

Université de Montréal

**Développement 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

© David St-Hilaire, 2009.

Université de Montréal  
Faculté des études supérieures

Ce mémoire intitulé:

**Développement 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

<b>RÉSUMÉ</b> . . . . .	<b>iii</b>
<b>ABSTRACT</b> . . . . .	<b>iv</b>
<b>TABLE DES MATIÈRES</b> . . . . .	<b>v</b>
<b>LISTE DES FIGURES</b> . . . . .	<b>x</b>
<b>REMERCIEMENTS</b> . . . . .	<b>xi</b>
<b>CHAPITRE 1 : INTRODUCTION</b> . . . . .	<b>1</b>
1.1 Problématique . . . . .	3
1.2 Méthodologie . . . . .	3
1.3 Aperçu du mémoire . . . . .	4
<b>CHAPITRE 2 : DÉVELOPPEMENT DE JEUX VIDÉO</b> . . . . .	<b>6</b>
2.1 Historique . . . . .	6
2.1.1 préhistoire 1948-1970 . . . . .	6
2.1.2 Système arcade 1970-1985 . . . . .	6
2.1.3 Premières consoles 1972-1984 . . . . .	7
2.1.4 Ordinateurs personnels 1977-... . . . .	7
2.1.5 Consoles portables 1980-... . . . .	7

2.1.6	Consoles intermédiaires 1984-2006 . . . . .	7
2.1.7	Consoles modernes 2005-... . . . .	7
2.2	Contraintes de programmation . . . . .	8
2.2.1	fluidité . . . . .	8
2.2.2	Modularité . . . . .	8
<b>CHAPITRE 3 : LE LANGAGE SCHEME . . . . .</b>		<b>9</b>
3.1	Héritage de LISP . . . . .	9
3.1.1	Histoire de LISP . . . . .	9
3.1.2	Avènement de Scheme . . . . .	9
3.1.3	Langages inspirés de Scheme . . . . .	10
3.2	Programmation fonctionnelle . . . . .	10
3.2.1	Distinction . . . . .	10
3.2.2	Programmation fonctionnelle pure . . . . .	10
3.2.3	Effets de bords dans Scheme . . . . .	10
3.3	Macros . . . . .	10
3.3.1	Introduction . . . . .	10
3.3.2	Exemples simples . . . . .	11
3.3.3	Problème de l'hygiène des macros . . . . .	11
3.4	Continuations . . . . .	11
3.4.1	Introductions du sujet et explications . . . . .	11
3.4.2	Réification de continuations . . . . .	12

3.4.3	Exemples d'utilisations . . . . .	12
3.4.4	Forme d'écriture de code en CPS . . . . .	12
3.4.5	Exemple d'implantation . . . . .	13
3.5	Dynamisme du langage . . . . .	13
3.6	Gestion mémoire automatique . . . . .	13
3.6.1	Motivation . . . . .	13
3.6.2	Survol des techniques . . . . .	14
3.7	Débuggage . . . . .	14
<b>CHAPITRE 4 : PROGRAMMATION ORIENTÉE OBJET . . . . .</b>		<b>15</b>
4.1	Description du langage . . . . .	15
4.1.1	Définition de classes . . . . .	16
4.1.2	Définition de fonctions génériques . . . . .	16
4.1.3	Fonctions et formes spéciales utilitaires . . . . .	16
4.2	Implantation . . . . .	16
4.2.1	Implantation de define-class . . . . .	17
4.2.2	Implantation de define-generic . . . . .	17
4.2.3	Implantation de define-method . . . . .	17
4.3	Conclusion . . . . .	18
<b>CHAPITRE 5 : SYSTÈME DE COROUTINES . . . . .</b>		<b>19</b>
5.1	Description du langage . . . . .	19

