cette même année s'estime à 76% du marché de tous les logiciels vendus [2].

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 donc beaucoup d'efforts doivent 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 [3]. 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 hommes par année.

Puisque la création d'un jeu vidéo peu 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 peu engendrer une diminution énorme des coûts de production et

aussi 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, par exemple en C, C++ ou encore C# [4]. Ces langages sont généralement utilisés parce qu'ils sont déjà bien établis et que la main d'oeuvre est facilement accessible.

[TODO: *expliquer haut niveau / bas niveau*]

Les langages de haut niveau sont généralement caractérisés par le fait qu'ils font une bonne abstraction du système utilisé et permettent d'utiliser celles-ci 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 important. 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 les délais occasionnés par les cycles de développement des jeux en permettant aux programmeurs et designers de s'exprimer plus facilement. Le langage de programmation Scheme [5] semble être un bon candidat en tant que langage de haut niveau. En effet, le langage Scheme offre les fonctionnalités de haut niveau

suivantes :

[TODO: *Gardes les bullets ou changer en texte ? ?*]

- Typage dynamique
- Fonctions de premier ordre
- Système de macros évolué
- Accès direct aux continuations du calcul
- Un système de déboggage très efficace

Ces particularités du langage Scheme sont discutées plus en détail dans le chapitre 3.

Le système Gambit-C, l'une des implantations de Scheme les plus performantes [6] sera utilisée 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 ou des *threads* (processus légers).

Il semble donc qu'une utilisation judicieuse de ce système pourrait 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 :

[TODO: *Inserer dans un texte ou garder en quotation ?*]

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. Concurremment, 2 jeux ont été développés pour raffiner nos approches et les évaluer dans un contexte réel.

Le premier jeux 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 2 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 exemple d'utilisation du langage Scheme 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 tout 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 situent 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 aux niveau de la programmation exprimés par les jeux vidéo sont énoncées afin de permettre 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 renferme une mine d'informations importante qui permet d'exposer la manière avec laquelle les jeux vidéo évoluent. Seulement un bref résumé de cette histoire est présenté afin de

donner présenter les grandes ligne de l'évolution des jeux au cours des cinquante dernières années [7] [8].

[TODO: *Mettre des references partout, ou bien ne pas en mettre du tout ?*]

2.1.1 Préhistoire 1948-1970

Les jeux vidéo ont fait leurs apparitions avant même les premiers ordinateurs. En effet, le *Cathode Ray Amusement Device* [9] 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 leurs apparition sur les premiers ordinateurs universitaires, notamment au *MIT* et à l'université de Cambridge. On y retrouve notamment le jeu *Spacewar !* qui est l'un des premier jeu multi-joueurs mettant les joueurs en adversité dans leurs vaisseaux spatiaux pouvant lancer des missiles.

Avec un intérêt stimulé 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 Standford ont porté 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 fondé et démarra l'industrie du jeu vidéo sur arcade avec le jeu *Pong*, un jeu de tennis de table permettant à un joueur de jouer contre une intelligence artificielle ou bien de se mesurer à un autre joueur.

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 simple, ils étaient beaucoup appréciés pour l'amusement qu'ils procuraient aux joueurs qui tentaient toujours de se surpasser.

[TODO: *Marc, quelles genre de machines est-ce que c'etait ? Processeur 8bit ?*]

On retrouve toujours des salles d'arcades de nos jours, mais celles-ci sont devenues beaucoup moins populaire que les consoles de jeu qui se retrouvent dans nos salons.

2.1.3 Premières consoles 1972-1984

La première console de jeu vidéo, le *Magnavox Odyssey* fut son apparition en Amérique du Nord en 1972. Cette console permettant aux joueurs de jouer sur le télévision assis confortablement dans leurs salon. Elle permettait aussi d'insérer des jeux sous forme de cartouches. Un total de 28 jeux ont été disponible pour cette

console, qui malgré l'absence de périphérique audio, a été vendu 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 puissante 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 porter les jeux d'arcades populaires sur leurs consoles.

