Hubert Renaud-Roland
Bailleul Jonathan
Bazzaro Fabrice
Pujos Michaël
Degand Benoît

Vendredi 5 mai 2000

Implémentation d'une bibliothèque graphique en Common Lisp

Glos

Mémoire

Maîtrise Informatique
Université Bordeaux I

Chargé de TER : Robert Strandh
Responsable de projet : Pascal Guitton

Résumé

Glos est une bibliothèque graphique entièrement écrite en ANSI Common Lisp dont la syntaxe et les fonctionnalités sont inspirées d'OpenGL, la référence dans ce domaine. Glos implémente un sous-ensemble utilisable d'OpenGL suffisamment performant pour animer des objets 3D en temps réel, visualisés via le système de fenêtrage X. Ce document présente en détail les choix retenus pour l'obtention de performances optimales sous l'environnement CMU Common Lisp. De plus, il confronte avantages et inconvénients du langage pour l'écriture de ce type de bibliothèque. Enfin, une analyse détaillée de performances conclut que la bibliothèque Glos est environ 3 à 4 fois plus lente que MesaGL (implémentation d'OpenGL écrite en C), selon les applications.

Table des matières

Chapitre 1

Introduction

Le langage Common Lisp nous parait intéressant dans la mesure où il permet d'aborder de manière claire et concise le développement de nombreux programmes. Il semble cependant avoir acquis une réputation selon laquelle il serait exclusivement spécialisé dans la résolution de problèmes d'intelligence artificielle. Ce serait alors un excellent langage d'apprentissage et un remarquable outil de recherche universitaire, mais certainement pas un langage capable d'aborder des sujets plus généraux et, par exemple, utiles au monde professionnel.

Les arguments récurrents qui alimentent cette thèse seraient les grands besoins en mémoire de Common Lisp et sa lenteur chronique. Nous ne partageons pas ce jugement un peu hâtif, qui s'est probablement fondé à l'époque où la mémoire et le temps machine coûtaient trop cher pour envisager l'option Common Lisp dans le cadre d'applications pouvant être développées dans un langage plus économe et proche de la machine. Common Lisp est en effet un langage ancien, mais il a beaucoup évolué en fonctionnalités et en possibilités depuis sa création, de même que la puissance des ordinateurs.

D'ailleurs, une étude récente de la NASA (*"Lisp as an Alternative to Java"*, Eran Gat) montre que Common Lisp n'est pas spécialement plus gourmand en temps et en mémoire que ses homologues actuels. De plus, un certain nombre d'industriels et de développeurs, bien qu'assez isolés, y recourent avec succès dans des domaines très variés. Ils manquent rarement de souligner les avantages que procure Common Lisp, notamment en termes de progression linéaire du développement et de réduction du risque d'erreurs.

Donc, porter aujourd'hui un regard neuf sur Common Lisp pourrait s'avérer très fructueux. En tous cas, il semble intéressant d'étudier l'influence que peut avoir ce langage sur l'écriture d'un programme assez complet, ne serait-ce que pour faire progresser nos propres connaissances en programmation. Nous avons donc choisi d'implémenter une bibliothèque graphique car c'est un sujet d'étude qui nous tient à coeur et qui fait intervenir des aspects variés de la programmation : calcul mathématique, interactions avec le matériel, ou algorithmique standard.

Une bibliothèque graphique nécessite surtout un certain niveau de performance : il nous semble intéressant de voir si, en utilisant Common Lisp, on peut obtenir des performances acceptables. En fait, il est surtout intéressant de quantifier d'une part les avantages apportés à l'écriture et à l'utilisation d'une telle bibliothèque, et d'autre part le coût de ces avantages en termes de performances par rapport à une implémentation classique.

Pour que ce rapport indique que le choix de Common Lisp est bien raisonnable, nous allons essayer de réduire autant que possible le coût en performances. En conséquence, nous allons d'abord envisager des optimisations d'ordre algorithmique et ensuite des optimisations spécifiques au compilateur choisi. Par ce fait, nous allons devoir évaluer la nécessité et l'opportunité de ces optimisations, leurs modalités de mise en oeuvre, et aussi dans quelle mesure elles peuvent porter atteinte à la clarté et à la concision du code.

Chapitre 2

Bibliothèque graphique en Common Lisp

2.1 Qu'est ce que Glos ?

Glos est une bibliothèque graphique écrite en Common Lisp, fonctionnant avec le système de fenêtrage X, et fortement inspirée d'OpenGL (développé par Silicon Graphics). A la différence d'OpenGL qui est capable d'utiliser l'accélération matérielle pour le rendu, Glos effectue le rendu en mode logiciel. Glos fournit un ensemble de primitives (API) au programmeur pour spécifier facilement des scènes en 3 dimensions composées d'objets. Les objets sont décrits à l'aide d'un petit ensemble de primitives graphiques : points, lignes et triangles. Glos effectue le tracé de la scène en effectuant tous les calculs nécessaires en fonction des paramètres spécifiés par l'utilisateur (éclairage, point de vue de la scène. . .).

2.2 Le choix de Common Lisp

Créé par John MacCarthy en 1959, Lisp est le plus ancien des langages de programmation après Fortran. Bien qu'il ait été créé au départ pour résoudre des problèmes liés à l'intelligence artificielle, ce langage est aujourd'hui capable d'aborder des sujets aussi variés que ses concurrents actuels, pourtant beaucoup plus récents. Cette diversité et cette modernité tiennent beaucoup au fait que les programmeurs, qui utilisent Lisp dans des situations concrètes et variées, peuvent d'eux-mêmes enrichir ce langage en fonction de leurs besoins. Ces enrichissements (macros, paquetages) peuvent ensuite profiter à la communauté Lisp, très structurée, soit par intégration dans la norme ANSI suite à un vote du comité de normalisation, soit par une simple mise à disposition dans un dépôt de sources. ANSI Common Lisp est donc le résultat de l'uniformisation des nombreux dialectes qui ont émergé, et en même temps un condensé de l'expérience des programmeurs qui l'utilisent activement.

2.2.1 Points forts du langage

2.2.1.1 Gestion homogène des types de données

- Les variables manipulées n'ont pas de type précis a priori, seules les valeurs qu'elles peuvent prendre en ont. En effet, la spécification ANSI du langage contraint par défaut tout système Common Lisp à une vérification de type à l'exécution. En conséquence, le programmeur n'a pas besoin d'insérer systématiquement des déclarations de type qui alourdissent le code et compliquent la syntaxe.
- Aussi, de nombreuses structures de données existent déjà dans le langage (listes, tables de hachage, tableaux, etc.). La généricité des types de données étant acquise, il est très facile de combiner ces structures pour en créer de plus complexes. De ce fait, le programmeur n'a pas à consacrer du temps (et beaucoup de code) à créer les types de données dont il peut avoir besoin. En outre, on évite de multiplier les implémentations spécialisées et incompatibles d'un même type trivial.
- Enfin, la sémantique par référence uniforme permet de masquer définitivement la notion de pointeur : nous manipulons tous les objets sans se soucier de leur représentation interne.

2.2.1.2 Gestion automatique de la mémoire

Il est inutile de réserver de la mémoire pour instancier un objet : le système Common Lisp se charge lui-même des allocations et désallocations, tout comme de faire intervenir le ramasse-miettes. De plus, l'absence de pointeurs explicites empêche d'allouer de la mémoire pour un objet d'un type de base.

Le code est donc exempt de telles instructions et gagne en lisibilité. Il gagne surtout en sécurité : une petite erreur dans la gestion manuelle de la mémoire peut entraîner des arrêts, voire même engendrer d'autres erreurs moins reconnaissables dans d'autres sections du programme. Il s'agit bien sûr d'erreurs qui surviennent pendant l'exécution d'un programme, dont la correction est délicate. En conclusion, il est difficile d'affirmer si un programme confiant la gestion de la mémoire au programmeur est sûr.

2.2.1.3 Concision générale du code

Le fait que les mécanismes précédemment cités soient pris en charge par le langage permet au code d'être libéré d'un grand nombre d'instructions : en Common Lisp, des sections entières de code que l'on a l'habitude de voir dans certains langages n'ont pas lieu d'être, voire se résument à quelques lignes.