5.1.1	Création de coroutines . . . . .	19
5.1.2	Manipulation du flot de contrôle . . . . .	20
5.1.3	Système de communication inter coroutines . . . . .	21
5.1.4	Démarrage du système . . . . .	22
5.2	Implantation . . . . .	22
5.2.1	Implantation des coroutines . . . . .	22
5.2.2	Scheduler . . . . .	22
5.2.3	Système de messagerie . . . . .	22
5.3	Conclusion . . . . .	23
<b>CHAPITRE 6 : ÉVALUATION ET EXPÉRIENCES . . . . .</b>		<b>24</b>
6.1	Développement de « Space Invaders » . . . . .	24
6.1.1	Objectifs . . . . .	24
6.1.2	Version initiale . . . . .	25
6.1.3	Version orientée objet . . . . .	25
6.1.4	Version avec système de co-routine . . . . .	25
6.1.5	Conclusion . . . . .	26
6.2	Développement de « Lode Runner » . . . . .	26
6.2.1	Objectifs . . . . .	26
6.2.2	Synchronisation . . . . .	26
6.2.3	Machines à états . . . . .	27
6.2.4	Intelligence Artificielle . . . . .	27



6.2.5	Conclusion . . . . .	27
<b>CHAPITRE 7 : TRAVAUX RELIÉS . . . . .</b>		<b>28</b>
7.1	Comparaison de langages . . . . .	28
7.1.1	Lua . . . . .	28
7.1.2	C++ . . . . .	29
7.2	Jeux en Lisp . . . . .	29
7.2.1	QuantZ . . . . .	29
7.2.2	Naughty Dogz . . . . .	29
<b>CHAPITRE 8 : CONCLUSION . . . . .</b>		<b>31</b>
<b>BIBLIOGRAPHIE . . . . .</b>		<b>32</b>

## **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 ouvre les portes aux 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 appliqués à 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\$/année, cela reviendrait à un travail d'environ 300 à 500 hommes/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 dans des langages de relativement bas niveau, comme par exemple en C, C++ ou encore C# [4]. Ces langages sont généralement utilisés parce que la main d'oeuvre est facilement accessible et sont déjà des langages bien établis.

Les langages de haut niveau sont généralement caractérisés par le fait qu'ils font une bonne abstraction du système utilisé et permettent d'utiliser ces abstractions de manière naturelle et donc facilitent 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 et donc, actuellement la performance n'est plus aussi importante de nos jours. 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 :

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

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

[TODO: *introduire Gambit-C*]

Il semble donc qu'une utilisation judicieuse de ce langage de programmation 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 :

Quelles sont les forces et les faiblesses du langage 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 ni-

veau du développement de jeux vidéo. Concurrément, 2 jeux ont été développés pour raffiner nos approches et les évaluer dans un contexte réel.

Le premier jeux a servi de plateforme 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, comme 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 se 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 de 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**

Importance du marche des JV

Diversite des jeux (genres, plates-formes)

Caractéristiques de jeux modernes

Les jeux vidéo font parti d'un domaine de l'informatique en pleine effervescence grâce à une demande constante de nouveaux produits. Ces produits possèdent plusieurs caractéristiques de qualité auxquelles les consommateurs s'attendent à obtenir en effectuant l'acquisition d'un nouveau titre. Ces attentes du consommateur peuvent se traduire par les besoins suivant :

mettre ici une liste des attentes pour un jeu moderne

#### **2.1 Historique**

##### **2.1.1 préhistoire 1948-1970**

CRT games, Mainframe games

##### **2.1.2 Système arcade 1970-1985**

Pong, Space Invaders, Pac-Man

### **2.1.3 Premières consoles 1972-1984**

Magnavox Odyssey, Atari XX00 et ColecoVision

### **2.1.4 Ordinateurs personnels 1977-...**

Évolution parallèle constante

Toujours été jusqu'à la venue des consoles modernes la plateforme offrant les meilleures performances.

### **2.1.5 Consoles portables 1980-...**

GameBoy etc...

### **2.1.6 Consoles intermédiaires 1984-2006**

NES, SNES, N64, GameCube

### **2.1.7 Consoles modernes 2005-...**

Interfaces plus 'casual' = Wii

Consoles très performantes

## 2.2 Contraintes de programmation

### 2.2.1 fluidité

#### 2.2.1.1 Taux de rafraîchissement

#### 2.2.1.2 Réponse quasi temps réelle

### 2.2.2 Modularité

#### 2.2.2.1 Complexité des jeux ... gros logiciels

#### 2.2.2.2 Développement itératif ... plusieurs modifications du système

## CHAPITRE 3

### LE LANGAGE SCHEME

Choix du langage influence beaucoup la structure du code, le manière de développer le logiciel, etc...