2.1.4 Ordinateurs personnels 1977-...

Les premiers ordinateurs personnels furent leurs apparition vers la fin des années 1970, dont 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 disquette, en passant bien sur 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 système 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 graphique ou utilisant des périphériques plus variés sur les ordinateurs personnels. Ainsi, les jeux d'ordinateurs existait dans un monde parallèle à celui des consoles, ne se livrant 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. Ainsi, les jeux sur des consoles actuelles rivalisent avec 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 compagnie du jeu vidéo comme *Nintendo* et *Bandai* se sont intéressée à de telles consoles et en 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 à par la suite fusionner ces dernière 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 ont continuées d'évoluer grandement et elles sont actuellement équivalente aux consoles de une ou deux génération précédente. 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 complexe 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 mauvaises gestions ont fait en sorte que l'industrie du jeux vidéo à 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 avait 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 ont certainement favorisé l'adoption de cette nouvelle console. On retrouve entre autre des jeux de tout genre comme le jeu de plate-forme *Super Mario Bros*, le jeux 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, c'est la guerre des consoles qui se livre 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 tentent de séduire un certain public cible.

Au moment de l'écriture de ce mémoire, on retrouve la console *Wii* de Nintendo conçu pour un publique constitué de joueurs occasionnels grâce au contrôleur révolutionnaire 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 sérieux 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 jeu équivalents.

2.2 Industrie du jeu vidéo

Comme 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.

Ces produits possèdent plusieurs caractéristiques de qualité communes auxquelles les consommateurs s'attendent à obtenir en effectuant l'acquisition d'un nouveau titre. 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

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 au joueurs, une étroite collaboration entre les développeurs et les designers doit être établie. Cela implique aussi de faire beaucoup d'essaies 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 ou 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 rigeur de programmation afin d'assurer une stabilité de connections et empêcher les joueurs de tricher. Ces sujets ne sont pas discuter dans ce mémoire.

La diversité des jeux développés a mené à une caractérisation des jeux et une classification de ceux-ci. 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 genres hybrides de ces catégories, mais celles-ci donnent une idée de la diversité du contenu des jeux vidéo produits.

[TODO: *Ajouter du texte liant a la prochaine section ?*]

2.3 Contraintes de programmation

Le développement de jeu vidéo implique une programmation sous des contraints sévères afin que le jeu respecte les normes de l'industrie et respecte les attentes des joueurs.

Une contrainte importante portant sur les jeux vidéo est relié 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. Les jeux doivent donc être des programmes très efficaces de manière à inviter le joueur à entrer le plus possible dans la narration du jeu.

Une autre contrainte importante relié 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.

[TODO: *Autres choses ?*]

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 du projection cinématographique.

[TODO: *references necessaire ?*]

Ceci étant dit, un taux de rafraîchissement de 60 trames par secondes n'est pas nécessaire pour tout 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.

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 optimisée 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 délais de réponse face aux entrées du joueur. Ce délais 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 joueurs se doit aussi d'être

très efficace afin de ne pas encombré 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 ce 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 de jeu, surtout de jeux modernes, car ce sont de 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 parallèles 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 donc qui permettent d'être réutilisée par la suite. Une réutilisation de composantes 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 placé pour bien modulariser les composantes d'un jeu seront retables pour le développement des prochains jeu.

Bien sûr, toute abstraction possède sont coût, notamment en coût de perfor-

mances. Ainsi, il faut bien pouvoir estimer les endroits où l'utilisation de modules externes n'apportera pas de trop grands impacts 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 à bien modulariser les composantes d'un jeu.

2.3.3 Malléabilité

Puisque le développement d'un jeu implique généralement de faire beaucoup de prototypage, le code d'un jeu se doit 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

[TODO: *Langages de haut niveau permettrait d'aider au développement de jeu en apportant une grande expressivité et puissance d'abstraction.*] [TODO: *petites*

réduction du temps de développement == grandes économies budgétaires] [TODO: *Bonnes abstraction == réutilisation*]

CHAPITRE 3

LE LANGAGE SCHEME

[TODO: *Choix du langage influence beaucoup la structure du code, le manière de développer le logiciel, etc...*]

[TODO: *Scheme est un langage qui semble être peu utilisé dans des produits commerciaux.*]

[TODO: *d'autres idées ?*]

[TODO: *Référence temporaire pour pas faire planter le makefile [10]*]

3.1 Héritage de LISP

3.1.1 Histoire de LISP

[TODO: *McCarthy, GC, List Processing, AI...*]