Cette concision est encore accrue par l'usage des macros. Ce mécanisme permet d'expliciter les idiomes du domaine de programmation, c'est à dire de remplacer certains motifs de code fréquemment employés dans le contexte courant par un appel de macro. Cet appel s'apparente grossièrement à un appel de fonction dans la mesure où il possède un nom porteur de sens. Cependant, il prend essentiellement du code non évalué en argument.

Ce mécanisme de macros, original par sa portée, permet donc à la sémantique du code de prendre le dessus sur sa syntaxe.

2.2.2 Contreparties

Ces avantages concédés au programmeur ont cependant un certain coût. La vérification de type se déroule pendant l'exécution du programme et est notoirement responsable de la lenteur de certains d'entre eux.

Aussi, la gestion transparente de la mémoire oblige le compilateur à manipuler des descripteurs (pointeurs), impliquant des allocations dynamiques dans le tas et des accès mémoire : dans ces conditions, le ramasse-miettes est amené à intervenir. Les spécifications de ces mécanismes dépendent de l'implémentation du compilateur retenu.

Précisons que l'usage des macros ne coûte rien : leur expansion étant effectuée à la compilation, elles n'ont aucune incidence négative sur la vitesse d'exécution du programme.

2.2.3 Les raisons de notre choix

La concision qu'offre Common Lisp n'est pas un simple agrément pour les lecteurs de code : la compréhension directe et rapide qu'elle permet profite en premier lieu au programmeur. En effet, il peut écrire rapidement et directement un code qui répond aux problèmes qu'il doit résoudre sans se perdre dans des complications syntaxiques. Il a donc une meilleure vue d'ensemble sur son programme, et peut plus naturellement envisager des extensions ou des améliorations.

Il est d'ailleurs notoire que le code d'un programme réalisé en Common Lisp est beaucoup plus court et lisible que celui de la plupart de ses homologues : en tout cas, notre expérience personnelle tend à affirmer cette hypothèse.

Bien que certaines facilités du langage aient un coût, il nous semble intéressant de déterminer dans quelles conditions précises ce coût est vraiment excessif, et dans quelles autres nous pouvons profiter pleinement des atouts de Common Lisp. Ce coût peut aussi être justifié si notre bibliothèque parvient à offrir en contrepartie des facilités inaccessibles à ses homologues.

2.3 Projets analogues

2.3.1 Projets du domaine universitaire ou public

Il existe assez peu de projets en Common Lisp dans le domaine de la 3D. En effet, les dépôts de code source (ALU[1], CMU AI[2], CLOCC[3]) ne recensent aucun projet majeur de ce type.

Les projets examinés offrent généralement des possibilités assez réduites, bornées par le tracé de polygones, l'application de transformations standards et l'affichage par CLX. Nous citerons par exemple Viewing de Paul Emerson et 3D geometry de Daniel LaLiberté, que nous avons contacté pour obtenir la dernière version de son projet. Nous n'avons pas trouvé de bibliothèque 3D exhaustive ou performante à la manière d'OpenGL, effectuant par exemple des optimisations, de l'éclairage ou du découpage de faces.

Par contre, nous trouvons de plus en plus fréquemment des interfaces OpenGL, traduisant des appels de primitives Lisp en MesaGL (par exemple : GL de Richard Mann). De tels produits offrent un accès à la totalité des primitives OpenGL, puisque la bibliothèque 3D utilisée est en réalité MesaGL. De plus, les performances obtenues sont optimales car le code exécuté est du C, tirant parti de drivers optimisés et gérant les accélérations 3D.

Cependant, ces interfaces n'aident en rien à savoir de quelle manière le langage Common Lisp peut influencer l'écriture d'une bibliothèque 3D, problème auquel notre bibliothèque sera plus en mesure d'apporter des éléments de réponse.

2.3.2 Produits commerciaux

- Miraï, le successeur de N-World, est développée par Nichimen Graphics. C'est une suite de logiciels comprenant un modeleur 3D, le point fort du produit, accompagné d'outils d'animation, de modélisation et de dessin. Elle est principalement destinée au développement de jeux vidéo, domaine dans lequel elle est reconnue. Miraï est essentiellement programmée en Common Lisp à l'aide d'Allegro CL : CLOS et le MOP[4] sont intensivement utilisés. Le reste du code (10%) concerne des routines de bas niveau et le programme de rendu 3D.

  Ce produit est le résultat d'un long développement qui a impliqué plusieurs sociétés successives (III, Symbolics Inc, Nichimen). Ce fut d'abord un modeleur 3D, pionnier du genre dans les années 80, qui

[1] Association of Lisp Users
[2] Carnegie Mellon University Artificial Intelligence Repository
[3] Common Lisp Open Code Collection
[4] Meta-Object Protocol

a évolué pour se spécialiser dans le développement de jeux vidéo. Il a servi à modéliser les personnages du jeu Mario64 (Nintendo) ou les objets 3D répertoriés par la base de données de Viewpoint DataLabs. Aussi, une des premières versions du produit a servi aux effets spéciaux du film Tron (Walt Disney).

Nous avons eu connaissance de Nichimen Graphics grâce au site de Franz Inc[5]. Nous avons pris contact avec M. Luca Pisati, Directeur du développement, qui nous a fourni des informations plus précises que celles à disposition sur le site de Nichimen Graphics[6]. Nous avons aussi discuté en détail des choix et des fonctionnalités de notre projet : il semble que pour obtenir un niveau de performance intéressant, l'usage de routines de bas niveau en C voire en assembleur soit préférable. Cependant, nos options permettent d'obtenir des performances raisonnables dans le cadre de notre projet.

- GOOL (Game Object Oriented Language) est un langage de programmation destiné au développement de jeux vidéo. Il a été créé par la société Naughty Dog pour son propre usage, et a servi aux jeux Crash Bandicoot et Way of the warrior. Ce dialecte de Common Lisp dispose de son propre compilateur, capable d'appeler certaines fonctions C de manière transparente.

  Dans ces jeux en 3D temps réel, GOOL a servi à définir les objets (animation, comportement par intelligence artificielle, contrôle), ainsi que la plupart des sections non critiques du code (menus, démonstrations). Cependant, les calculs de collision, certains calculs mathématiques et certaines parties du rendu graphique ont été programmés en C ou en assembleur.

  Remarque : ces informations sur GOOL proviennent d'un article d'OBJECT magazine du mois de mai 1997 dont une reproduction nous a été transmise par Franz Inc.

En résumé, le succès de certains projets commerciaux nous encourage à entreprendre le développement de notre bibliothèque graphique en Common Lisp. Ce serait d'autant plus utile que, à première vue, il n'y a pas de projet équivalent dans le domaine public.

[5] `www.franz.com`
[6] `www.nichimen.com`

Chapitre 3

Choix effectués pendant le projet

3.1 Outils

3.1.1 CMU Common Lisp : l'environnement de développement

Le choix de CMU Common Lisp comme environnement de développement aurait pu être un choix par défaut car nous avions déjà appris à l'utiliser. En fait, cet environnement répond particulièrement à nos attentes dans la mesure où :

1. Développé par l'Université de Carnegie Mellon, il est gratuit. En outre, il est disponible sur la plupart des plates-formes Unix.
2. Le compilateur fourni (Python) est reconnu comme capable d'optimiser efficacement le code généré.
3. Il est fourni en standard avec une version performante de CLX, l'interface vers le système de fenêtrage X.
4. Il intègre des outils indispensables tels qu'un débogueur, un profileur très complet, et un gestionnaire de dépendances (Defsystem).
5. Il offre aussi la possibilité de partager des objets avec d'autres systèmes et d'appeler des fonctions en C (éventuellement certaines routines d'affichage ou d'accès mémoire).

A titre de comparaison, nous avons commandé à Franz Inc. une version gratuite de leur système Allegro CL, déjà employé dans certains produits commerciaux ayant trait à la 3D. Nous avons effectué quelques tests d'affichage à titre de comparaison : nous avons constaté des défauts de tracé et surtout une lenteur perceptible. Cela nous a encouragé à retenir l'environnement CMU Common Lisp qui semble être la meilleure option dans l'immédiat (il n'est pas à exclure que certaines optimisations spécifiques à Allegro CL puissent influer sur les tests effectués).

