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

jeu vidéo représentent une importante industrie ($$)

prog de jeu complexe : bcp années/hommes de travail

Comment faciliter le dév ?

Prog haut niveau vs bas niveau

Introduire Scheme

1.1 Problématique

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

> Est-ce possible ou envisageable de concevoir et développer des jeux vidéo en Scheme ? Quels en sont les avantages et les inconvénients ?

1.2 Méthodologie

- Dev 1er jeu simple pour determiner les besoins pour Scheme
- Augmenter Scheme pour repondre a ces besoins
- Ecrire un nouveau jeu utilisant les techniques developpees pour le 1er jeu

Afin de pouvoir répondre à ces questions, 2 jeux vidéo ont été développés en utilisant le langage de programmation Scheme. Le premier jeux a servi de platforme

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 premier jeu, un autre jeu plus complexe que le premier a été écrit afin de consolider les méthodologies précédemment utilisées.

1.3 Aperçu

Ce mémoire peut être divisé en quatre parties distinctes. La première partie est une introduction sur les sujets traités subséquemment. On discutera ainsi de programmation fonctionnelle et du langage de programmation Scheme. Ce chapitre fera mention donnera un aperçu général de ces langages et permettra de saisir les concepts fondamentaux du langage Scheme. Aussi, une discussion sur l'industrie des jeux vidéo et des défis découlant du développement sera présentée.

La deuxième partie du mémoire portera sur des extensions faites au langage Scheme qui ont été utilisées dans le but d'améliorer le développement de jeux vidéo. On discutera de programmation orientée objet et d'un système d'objets conçu pour répondre aux besoins de la programmation de jeux vidéo. Aussi, un

système de coroutines conçu dans les mêmes optiques sera présenté.

La troisième partie du mémoire portera 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. Aussi, une discussion sur l'utilisation des travaux reliés sera faite. On y discutera de l'utilisation de d'autres langages pour le développement de jeux vidéo et citera des exemple d'utilisation du langage Scheme dans des jeu vidéo commerciaux et de l'expérience tirée de cette utilisation par les développeurs.

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

CHAPITRE 2

DÉVELOPPEMENT DE JEUX VIDÉO

Importance du marche des JV

Diversite des jeux (genres, plates-formes)

Charactéristiques 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 venu des console modernes la plateforme offrant les meilleurs 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 [1]

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 entré et de généré du code source (toujours sous forme de listes (données Scheme)). Revient a 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 dévaluation 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 (lambda (res-f) (+ 1 (- 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 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.

Possibilité de faire le déboggage 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 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

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 reliés 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

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

Séparation entre le travail fait durant l'expantion 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 ¡id¿ ¡thunk¿)

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 cascadés ?

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 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

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 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

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 résoudres ces derniers

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.

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 artivicielle, niveaux, schema d'animations plus complexe, etc...

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

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

Utiliation des fonctions génériques

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