Scheme est un langage qui semble être peu utilisé dans des produits commerciaux.

d'autres idées ?

Référence temporaire pour pas faire planter le makefile [6]

### 3.1 Héritage de LISP

#### 3.1.1 Histoire de LISP

McCarthy, GC, List Processing, AI...

Common LISP = Grosse Bébitte

#### 3.1.2 Avènement de Scheme

Épuration de LISP à l'essence du langage

évolution du langage via RNRS, SRFIs

simplicité du langage, beaucoup d'extension disponibles

### 3.1.3 Langages inspirés de Scheme

Javascript, Ruby : lang de scripting. Concept de fermetures

Java ??

## 3.2 Programmation fonctionnelle

### 3.2.1 Distinction

Fonctions sont des données de premier ordre

Fonctions d'ordres supérieures (avec exemples)

### 3.2.2 Programmation fonctionnelle pure

Pas de mutation. Comme des blocs lego

Haskell, ML

Exemple en Scheme

### 3.2.3 Effets de bords dans Scheme

Entrees sorties (variables à portée dynamique) Mutations de variables

## 3.3 Macros

### 3.3.1 Introduction

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.

Explication des macros scheme : Code source == données scheme, dispose d'une pré-évaluation permettant de prendre du code source en entrée et de générer du code source (toujours sous forme de listes (données Scheme)). Revient à faire une manipulation d'ASA.

Dispose de toute la puissance de calcul durant l'expansion :)

### 3.3.2 Exemples simples

macro inc

macro while

### 3.3.3 Problème de l'hygiène des macros

Capture de nom intentionnelle, ou non intentionnelle.

Différentes formes spéciales (define-macro, define-syntax)

## 3.4 Continuations

### 3.4.1 Introductions du sujet et explications

En C, continuation equiv au code situé à l'adresse de retour de la fonction exécutée.

Explication de l'ordre d'évaluation en Scheme.

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  $(\text{lambda } (\text{res-f}) (+ 1 (- \text{res-f } 2)))$ .

### 3.4.2 Réification de continuations

Explication de call-with-current-continuation permettant de faire surfacer une continuation.

### 3.4.3 Exemples d'utilisations

#### 3.4.3.1 Échappement au flux de contrôle

Utilisation comme un return

#### 3.4.3.2 Système de coroutine

Exemple du cours de Marc

### 3.4.4 Forme d'écriture de code en CPS

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

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

Exemple du cours de Marc

blabla

## 3.5 Dynamisme du langage

Typage dynamique vs typage statique

Exemple de code utilisant le typage dynamique.

Evaluation dynamique de code avec load ou eval. idées ?

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.

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

Gestion mémoire manuelle ou semi-automatique causent énormément de problèmes de fuites de mémoire (ou de pointeurs fous).

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

Date des premières version de LISP.



### 3.6.2 Survol des techniques

Mark and sweep

Stop and copy

GC générationnels

GC temps réels ou incrémentaux

### 3.7 Débuggage

Possibilité de pouvoir exécuter du code arbitraire lors d'un crash

Possibilité de pouvoir inspecter chacune des frames de la pile de continuations 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.

Possibilité de faire le débogage à distance (ex : iPhone)

## CHAPITRE 4

### PROGRAMMATION ORIENTÉE OBJET

Parler des define-type de Gambit-C et du SRFI-9

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

Parler de CLOS : intégré à Common LISP, fonctions génériques, héritage multiple

Besoins pour jeux vidéo

#### 4.1 Description du langage

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

Résumé des fonctionnalités

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

#### 4.1.1 Définition de classes

Compatibilité avec les define-type

Définitions simples (instance slots et class slots)

Héritages des membres

Utilisation de hook sur les slots

Constructeurs

#### 4.1.2 Définition de fonctions génériques

Dispatch simple

Dispatch multiple (avec les problèmes liés à la résolution de la méthode à choisir)

call-next-method

Type '\*\*'

#### 4.1.3 Fonctions et formes spéciales utilitaires

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

### 4.2 Implantation

Aperçu global : Utilisation de macros scheme pour effectuer la génération de code nécessaire au fonctionnement du système.

Séparation entre le travail fait durant l'expansion macro et l'exécution

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**

Structures de données (descripteurs de classes, format des instances, etc..)

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.