3.1.2 Le système d'affichage CLX

CLX est la bibliothèque Common Lisp servant d'interface de bas niveau vers le protocole X. Son code est relativement simple : la bibliothèque analyse les arguments qui lui sont transmis et les envoie ensuite au serveur X par un système de sockets. CLX est aussi rapide que les fonctions C en ce qui concerne les appels au serveur X.

Très tôt dans le projet, nous avons dû trouver une méthode pour afficher des graphismes animés dans une fenêtre en utilisant au mieux CLX.

Deux choix se sont offerts à nous, sachant que nous voulions de bonnes performances, et le plus grand contrôle possible sur l'affichage individuels de pixels :

- Utiliser la structure de donnée `pixmap` : des ressources du serveur X dans lesquelles nous pouvons dessiner des points, des lignes et des polygones, grâce à un ensemble de primitives CLX bien définies. En particulier, l'accès à un pixel nécessite un appel à la fonction `draw-point`.
- Utiliser la structure de donnée `image` : le type `image` contient un tableau à deux dimensions contenant des pixels et décrivant une image quelconque. Cette structure offre une grande liberté quand au format de données des pixels dans le tableau. De plus l'accès en lecture/écriture d'un pixel se fais par simple accès dans le tableau. L'image est ensuite affichée avec la primitive `put-image`.

Nous avons donc, après quelques tests, choisi d'utiliser le type `image` : d'abord, il nous offre un plus grand contrôle au niveau des pixel, mais surtout il ne nécessite aucun appel de fonction pour écrire la couleur d'un pixel. Nous n'avons pas testé directement le type `pixmap` en ce qui concerne la rapidité d'affichage pour l'animation, mais un extrait de la documentation de MesaGL confirme le fait que le type `pixmap` est généralement moins efficace que le type `image` : *" a pixmap is faster when drawing simple lines and polygons while an XImage is faster when Mesa has to do pixel-by-pixel rendering. If you need depth buffering the XImage will almost surely be faster."*[1]

3.1.3 Defsystem : le gestionnaire de dépendances

Très vite le code de notre projet s'est réparti sur plusieurs fichiers regroupant des fonctions suivant leur champ d'application, fichiers entre lesquels des relations de dépendance se sont dessinées. Nous avons donc eu besoin

[1] fichier README.X11 de la documentation MesaGL

d'employer un outil de gestion de dépendances offrant des fonctionnalités proches de `GNU make` pour charger ou compiler les fichiers de notre projet, et écarter les risques d'inclusions multiples.

Nous avons demandé sur `comp.lang.lisp` si un tel outil existait : M. Marco Antoniotti, membre de l'équipe de développement de CMU Common Lisp, nous a signalé l'existence de `Make`, une implémentation portable de Defsystem, disponible sur le dépôt de sources du CLOCC. A notre demande, il nous a fourni des exemples d'utilisation, ainsi qu'un lien vers une documentation complète hébergée sur le site de l'EncyCMUCLopedia[2]. Nous avons été étonnés de ne pas trouver des fonctionnalités de ce genre dans la norme ANSI du langage. Cependant, Defsystem est très reconnu et largement utilisé par la communauté Common Lisp : il était d'ailleurs fourni (sans documentation) avec notre distribution de CMU Common Lisp.

3.1.4 Ilisp : le mode Emacs pour Common Lisp

Nous avons adopté ce paquetage Emacs Lisp dès le début de notre projet. En effet, il est référencé dans tous les sites relatifs à Common Lisp (dont `cons.org` et ALU) comme étant un outil indispensable pour développer en Common Lisp sous Emacs.

Il est en effet capable de gérer les interactions entre un environnement Common Lisp et les tampons Emacs correspondants, et ce pour la plupart des environnements Scheme ou Common Lisp (dont CMU Common Lisp). Il présente de nombreux avantages : nous citerons par exemple les facilités de compilation et d'évaluation, ainsi que la recherche de symboles Common Lisp en corrélation avec l'HyperSpec [8].

3.1.5 Cover : un outil de couverture de code

Afin de s'assurer de la correction d'un programme, il est nécessaire de vérifier sa couverture. En effet, les tests effectués pour vérifier le bon fonctionnement d'un programme ont peu de valeur si nous ne sommes pas certain d'avoir exécuté toutes les sections de son code. Nous nous sommes donc procuré le programme Cover, de Richard Waters, référencé sur l'ALU (le lien ne fonctionnant pas, nous avons effectué une recherche sur `DejaNews` pour trouver le mél de l'auteur, qui nous a ensuite transmis la dernière version de son programme).

Cependant, nous ne l'avons pas utilisé car nous n'avons pas entrepris d'étude formelle de couverture. En fait, pour nos besoins d'optimisation, nous avons utilisé intensivement le paquetage `profile` de CMU Common Lisp pour rechercher les sections de code consommatrices de temps machine et de mémoire. Nous avons donc parcouru la totalité du code de

[2] `http ://cvs2.cons.org :8000/cmucl/doc/EncyCMUCLopedia/`

la bibliothèque à la recherche d'allocations mémoire indésirables, aidés par quelques fonctionnalités de couverture de code qui nous permettent notamment d'évaluer le nombre d'appels de chaque fonction et leur consommation en temps et en mémoire.

En résumé, comme nous avons optimisé et testé dans le détail le code de notre bibliothèque, nous avons pu, en pratique, évaluer de manière assez fine sa couverture. Pour des raisons de temps, nous nous en tiendrons là.

3.2 Démarches

3.2.1 Matrices de transformation

Leur développement est incontournable pour toute bibliothèque 3D. Elles permettent de représenter les transformations dans l'espace, de les combiner et de les appliquer à des points. Ces calculs nécessitent un grand nombre d'opérations arithmétiques, et consomment un temps machine non négligeable dans le processus global d'affichage. Ils doivent donc donner lieu à des optimisations efficaces.

Conformément aux spécifications d'OpenGL, nous avons adopté un système de coordonnées homogènes de dimension 4×4 : c'est une obligation si nous voulons pouvoir manipuler toutes les transformations sous forme de matrices. En effet, la translation dans l'espace 3D n'est pas exprimable par une matrice de dimension 3×3. Cette conformité vis-à-vis des spécifications d'OpenGL nous permet de bénéficier des matrices de transformation déjà calculées et normalisées à son intention : il en existe de nombreuses variantes dont les divergences tiennent souvent aux choix du repère.

Le but de ce module est donc de fournir des primitives pour manipuler ces matrices de transformation. En résumé, il s'agit de :

- Modifier la matrice de transformation courante 4×4 en la multipliant par la matrice d'une transformation paramétrée 4×4.
- Calculer la projection d'un vecteur, i.e. multiplier la matrice de transformation courante 4×4 par la matrice du vecteur 4×1.

Par commodité, nous allons surtout traiter de la multiplication de matrices 4×4, car la projection n'en est qu'une spécialisation.

La démarche adoptée pour développer ce module fut la suivante : nous en avons d'abord réalisé une implémentation triviale qui était immédiatement nécessaire au projet. Puis, nous avons commencé à développer un module spécial pour chercher à mettre en œuvre des optimisations sans être contraint par les nécessités induites par la nature du projet (effets de bord, calculs d'indice, et surtout optimisations de compilation).

Ce module expérimental nous a permis de réaliser et de tester un certain nombre d'optimisations algorithmiques : nous avons appliqué certaines d'entre elles. Cependant, nous en avons envisagé d'autres qui n'ont pas été retenues. En effet, il était difficile d'en tirer parti dans notre cas particulier.