[TODO: *Common LISP = Grosse Bébitte*]

3.1.2 Avènement de Scheme

[TODO: *Épuration de LISP à l'essence du langage*]

[TODO: *évolution du langage via RNRS, SRFIs*]

[TODO: *simplicité du langage, beaucoup d'extension disponibles*]

3.1.3 Langages inspirés de Scheme

[TODO: *Javascript, Ruby : lang de scripting. Concept de fermetures*]

[TODO: *Java ? ?*]

3.2 Programmation fonctionnelle

3.2.1 Distinction

[TODO: *Fonctions sont des données de premier ordre*] [TODO: *Fonctions d'ordres supérieures (avec exemples)*]

3.2.2 Programmation fonctionnelle pure

[TODO: *Pas de mutation. Comme des blocs lego*]

[TODO: *Haskell, ML*]

[TODO: *Exemple en Scheme*]

3.2.3 Effets de bords dans Scheme

[TODO: *Entrees sorties (variables à portée dynamique)*] [TODO: *Mutations de variables*]

3.3 Macros

3.3.1 Introduction

[TODO: *Discussion sur les macros de C et leurs limitations : permet de faire des abstractions dans le code résultant une modification du code source avant la compilation, mais uniquement limité à du remplacement de code.*]

[TODO: *Explication des macros scheme : Code source == données scheme, dispose d'une pré-évaluation permettant de prendre du code source en entré et de généré du code source (toujours sous forme de listes (données Scheme)). Revient a faire une manipulation d'ASA.*]

[TODO: *Dispose de toute la puissance de calcul durant l'expansion :)*]

3.3.2 Exemples simples

[TODO: *macro inc*]

[TODO: *macro while*]

3.3.3 Problème de l'hygiène des macros

[TODO: *Capture de nom intentionnelle, ou non intentionnelle.*]

[TODO: *Différentes formes spéciales (define-macro, define-syntax)*]

3.4 Continuations

3.4.1 Introductions du sujet et explications

[TODO: *En C, continuation equiv au code situé à l'adresse de retour de la fonction exécutée.*]

[TODO: *Explication de l'ordre dévaluation en Scheme.*]

[TODO: *Continuation d'un programme simple en Scheme. (+ 1 (- [f x y] 2)) : continuation de l'appel à f est la soustraction du résultat par deux, puis l'on ajoute à 1 se résultat. Eq à (lambda (res-f) (+ 1 (- res-f 2))).*]

3.4.2 Réification de continuations

[TODO: *Explication de call-with-current-continuation permettant de faire*] [TODO: *surfacer une continuation.*]

3.4.3 Exemples d'utilisations

3.4.3.1 Échappement au flux de contrôle

[TODO: *Utilisation comme un return*]

3.4.3.2 Système de coroutine

[TODO: *Exemple du cours de Marc*]

3.4.4 Forme d'écriture de code en CPS

[TODO: *Expliquer la différence entre un appel terminal et un appel non-terminal. Expliquer comment les appels terminaux peuvent être optimisés (requis en Scheme).*]

[TODO: *Écrire le code de manière à rendre explicite le passage et les appels de continuations. Toujours des appels terminaux. Transformation souvent utilisée par les compilateurs pour implanter l'optimisation d'appels terminaux.*]

3.4.5 Exemple d'implantation

[TODO: *Exemple du cours de Marc*]

3.5 Dynamisme du langage

[TODO: *Typage dynamique vs typage statique*]

[TODO: *Exemple de code utilisant le typage dynamique.*]

[TODO: *Evaluation dynamique de code avec load ou eval. idées ?*]

[TODO: *Langages comme C++ favorisent bcp de focuser sur la conceptions des classes. Implique une structure rigide qui se doit d'être bien conçu depuis le départ.*]

[TODO: *Langages plus dynamiques permettent des cycles de prototypage rapide en apportant beaucoup de flexibilité aux développeurs.*]

3.6 Gestion mémoire automatique

3.6.1 Motivation

[TODO: *Gestion mémoire manuelle ou semi-automatique causent énormément de problèmes de fuites de mémoire (ou de pointeurs fous).*]

[TODO: *En connaissant les racines, on peut automatiser la récupérations de mémoire, au coût d'un certain temps de calcul.*]