Constructeurs et describe comme fonctions génériques

#### **4.2.2 Implantation de define-generic**

Registre des méthode : Conservations d'informations sur les fonctions génériques et leurs instances

Implantation du polymorphisme des fonctions génériques

Implantation du call-next-method faite avec l'utilisation de variables à portée dynamique

Coût de l'utilisation des fonctions génériques

#### **4.2.3 Implantation de define-method**

Stockages des fermetures durant l'expansion macro et l'exécution

### 4.3 Conclusion

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

## CHAPITRE 5

### SYSTÈME DE COROUTINES

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.

Implantation de son propre système : contrôle fin du comportement de threads

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

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

#### 5.1 Description du langage

Idée sur nos coroutines

Systèmes récursifs

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

##### 5.1.1 Création de coroutines

Création d'une coroutine détachée du système : `(new corout jidi jthunki)`

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**

Transfert à la prochaine coroutine

### **5.1.2.2 super-yield**

Transfert au prochain système de coroutine (frère du système courant).

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

Terminaison de coroutines.

### **5.1.2.4 sleep-for**

Sommeil pour un temps prédéterminé.

### **5.1.2.5 continue-with**

Continuation de la coroutine.

### **5.1.2.6 spawn-brother, spawn-brother-thunk**

Démarrage de nouvelles coroutines.

### 5.1.2.7 Composition of coroutines

Compositions ou séquençage de coroutine

## 5.1.3 Système de communication inter coroutines

### 5.1.3.1 !

Envoi de message à une coroutine

### 5.1.3.2 ?

Réception d'un message bloquante (avec possibilité de timeout)

### 5.1.3.3 ??

Réception sélective de message bloquante (avec possibilité de timeout)

### 5.1.3.4 recv, dynamic msg handlers

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

Système permettant de regrouper des coroutines et de leurs diffuser des messages.



### **5.1.4 Démarrage du système**

#### **5.1.4.1 simple-boot**

Démarrage rapide du système.

#### **5.1.4.2 boot**

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

Systèmes cascades ?

## **5.2 Implantation**

### **5.2.1 Implantation des coroutines**

Structure de données

États d'une coroutines

### **5.2.2 Scheduler**

Abstraction du temps via timer

États du scheduler

Algorithme de scheduling

### **5.2.3 Système de messagerie**

Structures de données

Envoi de messages

Réception de messages

Macro recv

### **5.3 Conclusion**

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 inconvénients que pose l'utilisant du langage Scheme pour le développement de ces jeux.

Débuter par un jeu simple afin de trouver les problèmes de base et trouver 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 liés aux nouvelles complexités présentes dans ce deuxième jeu.

#### 6.1 Développement de « Space Invaders »

##### 6.1.1 Objectifs

Expérimentation avec un jeu très simple

Trouver les problèmes fondamentaux pour le développement de jeux

Tenter de les résoudre

### 6.1.2 Version initiale

Premier jet dans le but de trouver des problèmes potentiels

- Comment faire des animations ? =  $\dot{}$  CPS
- Comment concevoir une partie a 2 joueurs ? =  $\dot{}$  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

Motivation : Utilisation de fonctions génériques

Hierarchie de classe

Code Highlight : Résolution de collisions

### 6.1.4 Version avec système de co-routine

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.

Difficultés : synchronisation des entités

Code Highlight : synchronisation des invaders

### 6.1.5 Conclusion

Trouvé plusieurs problèmes et pu résoudre ces derniers

Utilisation d'un système objet 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.

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

Jeux plus complexe : plus d'interaction du joueur, intelligence artificielle, niveaux, schéma d'animations plus complexe, etc...

Utiliser ce qui semblait de meilleur dans space-invaders de manière à non seulement confirmer la pertinence de ces méthodes, mais aussi à potentiellement en développer de nouvelles dû aux nouvelles contraintes de ce jeu.

### 6.2.2 Synchronisation

Utilisation du concept de frame pour faire la synchro. (manière traditionnelle)  
Réduit de beaucoup la complexité.

Danger si le framerate varie, la vitesse du jeu varie.

### **6.2.3 Machines à états**

Utilisation des fonctions génériques

Utilisez un LSD pour ça ??? 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 jeux 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://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](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](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<sup>5</sup> report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9) :26–76, 1998.
- [6] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass., 1996.