3.2.1.1 Optimisations appliquées dans le projet

3.2.1.1.1 Spécification statique du type manipulé

Nous avons retenu le type `single-float` (cf 4.1.1). Sans entrer dans les détails des optimisations de compilation, la spécification de types dans les sections critiques du code est obligatoire pour obtenir une bonne vitesse d'exécution. En effet, dès le début du projet, nous avons testé deux programmes effectuant des calculs matriciels simples. Si a priori, celui codé en Lisp tourne beaucoup plus lentement que son homologue C, l'écart de performances devient peu significatif dès lors que nous déclarons les types des arguments d'entrée/sortie dans le code Lisp (pas d'optimisations de compilation dans les deux cas). Cette optimisation, qui dépend de l'implémentation du compilateur, est donc considérée comme préalable à toute autre d'ordre algorithmique.

3.2.1.1.2 Précalcul de l'indice d'élément de matrice

Tout d'abord, essayons de quantifier les opérations arithmétiques nécessaires à une multiplication de matrices carrées de dimension n. Il faut effectuer n^3 multiplications de flottants, n^3 additions de flottants, $2n^3$ accès en lecture et n^2 accès en écriture. Regardons les calculs d'adresse que peuvent requérir ces derniers : pour accéder à un élément $mat[i,j]$, nous devons vraisemblablement calculer l'adresse $mat + i \times n + j$.

Nous avons eu l'idée d'utiliser un tableau unidimensionnel pour représenter nos matrices : l'accès à un élément devient alors direct $mat + x$, et économise une multiplication et une addition à chaque accès. Ce calcul d'indice s'effectue grâce à l'utilisation d'une reader-macro pour implémenter l'accesseur : si les indices fournis sont de type numérique, le calcul est effectué lors de la compilation. Dans le cas contraire, il est reporté à l'exécution du programme (ce qui nous ramène au cas général non optimisé).

En résumé, cette optimisation triviale peut nous permettre de gagner $2n^2 + 4n^3$ opérations intervenant dans un calcul d'adresse. Nous l'avons donc adoptée sans hésitation : les tests évaluant le gain de temps effectif montrent que nous divisons presque par deux le temps total de calcul. Cette optimisation semble donc adéquate.

Remarque : pour reporter le calcul d'indice à la compilation, il faut utiliser une série de macros dont les arguments sont forcément numériques. En pratique, cela donne au code un aspect un peu dérangeant, moins agréable à lire qu'un code équivalent ne tenant pas compte de cette contrainte très particulière. Mais le gain de temps obtenu justifie ce choix d'implémentation.

3.2.1.1.3 Suppression d'itérations

Une optimisation considérée comme triviale par les programmeurs de bibliothèques de matrices est la suppression des itérations. En effet, pour calculer une multiplication de 2 matrices $n \times n$, il est nécessaire d'itérer sur 3 niveaux : nous avons donc n^3 instructions de comparaison et d'incrément. Cela occasionne une perte de temps effroyable et surtout complètement inutile puisque nous nous sommes déjà accordés à utiliser des matrices de dimension finie 4×4.

La solution générale à ce problème d'itérateurs est la génération statique du code : les programmeurs C ou Fortran n'hésitent pas à calculer ce code par leurs propres moyens pour un nombre fini de dimensions réduites. En Common Lisp, grâce aux macros, cette génération se fait naturellement en utilisant les mécanismes standard du langage. Elle est d'ailleurs envisageable quelque soient les dimensions des matrices considérées.

Les tests montrent que le calcul de multiplications de matrices se déroule plus de 4 fois plus vite lorsque nous réalisons cette optimisation, qui s'avère donc absolument indispensable.

3.2.1.1.4 Effets de bord

Le but initial de ce module est de fournir des primitives pour modifier la matrice de transformation courante, et non d'en calculer une nouvelle. Si la plupart des optimisations déjà envisagées sont valides dans le cas général, il faut considérer comment nous allons procéder dans notre cas particulier.

Voici comment nous procédons pour modifier la matrice de transformation courante (MTC) lorsque nous voulons lui appliquer une transformation spatiale (MS) : $MTC := Copie(MTC) \times MS$.

Nous sommes forcés de la copier pour effectuer la multiplication : si nous la modifions au fur et à mesure, le résultat final serait faux. Enfin, nous l'affectons au résultat.

Remarque : la copie se fait dans un espace de mémoire déjà alloué pour éviter des allocations intempestives (technique des Pools, cf. 4.5.2.1).

3.2.1.1.5 Optimisations manuelles

Les matrices représentant les transformations dans l'espace sont "creuses" (i.e. comportent beaucoup de zéros), et contiennent des constantes ou des paramètres numériques. Donc, nous avons naturellement modifié les algorithmes de multiplication de matrices pour optimiser ces transformations : chacune est donc traduite par une fonction spécialement optimisée pour effectuer le moins de calculs et d'accès possible.

Nous verrons dans ce chapitre que nous avons envisagé une méthode automatique pour générer de telles fonctions.

3.2.1.2 Protocole de test employé et résultats

Nous avons effectué des tests pour nous décider à appliquer ou non les optimisations que nous trouvions au fur et à mesure du projet. Nous avons estimé qu'il était utile de faire de nouveaux tests sur la version finale du projet afin de faire une synthèse des gains de temps permis par les optimisations algorithmiques appliquées (remarque : c'est à partir de ces derniers résultats que nous avons commenté dans ce chapitre l'intérêt de chaque technique).

Pour chaque technique, nous avons choisi des sections de code effectuant les multiplications de matrices 4×4, et nous les avons compilés en `speed 0` pour ne pas prendre en compte les optimisations du compilateur. Nous avons ensuite, via la fonction `time`, chronométré l'opération suivante : 10000 multiplications d'une matrice quelconque par elle-même.

- Technique du calcul d'indice. Nous avons pris l'algorithme de multiplication de la version finale du projet, qui emploie cette technique : 18s. Puis, nous avons remplacé l'appel à la reader-macro par celui d'un accesseur qui effectue le calcul d'indice à l'exécution : 38s.
- Technique de suppression d'itérations. Nous avons pris un algorithme employant cette technique et dépourvu de calcul d'indice : 15s. Puis, nous avons pris une fonction triviale de calcul par itérations successives sur le même modèle : 70s.

Remarque : Il est très important de noter l'influence des optimisations de compilation sur le temps d'exécution final : en `speed 3` l'algorithme retenu s'exécute en 0.4s (au lieu de 18s), alors que l'algorithme par itération prend 2.5s. (au lieu de 70s).

Critiques : les résultats des tests sont très tranchés, ce qui semble montrer le bien-fondé de nos optimisations (0.4s avec, 2.5s sans). Cependant, il y a matière à débattre. En effet, nous n'avons pas vérifié si, en utilisant un tableau bidimensionnel pour implémenter les matrices, le compilateur n'effectuait pas par lui-même le calcul d'indice. Dans ce cas, nous aurions perdu notre temps à implémenter cette optimisation. Cependant, dans cette hypothèse, il est probable que le compilateur ait besoin d'indices numériques pour opérer : alors, notre optimisation serait quand même profitable puisqu'elle nous a amené à choisir cette option.

Aussi, nous n'avons pas réellement optimisé l'algorithme par itérations successives pour la compilation. L'algorithme final, lui, à été mis au point lorsque nous avons bien compris la plupart des techniques d'optimisation pour Python. C'est pourquoi sa compilation en `speed 3` se fait sans notes, ce qui n'est pas le cas de l'algorithme par itérations.

Lorsque nous réalisons l'efficacité des optimisations de Python, il se peut que le résultat final (0.4s/2.5s) soit à reconsidérer. Cependant, nous préférons croire que nos optimisations sont justifiées et efficaces, et qu'elles ne peuvent pas être gommées ou trop amoindries par les facultés d'optimisation de Python.

3.2.1.3 Optimisations envisagées

Dans notre module expérimental, nous avons notamment envisagé de calculer automatiquement le code le plus optimisé possible pour une fonction de multiplication de matrices (de dimensions compatibles). Les optimisations se font par réduction, autant que possible, du nombre d'opérations arithmétiques et du nombre d'accès aux matrices. Cela implique la suppression d'itérations et surtout la capacité à s'adapter aux profils des matrices à manipuler.

Pour spécifier le profil des matrices à considérer, nous fournissons des matrices génériques de même dimension contenant :

- `NIL` pour signaler un terme quelconque (et forcer l'accès à la matrice).
- Des constantes numériques.
- Des paramètres dont la liste doit être spécifiée, afin qu'ils figurent dans le bon ordre comme arguments de la fonction générée.