[TODO: *Date des premières version de LISP.*]

3.6.2 Survol des techniques

[TODO: *Mark and sweep*]

[TODO: *Stop and copy*]

[TODO: *GC générationnels*]

[TODO: *GC temps réels ou incrémentaux*]

3.7 Débuggage

[TODO: *Possibilité de pouvoir exécuter du code arbitraire lors d'un crash*]

[TODO: *Possibilité de pouvoir inspecter chacune des frames de la pile de continuactions afin de pouvoir obtenir de l'info sur l'environnement de celui-ci, l'endroit dans le code source où la continuation se trouve et même de pouvoir exécuter du code dans l'environnement d'une continuation de la pile choisie.*]

[TODO: *Possibilité de faire le déboggage à distance (ex : iPhone)*]

CHAPITRE 4

PROGRAMMATION ORIENTÉE OBJET

[TODO: *Parler des define-type de Gambit-C et du SRFI-9*]

[TODO: *Approche traditionnelle de la prog OO : Surdéfinition de méthodes de classe, héritage simple, polymorphisme, etc...*]

[TODO: *Parler de CLOS : intégré à Common LISP, fonctions génériques, héritage multiple*]

[TODO: *Besoins pour jeux vidéo*]

4.1 Description du langage

[TODO: *Langage orienté objet qui a pour but d'être efficace tout en apportant l'expressivité offerte par la programmation orientée object à la CLOS.*]

Résumé des fonctionnalités :

- Accès aux membres rapide
- Héritage multiple
- Polymorphisme
- Fonctions génériques à « dispatch »multiple

4.1.1 Définition de classes

[TODO: *Compatibilité avec les define-type*]

[TODO: *Définitions simples (instance slots et class slots)*]

[TODO: *Héritages des membres*]

[TODO: *Utilisation de hook sur les slots*]

[TODO: *Constructeurs*]

4.1.2 Définition de fonctions génériques

[TODO: *Dispatch simple*]

[TODO: *Dispatch multiple (avec les problèmes reliés à la résolution de la méthode à choisir)*]

[TODO: *call-next-method*]

[TODO: *Type '*'*]

4.1.3 Fonctions et formes spéciales utilitaires

[TODO: *Vérifications manuelles de typages et introspections (instance-object ?, instance-of ?, find-class ?, get-class-id, is-subclass ?, get-supers)*]

4.2 Implantation

[TODO: *Apercu global : Utilisation de macros scheme pour effectuer la génération de code nécessaire au fonctionnement du système.*]

[TODO: *Séparation entre le travail fait durant l'expantion macro et l'exécution*]

[TODO: *Passage d'informations entre le moment de l'expansion et l'exécution (informations sur les classes, les fonctions génériques, etc...)*]

4.2.1 Implantation de define-class

[TODO: *Structures de données (descripteurs de classes, format des instances, etc..)*]

[TODO: *Polymorphisme : chaque index dans les descripteurs de classes sont orthogonaux (implique que les descripteurs grossissent linéairement en fonction du nombre de classes) et passage aux classes enfants des indexes utilisés par les parents.*]

[TODO: *Constructeurs et describe comme fonctions génériques*]

4.2.2 Implantation de define-generic

[TODO: *Registre des méthode : Conservations d'informations sur les fonctions génériques et leurs instances*]

[TODO: *Implantation du polymorphisme des fonctions génériques*]

[TODO: *Implantation du call-next-method faite avec l'utilisation de variables à portée dynamique*]

[TODO: *Coût de l'utilisation des fonctions génériques*]

4.2.3 Implantation de define-method

[TODO: *Stockages des fermetures durant l'expansion macro et l'exécution*]

4.3 Conclusion

[TODO: *Ouverture sur le fait qu'un meta-protocole serait très intéressant à ajouter, mais à quel prix ?*]

CHAPITRE 5

SYSTÈME DE COROUTINES

[TODO: *Système de threads offerts ne sont pas toujours satisfaisant. Mentalité Scheme : au lieu de contraindre ce qu'on veut faire en fonction de ce qui nous est offert, étendre le langage pour nous permettre de faire ce que l'on veut et surtout de la manière désirée.*]

[TODO: *Implantation de son propre système : contrôle fin du comportement de threads*]