Ainsi, grâce à une macro, nous pouvons générer le code d'une fonction de multiplication optimale et spécifique à un profil donné. Pour cela nous effectuons à la compilation tous les calculs arithmétiques possibles et en conséquence nous supprimons les accès inutiles aux matrices. Il reste cependant à vérifier que nous utilisons ces fonctions spécialisées sur des matrices du profil correspondant (non traité).

Par exemple, nous avons pu facilement calculer le code des fonctions de transformation. Il a suffi de fournir en argument de la macro deux matrices 4×4 :

1. La première représente la matrice de transformation courante, supposée purement générique, et se traduisant par une matrice de profil remplie de `NIL`.
2. La seconde représente la matrice de la transformation, traduite par la matrice de profil correspondante.

Nous aurions donc pu générer la quasi-totalité des fonctions du module final de manière homogène et concise s'il n'y avait pas l'obligation de se plier à certaines contraintes particulières : effets de bord, calcul d'indice, optimisations de compilation. Il est possible d'adapter notre code pour qu'il satisfasse aussi ces contraintes, mais le temps nous a manqué.

Aussi, il est assez facile pour un programmeur humain de taper à la main le même code optimisé correspondant à ces transformations : c'est ce que nous avons fait (ce n'aurait pas été raisonnable dans le cas de la multiplication générique).

En réalité, cette optimisation ne visait pas seulement la factorisation de code : nous espérions pouvoir l'utiliser facilement pour réduire considérablement le volume de calcul matriciel de notre bibliothèque. Si l'idée nous a servi pour appliquer des transformations à la matrice de transformation courante, l'essentiel des calculs survient en réalité lorsque nous insèrons des points dans le système.

Ce calcul implique alors la matrice de transformation courante et le vecteur implicite du point. Le profil de la matrice devient de plus en plus générique au fur et à mesure que les transformations s'y accumulent, le vecteur implicite du point est de type quelconque (sauf la dernière composante qui vaut toujours 1 dans notre projet).

Nous avons eu une idée que nous n'avons pas testée et qui soulève sans doute autant de problèmes qu'elle ne peut en régler. Imaginons le code d'un programme d'essai qui utilise notre bibliothèque. Lors de sa compilation, nous comptabilisons toutes les transformations appliquées à la matrice de transformation courante, et nous déterminons le profil de cette matrice à chaque fois que nous détectons une projection de point (typiquement, lors d'une insertion d'un objet, nous effectuons des milliers de ces projections).

Ensuite, nous calculons les fonctions optimisées correspondantes et nous insèrons leur code là où elles sont utiles.

Les problèmes à résoudre sont :

- Modifier certaines primitives de la bibliothèque pour permettre à ces mécanismes de se coordonner à la compilation. C'est le point le plus difficile, car l'ordre d'expansion des macros n'est pas défini et qu'elles ne peuvent pas modifier leur expansion en fonction de valeurs non statiques.

- Déterminer le profil d'une accumulation de matrices de transformation. A priori, cela n'est pas impossible : les valeurs numériques et `NIL` se transmettent d'un niveau à un autre, seuls les paramètres du niveau supérieur sont conservés.

Ce n'est qu'une idée, qui nécessite un investissement de temps considérable, et qui peut s'avérer décevante si la matrice de transformation courante tend à devenir quelconque. D'ailleurs, il est probable qu'une analyse de texte brutale soit nécessaire pour mettre en œuvre les mécanismes décrits. Cependant, cela n'est peut-être pas irréalisable, alors que de nombreuses opérations arithmétiques sont en jeu dans un module qui constitue un des points critiques du projet.

3.2.1.4 Pourquoi ne pas utiliser Matlisp ?

Sur `comp.lang.lisp`, M. Tunc Simsek nous a signalé l'existence de la bibliothèque `Matlisp` dans le CLOCC, et nous a expliqué ses caractéristiques. C'est une interface vers une bibliothèque Fortran très optimisée, qui effectue des calculs rapides et est capable de contourner les problèmes d'itérations pour un nombre fini de dimensions. Les effets de bord sont aussi gérés.

Cette bibliothèque est très utilisée et reconnue par la communauté Lisp, elle est sans doute un très bon choix lorsqu'on souhaite disposer d'une bibliothèque utilisable et suffisamment efficace. Cependant, elle ne permet pas de gérer des profils de matrices et n'offre pas de possibilités d'optimisations personnalisées (selon M.Himsek). C'est pourquoi nous l'avons finalement écartée. Nous aurions aimé pouvoir la tester et l'évaluer par nous-mêmes, mais hélas nous avons découvert cette bibliothèque durant la dernière semaine de développement.

3.2.2 Paquetages

Les paquetages sont un mécanisme normalisé ANSI du langage Common Lisp. Ils permettent d'encapsuler une application et de lui procurer son propre espace de noms pour ses symboles. Le programmeur peut donc développer son application sans craindre que ces derniers n'entrent en conflits avec ceux définis dans l'environnement de l'utilisateur. De plus, il peut définir quelles fonctions seront accessibles et masquer les autres : cela permet d'avoir un minimum de sécurité, ou en tout cas de clarifier l'interface du programme.

Notons aussi qu'il est possible d'importer les symboles d'un paquetage dans l'espace de noms courant : cela permet de ne pas utiliser une notation préfixe (comme `glos:`) pour accéder aux symboles. Il est important de comprendre que, comme il s'agit d'un mécanisme statique, la plupart des instructions de définition de paquetages ne sont traitées que par le Reader.

Lorsque nous avons découvert ce mécanisme, nous avons surtout été intéressés par la possibilité de travailler dans des espaces de noms clos et sécurisés. Nous avons souhaiter les utiliser pour chaque fichier du projet : on peut masquer les fonctions et variables "privées", travailler sur chaque fichier sans trop normaliser les noms de symboles et accéder à ces derniers comme on accéderait à une classe. De toute évidence, nous avons voulu récupérer les fonctionnalités de mécanismes habituels en C/C++, langages auxquels nous étions trop habitués.

Toutes ces fonctionnalités peuvent être égalées de manière plus rigoureuse en Common Lisp : par exemple, les contextes de `let` et `flet` peuvent servir à définir des fonctions-outils et des variables communes à tout un groupe de fonctions. Ensuite, rien ne coûte de remplacer les appels `vec3:make`, `vec4:make` par `vec3-make` et `vec4-make`. D'ailleurs il est logique d'avoir un minimum de rigueur dans le nommage des primitives d'une même application.

Nous avons aussi envisagé une utilisation moins triviale de ces paquetages. En effet, nous avons remarqué que nous avions besoin de types de matrices 3×1, 4×1, 4×4 de flottants, et d'un type 4×4 générique : les vecteurs peuvent être considérés comme une spécialisation des matrices. Nous avons eu l'idée de générer statiquement des primitives pour tous ces types (addition, multiplication par un homologue, affichage,. . .) pour ne pas avoir à écrire du code redondant. Ces primitives sont bien sûr optimisées selon leur dimension et le type de données : le type employé est statique, et les itérations sont supprimées grâce aux techniques déjà abordées. Des macros imbriquées permettent cette génération de code, et encapsulent les primitives dans un paquetage dont le préfixe dépend des choix de dimension et de type.

Le paquetage sert essentiellement à ce que les primitives n'entrent pas en conflit dans le même espace de noms, car par souci d'uniformité leurs noms ne changent pas en fonction du type manipulé. Nous avons déjà vu que ce choix est peu adéquat.

Ce paquetage n'est donc pas défini via le Reader puisqu'il est généré après l'expansion de la macro : c'est possible grâce à une utilisation peu conventionnelle de certaines commandes de définition de paquetages.

Cependant, cette implémentation à été abandonnée pour plusieurs raisons :

- les paquetages n'ont rien à faire à l'intérieur d'une application. D'ailleurs, leur déclaration "contre nature" en est un bon indicateur.
- Toutes les primitives ne sont pas redondantes entre les diverses va-

riations du même type : souvent, pour un type donné, nous n'avons besoin que de quelques primitives. De même, nous avons souvent besoin d'enrichir spécifiquement le jeu de primitives d'un même type.

- Aussi, nous ne gérons pas les opérations entre variations d'un même type. Nous pourrions éventuellement spécifier à la macro les primitives de ce genre à rajouter.

La raison décisive est finalement qu'une implémentation simplifiée de primitives de matrices et de vecteurs s'écrit assez vite et sans trop de répétition de code. Toutefois, si nous faisions abstraction des éventuels problèmes d'optimisation de compilation, nous pourrions générer tout ce code par quelques appels de macros.

3.2.3 CLOS

Nous n'avons pas retenu CLOS pour plusieurs raisons. Tout d'abord nous n'avions pas de hiérarchie complexe d'objets à gérer ni besoin des mécanismes d'héritage. De plus, nous voulions être le plus performant possible et donc éviter le surcroît de traitements (probablement assez faible) introduit par la couche objet CLOS.

Un autre point important est que notre bibliothèque de bas niveau est destinée à être utilisée dans des programmes clients pouvant tout à fait utiliser CLOS pour la représentation de plus haut niveau de leurs objets. Un bon exemple illustrant ces propos est OpenInventor, une bibliothèque de haut niveau de gestion d'objets hiérarchiques écrite en C++ utilisant OpenGL écrit en C pour le tracé de bas niveau.

3.2.4 Intégration d'objets complexes

Notre bibliothèque ne dispose pas de primitives pour afficher des objets : ses primitives ne lui permettent d'afficher que des points, des lignes et des triangles. Pour manipuler des objets non triviaux, l'utilisateur de notre bibliothèque peut bien sûr essayer de composer un objet à partir de ces primitives. Cependant, il est plus pratique de faire appel à un modeleur adapté, puis de traduire le fichier obtenu en appels de primitives de la bibliothèque.

Par mesure de commodité, et parce que nous avions besoin d'afficher des objets à des fins de tests voire de démonstration, nous avons décidé de fournir un traducteur de fichiers d'objets. Le format retenu est celui de 3D Studio (traduit en ASCII) : les fichiers d'extension ASC sont en effet générés par beaucoup de modeleurs et sont assez répandus. Il est aussi facile de les comprendre et de les traiter : le traducteur étant un objectif relativement secondaire, ces arguments ont suffi à nous décider en faveur de ce format.

Cette traduction se fait en trois étapes :

1. Lecture du fichier et instanciation de structures de données intermédiaires : les objets nommés, leurs points, leurs faces (références aux points).
2. Traitements additionnels de ces instances : il faut calculer certains paramètres nécessaires à notre bibliothèque.

   Par exemple, nous avons besoin de calculer les normales aux faces pour déterminer leur éclairage, qui dépend aussi des faces voisines : ce traitement ne peut donc se faire qu'après la lecture de l'objet complet.
3. Traduction de ces instances en appels de primitives de notre bibliothèque. En résumé, nous effectuons un parcours des faces de l'objet, chacune se traduisant par trois appels à la primitive `Vertex`.

Pour effectuer la lecture du fichier, plusieurs options ont été envisagées :

- Lire le fichier ligne par ligne en cherchant des mots-clé. Cette solution a été retenue pour sa simplicité et sa rapidité de mise en œuvre : en effet, nous avions déjà réalisé un tel programme en C (à titre personnel) en très peu de temps.

  Par contre, nous avons été amenés à développer des fonctions de manipulation de fichiers et de chaînes de caractères qui nous faisaient défaut.
- Paramétrer le Reader Common Lisp pour qu'il soit capable d'instancier les structures de données en lisant le fichier.

  En effet, nous voulions autant que possible employer des mécanismes standard du langage Common Lisp pour produire un code naturel et conforme.

  Comme nous avions du mal à mettre en œuvre cette solution, nous l'avons finalement écartée pour économiser du temps. Cependant, il faut signaler que certaines discussions dans le forum `comp.lang.lisp` ont évoqué l'usage de ces mécanismes pour d'autres structures de données : cette option est donc envisageable.
- Utiliser un outil d'analyse lexicale/syntaxique, à l'instar de Flex et de Bison.

  Nous avons téléchargé et examiné Zebu, dont l'auteur a récemment signalé une mise à jour dans `comp.lang.lisp`. Dans la mesure où la grammaire d'un fichier `asc` est très simple, nous avons préféré écarter cette option. Cependant nous aurons appris que de tels outils existent en Common Lisp et que leur usage semble proche de celui du couple Flex/Bison.

Remarque : usage des macros.

Pour éviter des accès mémoire et les tests qu'impliquent les itérations, nous avons eu l'idée de générer statiquement tous les appels de primitives et de les regrouper dans une fonction portant le nom de l'objet correspondant.

Cette génération se fait bien sûr par une macro, qui prend en paramètre le chemin du fichier à traduire. Celle-ci est typiquement appelée dans un programme de test utilisant notre bibliothèque.

Lors de la compilation du programme, les fonctions vont être générées et intégrées au système qui les utilise pour afficher les objets. Ainsi, à l'exécution du programme de test, cet affichage se fait dans des conditions a priori optimales.

Cette solution à première vue élégante, qui essaie de réitérer les optimisations réalisées pour les primitives de transformation, et qui est parfois employée sous d'autres formes par les programmeurs OpenGL, présente cependant un inconvénient majeur. En effet, lorsque l'objet est assez complexe, la taille du code généré peut se compter en milliers de lignes. Dans ce cas, même si la génération des fonctions se fait assez rapidement, leur compilation et le chargement des fichiers binaires obtenus posent un problème à CMU Common Lisp, qui fait des appels intensifs au ramasse-miettes (par exemple, un programme de test compilé comportant trois de ces fonctions avait une taille de 700ko). Cela n'affecte certes pas la phase d'exécution mais représente quand même une gêne réelle et sensible. Ce problème est peut-être dû à une trop grande taille du code compilé : dans ce cas il serait éventuellement possible de paramétrer CMU Common Lisp pour éviter ce désagrément.

En parcourant l'objet à l'exécution, nous avons réalisé que la vitesse du programme n'était pas affectée de manière visible. Cette solution toute naturelle a donc été retenue : peut-être aurions nous dû commencer par là et effectuer un profilage de code avant d'entamer des optimisations potentiellement inefficaces. Il semble que la section critique dans le processus d'affichage concerne plutôt la manipulation de tampons d'image, voire les transformations 3D : dans ce cas il est possible qu'optimiser l'insertion d'objets dans le système n'augmente pas la vitesse globale du programme.

Chapitre 4

Les Techniques d'optimisation

Glos est une bibliothèque graphique temps réel et doit avoir un bon niveau de performance. Pour atteindre ce but, nous nous sommes intéressés à plusieurs techniques d'optimisation. Nous avons déjà traité d'optimisations algorithmiques de portée générale, nous allons maintenant envisager les optimisations possibles avec CMU Common Lisp et Python, son compilateur associé.

Il ne s'agit pas de parsemer le code d'instructions spéciales servant à améliorer sa vitesse globale d'exécution, et ainsi de le défigurer complètement au point de le rendre peu lisible et maintenable. Il s'agit avant tout de savoir quelles sections de code accaparent le plus de ressources temporelles ou spatiales. En effet, ce sont elles qui vont déterminer la vitesse finale d'exécution comme la quantité totale de mémoire allouée.

Nous avons donc employé des outils de profilage pour déterminer où il est nécessaire d'employer les techniques que nous allons décrire. D'ailleurs, si nous n'avons pas recouru à ces méthodes avant de décider d'optimisations algorithmiques dans le module des matrices, c'est que nous étions certains, par connaissance d'autres bibliothèques graphiques, qu'il constituerait un point critique de notre code. Les tests effectués a posteriori ont confirmé cette assertion.

4.1 Choix relatifs aux structures de données

Notre bibliothèque utilise un certain nombre de structures de données, par exemple pour stocker des vecteurs, des matrices 4x4, les coordonnées des objets et des informations relatives aux lumières de la scène. Le choix des types de données a été influencé par les critères de :

- performance des opérations sur ces types.