[TODO: *Parler de Termite : Calcul distribué sur plusieurs noeuds. Synchronisation par passage de message. Communication via TCP/IP.*]

[TODO: *Motivation de l'utilisation de coroutines (contrôle exact sur le flot de contrôle == système toujours dans un état consistant).*]

5.1 Description du langage

[TODO: *Idée sur nos coroutines*]

[TODO: *Systèmes récursifs*]

[TODO: *Timers : abstraction du temps écoulé permettant l'accélération ou le ralentissement de l'exécution d'une simulation.*]

5.1.1 Création de coroutines

[TODO: *Création d'une coroutine détachée du système : (new corout ¡id¿ ¡thunk¿)*]

[TODO: *intégrée au système via le démarrage (boot) ou via une autre coroutine (spawn-brother)*]

5.1.2 Manipulation du flot de contrôle

5.1.2.1 yield

[TODO: *Transfert à la prochaine coroutine*]

5.1.2.2 super-yield

[TODO: *Transfert au prochain système de coroutine (frère du système courant).*]

5.1.2.3 terminate-corout, kill-all !, super-kill-all !

[TODO: *Terminaison de coroutines.*]

5.1.2.4 sleep-for

[TODO: *Sommeil pour un temps prédéterminé.*]

5.1.2.5 continue-with

[TODO: *Continuation de la coroutine.*]

5.1.2.6 spawn-brother, spawn-brother-thunk

[TODO: *Démarrage de nouvelles coroutines.*]

5.1.2.7 Composition of coroutines

[TODO: *Compositions ou séquençage de coroutine*]

5.1.3 Système de communication inter coroutines

5.1.3.1 !

[TODO: *Envoi de message à une coroutine*]

5.1.3.2 ?

[TODO: *Réception d'un message bloquante (avec possibilité de timeout)*]

5.1.3.3 ? ?

[TODO: *Réception sélective de message bloquante (avec possibilité de timeout)*]

5.1.3.4 recv, dynamic msg handlers

[TODO: *Forme spéciale permettant la réception sélective de messages via un « pattern matching »qui permet une notation concise et l'utilisation aisée du contenu des messages reçus.*]

5.1.3.5 Messaging lists

[TODO: *Système permettant de regrouper des coroutines et de leurs diffuser des messages.*]

5.1.4 Démarrage du système

5.1.4.1 simple-boot

[TODO: *Démarrage rapide du système.*]

5.1.4.2 boot

[TODO: *Démarrage permettant de spécifier un timer spécifique à l'utilisation et une fonction personnalisée effectuant la gestion des valeurs de retour des coroutines.*]

[TODO: *Systèmes cascadés ?*]

5.2 Implantation

5.2.1 Implantation des coroutines

[TODO: *Structure de données*]

[TODO: *États d'une coroutines*]

5.2.2 Scheduler

[TODO: *Abstraction du temps via timer*]

[TODO: *États du scheduler*]

[TODO: *Algorithme de scheduling*]

5.2.3 Système de messagerie

[TODO: *Structures de données*]

[TODO: *Envoi de messages*]

[TODO: *Réception de messages*]

[TODO: *Macro recv*]

5.3 Conclusion

[TODO: *Ouverture sur le profilage des coroutine*]

CHAPITRE 6

ÉVALUATION ET EXPÉRIENCES

Développement de jeu fait ayant comme but de trouver les problèmes rencontrés durant la création de jeux vidéo et de proposer des méthodes pour résoudre ces problèmes.

Aussi, on cherche à identifier les avantages que nous a fourni Scheme et à identifier les incovénients que pose l'utilisant du langage Scheme pour le développement de ces jeux.

Débute par un jeu simple afin de trouver les problèmes de base et trouve des solutions à ces problèmes.

Ensuite, un deuxième jeu a été écrit afin de consolider les solutions trouvées précédemment et de potentiellement trouver d'autres problèmes reliés aux nouvelles complexité présentent dans ce deuxième jeu.