- minimisation de l'allocation de mémoire (consing).

Ces deux critères sont déterminés par le mode de représentation des types par CMU Common Lisp : avec ou sans descripteur. La représentation avec descripteur utilise un pointeur vers l'objet et d'autres informations sur son type et alloue de la mémoire sur le tas (qui doit être ultérieurement libérée par le ramasse-miettes) alors que la représentation sans descripteur représente directement l'objet et ne génére pas d'allocations. Nous avons donc privilégié les types de base ayant une version non descripteur comme `single-float`, `(unsigned-byte 32)`.

L'usage des listes a été proscrit pour leur lenteur dans le code de l'API Glos et nous avons plutôt utilisé des structures et des tableaux bien plus efficaces. Le manuel d'utilisation CMU Common Lisp contient beaucoup d'informations sur l'optimisation des types et nous a permis de faire les bons choix.

4.1.1 Les types numériques simples : entiers et flottants

Glos utilise les entiers (principalement pour les algorithmes de remplissage de triangle) et les flottants (pour les transformations, l'éclairage,...). Common Lisp possède des types numériques de haut niveau comme `bignum`, `integer` ou `ratio`. Les opérations sur les nombres comme `+` sont génériques : `+` peut prendre des arguments de n'importe quel type numérique comme par exemple un `single-float` et un `bignum`.

Cette généricité a un gros avantage de simplicité pour l'utilisateur mais ralentit dans des proportions non négligeables les programmes comme Glos faisant beaucoup de calculs. Heureusement CMU Common Lisp est capable d'utiliser des versions optimisées d'opérations arithmétiques s'il sait que ses arguments sont d'un certain type. C'est le cas pour les entiers avec les sous-types de `(unsigned-byte 32)` et les flottants `single-float` et `double-float` qui ont tous une représentation machine directe (contrairement aux `bignum` par exemple).

Nous avons dû choisir le type de flottants à utiliser pour tout le code et nous y tenir car il est en effet peu souhaitable de mélanger des calculs comportant des flottants en simple et double précision. L'élément déterminant qui nous a conduit à choisir les flottants simple précision `single-float` est le fait que MesaGL utilise ce type en interne pour tous ses calculs, justifiant ainsi que la précision offerte est suffisante.

D'autre part les flottants simple précision ont l'avantage de consommer moins de mémoire que les doubles. En ce qui concerne la rapidité de traitement elle est certainement équivalente puisque CMU Common Lisp utilise le coprocesseur mathématique (FPU) pour les calculs sur les flottants.

4.1.2 Les tableaux et structures spécialisés

Common Lisp est capable de gérer des tableaux de taille variable et dont les éléments sont de type quelconque. Clairement, les éléments de tels tableaux utilisent une représentation descripteur. C'est pourquoi, tous les tableaux contenant des données auxquelles nous accédons fréquemment sont du type `simple-array` et spécialisés, c'est à dire de taille fixe et contenant des éléments de même type (grâce à l'option `:element-type` de `make-array`) ayant une représentation non descripteur. De même pour les structures dont tous les champs (slots) sont spécialisés (le type de chaque champ est déclaré) pour un accès plus rapide.

4.2 Typage des variables et inférence de types

Par défaut, Common Lisp est un langage effectuant un typage dynamique : le type des variables est déterminé à l'exécution et n'est pas connu à la compilation. Bien qu'ayant des avantages pour certaines applications, le typage dynamique peut faire chuter les performances pour des programmes complexes. La solution à ce problème est de typer explicitement les variables avec la directive `(declare (type ...))`. C'est pourquoi nous utilisons systématiquement des déclarations de type pour toutes les variables utilisées, globales et locales aux fonctions. Lorsque le type des variables est connu dans une expression, le mécanisme d'inférence de type de Python sait déterminer le type de cette expression et optimiser en conséquence. Toutefois l'inférence privilégie toujours le résultat correct de l'expression faisant obstacle à certaines optimisations. Par exemple :

```
(declare  (type  fixnum x))
(1+ x)
```

l'inférence de type déduit que `(1+ x)` n'est pas forcément de type `fixnum`. En effet, dans le cas où x est le plus grand `fixnum`, il y a débordement. La directive spéciale `the` permet d'indiquer à Python que le type qui doit être pris en compte est celui spécifié par `the` et non celui inféré automatiquement. La deuxième ligne de l'exemple ci-dessus devient :

```
(the fixnum (1+ x)).
```

4.3 Options de compilation

Pour chaque fichier comportant du code intervenant lors de l'utilisation de la bibliothèque nous avons utilisé la commande `declaim` :

```
(declaim (optimize (speed 3)
                   (safety 0)
                   (compilation-speed 0)
                   (debug 0)))
```

Cela permet lors de la compilation d'optimiser le code généré pour la vitesse d'exécution `(speed 3)`, sans tenir compte de la vitesse de compilation `(compilation-speed 0)`, de la vérification à l'exécution de la validité des types et des débordements de tableau `(safety 0)` et des d'informations destinées au débogueur `(debug 0)`. Pendant le développement, l'utilisation de `debug` avec le valeur maximale `3` et de `safety` avec la valeur `1` a permis de déboguer et vérifier la validité du code.

4.4 Notes d'optimisation du compilateur

En mode agressif d'optimisation `(speed 3)`, le compilateur Python génére des notes d'optimisation dans le cas où il est incapable d'optimiser certaines expressions. Chaque note indique le coût (en temps d'exécution) de la non résolution de la note. Il est important lorsque la fonction le justifie (par exemple appelée très souvent) de résoudre les notes de coût élevé. Il n'est bien sûr pas utile de résoudre les notes issues du code des macros assurant la macroexpansion et de fonctions peu souvent invoquées ou dont la rapidité n'a pas un intérêt majeur.

Les notes les plus fréquentes sont celle de l'inférence de type de Python qui concernent des expressions dont le type est incertain. Ces notes peuvent souvent être résolues en donnant plus d'informations sur les types avec l'utilisation de `declare` et `the`.

4.5 L'allocation dynamique de la mémoire

4.5.1 Le problème

Notre bibliothèque temps réel ne doit souffrir d'aucun ralentissement dû au ramasse-miettes. En effet, le ramasse-miettes de CMU Common Lisp provoque une pause lorsqu'il se déclenche pour récupérer la mémoire inutilisée (garbage). Ce n'est pas très gênant pour une certaine classe de programmes, mais en ce qui concerne Glos cela se traduit par une pause sensible à intervalles réguliers dans l'animation de la scène 3D. Nous avons cherché à limiter au maximum l'intervention du ramasse-miettes et à avoir le plus grand contrôle possible sur les allocations dynamiques de mémoire malgré le fait que Lisp gère automatiquement les allocations.

Pour ce faire, il a fallu isoler les fonctions appelées souvent et générant beaucoup d'allocation de mémoire à l'aide du profileur intégré à CMU Common Lisp. La fonction `profile` calcule le temps d'exécution de chaque fonction avec le nombre d'appels de la fonction, mais également la mémoire allouée, c'est à dire la mémoire perdue et récupérée par le ramasse-miettes. Il n'a pas toujours été facile de trouver et supprimer les allocations de mémoire car sa gestion étant entièrement automatique, son fonctionnement est parfois obscur. Toutefois, cette étape a été absolument nécessaire dans notre quête de la performance. Par la suite, nous décrivons certaines techniques qui nous ont permis de réduire considérablement les allocations dynamiques de mémoire.