6.1 Développement de « Space Invaders »

6.1.1 Objectifs

[TODO: *Expérimentation avec un jeu très simple*]

[TODO: *Trouver les problèmes fondamentaux pour le développement de jeux*]

[TODO: *Tenter de les résoudre*]

6.1.2 Version initiale

[TODO: *Premier jet dans le but de trouver des problèmes potitiels*]

- Comment faire des animations ? =¿ CPS
- Comment concevoir une partie a 2 joueurs ? =¿ coroutines
- Difficulté à décrire la résolution de collision de manière efficace
- Est-il possible d'écrire le comportement d'une entité de manière indépendante, i.e. que le code soit centralisé dans une même fonction ?

6.1.3 Version orientée objet

[TODO: *Motivation : Utilisation de fonctions génériques*]

[TODO: *Hierarchie de classe*]

[TODO: *Code Highlight : Résolution de collisions*]

6.1.4 Version avec système de co-routine

[TODO: *Motivation : Intégrer les coroutines a chaque objet de manière à ce que chaque instance soit une entité à part entière qui doit régir son propre comportement.*]

[TODO: *Difficultés : synchronisation des entités*]

[TODO: *Code Highlight : synchronisation des invaders*]

6.1.5 Conclusion

[TODO: *Trouvé plusieurs problèmes et pu résoudres ces derniers*]

[TODO: *Utilisation d'un système object a grandement contribué à améliorer le code du jeu.*]

L'intégration du système de coroutines aux objets du jeu a causé plus de problème qu'elle en a résolu. L'utilisation des coroutines serait mieux d'être limité à l'implantation du jeu multijoueur.

[TODO: *ouverture : Essayer ces techniques dans un jeu plus complexe pour voir si elles sont toujours valides*]

6.2 Développement de « Lode Runner »

6.2.1 Objectifs

[TODO: *Jeux plus complexe : plus d'interaction du joueur, intelligence artivicielle, niveaux, schema d'animations plus complexe, etc...*]

[TODO: *Utiliser ce qui semblait de meilleur dans space-invaders de manière a non seulement confirmer la pertinance de ces methodes, mais aussi a potentiellement en developper de nouvelles dû aux nouvelles contraintes de ce jeu.*]

6.2.2 Synchronisation

[TODO: *Utilisation du concept de frame pour faire la synchro. (manière traditionnelle) Réduit de beaucoup la complexité.*]

[TODO: *Danger si le framerate varie, la vitesse du jeu varie.*]

6.2.3 Machines à états

[TODO: *Utiliation des fonctions génériques*]

[TODO: *Utilisez un LSD pour ca ? ? ? Des idées ?*]

6.2.4 Intelligence Artificielle

6.2.4.1 À venir...

6.2.5 Conclusion

6.2.5.1 À venir...

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 complètement différentes.

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

7.1.1 Lua

Langage utilisé très fréquemment pour effectuer le « scripting »dans les jeu vidéo.

Differences entre lua et Scheme

- Lua est de petite taille en mem
- ..

7.1.2 C++

Langage principal de développement de jeu vidéo en industrie.

Differences entre Scheme et C++

- Gestion memoire manuelle vs GC
- méthode surdéfinies vs fonctions génériques
- ...

7.2 Jeux en Lisp

7.2.1 QuantZ

Jeu de type « casual »de très bonne qualité écrit presque entièrement en Scheme.

À voir avec Robert

FRP ?

Techniques anti-gc

Delegation de fermetures

7.2.2 Naughty Dogz

Compagnie très connue associée à Sony qui utilisent Scheme pour produire leurs jeux vidéo.

7.2.2.1 GOAL

Compilateur Scheme utilisé pour produire les jeux sur PlayStation 2

- http ://en.wikipedia.org/wiki/Game_Oriented_Assembly_Lisp
- http ://grammerjack.spaces.live.com/blog/cns !F2629C772A178A7C !135.entry

7.2.2.2 Drake's uncharted Fortune

CHAPITRE 8

CONCLUSION

L'expérience d'écriture de ces jeux aura permis de faire le point sur les avantages et les inconvénients de l'utilisation d'un langage tel que Scheme pour le développement de jeu vidéo.

+ puissance d'expression / d'abstraction

+ langage dynamique (développement en-direct, malléabilités)

+ création de langages spécifiques au domaine

- Garbage Collection et sur-allocation

- Profilage plus difficile avec des LSD (pour Gambit-C et statprof)

- Balance entre abstraction et efficacité

BIBLIOGRAPHIE

[1] 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.

[2] 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.

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

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

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

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

[7] 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.

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

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

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