4.5.2 Les solutions

4.5.2.1 Utilisation de réservoirs (pools) préalloués de structures de données

Dans le cas où une fonction nécessite un grand nombre de variables de même type pour ses calculs, où la taille mémoire de ce type est suffisamment grande et donc responsable d'allocations de mémoire, il est alors envisageable d'utiliser un tableau préalloué, dont les éléments sont tous du type désiré (réservoir ou pool). C'est le cas par exemple des fonctions de transformations qui font appel à de nombreux vecteurs et matrices 4×4 temporaires. Le réservoir est un tableau de taille fixe possédant un pointeur de remplissage (fill pointer) pointant initialement sur le dernier élément du réservoir. Cet élément peut être accédé en stockant le résultat de la fonction `vector-pop` dans une variable. Toute modification de la variable modifie l'élément du réservoir associé. Cela agit en réalité comme un pointeur vers cet élément. Comme celui-ci est préalloué, il n'y a pas d'allocation de mémoire. La fonction `vector-pop` décrémente le pointeur de remplissage du réservoir et peut donc être appelée autant de fois que nécessaire pour des variables locales de la fonction. C'est pour cela que le réservoir doit avoir une taille suffisante selon les besoins de la fonction qui l'utilise. Lorsqu'une fonction n'a plus besoin de la variable locale, elle doit appeler `vector-push` pour incrémenter le pointeur de remplissage et rendre l'emplacement accessible à d'autres fonctions. Le fichier `Transforms.lisp` illustre bien l'utilisation de réservoirs.

Remarque : cette technique est décrite en détail dans le livre de Paul Graham [4], dans la section relative aux optimisations.

4.5.2.2 Utilisation de la fermeture d'une fonction

Nous utilisons une fermeture pour ne pas allouer une variable à chaque appel de fonction. Prenons l'exemple d'une fonction utilisant un vecteur (tableau) temporaire pour ses calculs. La définition de la fonction est placée dans le corps d'un `let`, et le vecteur dans la liste des variables du `let`. Ainsi le vecteur sera dans la fermeture de la fonction et ne sera pas alloué dynamiquement à chaque appel. Voici un exemple concret issu du code de Glos relatif à l'éclairage :

```
(let ((sum (vec3-make-null)))
  (declare (type vec3 sum))
  (defun do-lighting ()
    (...utilisation de sum dans le code...))
```

4.5.2.3 La résolution des notes liées à l'allocation

Nous avons résolu certaines notes générées par le compilateur liées à l'allocation de mémoire. Par exemple le message `''Note : Doing float to pointer coercion (cost 13)''` indique le passage à l'exécution de la représentation non descripteur d'un flottant en représentation descripteur (avec pointeur), et donc une source d'allocation de mémoire. Cela se produit principalement lorsqu'une fonction renvoie une valeur de type flottant (dans ce cas c'est normal et il n'y a rien à faire) et dans certains cas de fonctions prenant en paramètre des flottants et pour lequel nous n'avons pas trouvé d'explication logique. Ce genre de problème est délicat à résoudre simplement car le manuel de CMU Common Lisp est vague à ce sujet. Nous avons quand même trouvé des méthodes qui permettent souvent (mais pas toujours) de supprimer ces problèmes en utilisant les macros et les fonctions en ligne (inline).

Chapitre 5

Bilan

5.1 Evaluation des performances

5.1.1 Mode opératoire

Nous avons voulu avoir une idée précise des performances de Glos face à MesaGL en gardant à l'esprit que MesaGL doit respecter la spécification OpenGL complète et de ce fait à plus de travail à effectuer en interne. De plus, MesaGL effectue automatiquement le découpage des triangles (clipping) par rapport au volume de vue (frustum) alors que Glos ne le fais pas.

Le programme de test utilisant Glos est `DemoASC` animant des objets au format ASC (2 révolutions complètes de l'objet autour de l'axe Y).Le programme de test utilisant Glos est `DemoASC` animant des objets au format ASC (2 révolutions complètes de l'objet autour de l'axe Y). Le programme C équivalent, utilisant MesaGL, a été adapté d'un programme existant écrit par l'un des membres du groupe à titre personnel. Ce programme effectue la même animation que `DemoASC`.

Le programme C équivalent utilisant MesaGL a été adapté pour obtenir la même animation que `DemoASC` d'un programme existant écrit par l'un des membre du groupe à titre personnel.

Afin d'éviter que les temps de récupération de la mémoire du ramasse-miettes interviennent dans le temps d'exécution du programme Lisp, la limite de la taille mémoire à partir de laquelle il se déclenche automatiquement a été augmentée (de 2Mo à 6Mo). De plus, une instruction d'invocation du ramasse-miettes a été ajoutée juste avant le premier affichage de la scène, pour qu'il n'intervienne pas dans la suite de l'animation.

Afin de dégager un maximum d'informations dans des conditions d'utilisation différentes, nous avons conduit 2 types de tests en faisant varier :

- les objets affichés pour faire varier le nombre de faces affichées (table 5.1).
- les paramètres d'affichage : avec ou sans éclairage, avec ou sans zbuffer (table 5.2).

Un dernier test indique les fonctions prenant le plus de temps d'exécution processeur (table 5.3).

5.1.2 Résultats des tests

3 lumières, zbuffer activé, gouraud

	Faces	Glos	MesaGL	MesaGL/Glos
Dolphin.asc	563	44.23 fps	166.43 fps	3.76
Ludwig.asc	4994	8.76 fps	30.72 fps	3.50
teapot.asc	1050	29.61 fps	110.61 fps	3.86
box.asc	12	112.85 fps	536.56 fps	4.75

TAB. 5.1: Objets différents.

Dolphin.asc : gouraud

	Glos	MesaGL	MesaGL/Glos
Light, zbuffer	44.23 fps	166.43 fps	3.76
sans Light, zbuffer	62.39 fps	201.96 fps	3.23
Light, sans zbuffer	53.13 fps	215.26 fps	4.05
sans Light, sans zbuffer	81.08 fps	282.62 fps	3.48

TAB. 5.2: Paramètres d'affichage différents.

Dolphin.asc, 3 lumières, zbuffer activé, gouraud

Pourcentages	Secondes	Alloués	Appels	Sec/Appels	Fonction
30.288717	5.130	0	10,800	0.00048	DO-LIGHTING
18.185038	3.080	598,016	720	0.00428	PUT-IMAGE
17.358446	2.940	0	720	0.00408	CLEAR[1]
14.524414	2.460	0	168,508	0.00001	FILLPOLY3ZGOUR
11.218043	1.900	0	10,800	0.00018	DO-TRANSFORMS
8.425342	1.427	1,679,224	720	0.00198	REDRAW
100	16.937	2,277,240	192,268		Total

TAB. 5.3: Fonctions consommant le plus de temps d'exécution.

[1] en réalité : CLEAR-COLOR-AND-DEPTH-BUFFERS

5.1.3 Discussion sur les résultats

Ces tests sont très intéressants et les résultats de la table 5.3 révèlent qu'environ 90% du temps d'exécution est utilisé par les 5 fonctions suivantes :

1. `do-lighting` : effectue les calculs liés à l'éclairage de la scène.
2. `put-image` : envoie le contenu du buffer image (tableau à deux dimensions de la taille de la fenêtre d'affichage) contenant l'image de la scène envoyée au serveur X pour l'affichage.
3. `clear` : efface le buffer image et le zbuffer (tous deux de même taille).
4. `do-transforms` : effectue les transformations pour tous les vertices.
5. `redraw` : le reste des fonctions dans la boucle d'affichage.

L'examen de la table 5.3 indique que :

- les calculs relatifs à l'éclairage (`do-lighting`) occupent près d'un tiers du temps machine, ce qui est considérable. Cela peu probablement être réduit à 20% en optimisant l'algorithme et éventuellement la fonction mathématique exponentielle `expt` utilisée dans `do-lighting`, gourmande en temps d'exécution.
- environ 17% du temps est utilisé pour envoyer le buffer contenant l'image au serveur X grâce à la fonction CLX `put-image`. Plusieurs options sont envisageables pour essayer de diminuer ce pourcentage :
  - Modifier le code source de CLX pour voir s'il est possible d'optimiser ou réécrire `put-image`.
  - Modifier le code source de CLX pour qu'il supporte certaines extensions récentes du serveur X permettant l'accès direct à la mémoire vidéo comme l'extension DGA et XSHM.
  - Ne pas utiliser CLX mais utiliser le mécanisme d'interfaçage de CMU Common Lisp avec C pour accéder aux appels Xlib de la bibliothèque C correspondante. Cette solution est peu satisfaisante car elle nous prive des avantages de CLX et est probablement très difficile à implémenter.
- la mise à zéro des buffers image et zbuffer prends 18% du temps ce qui est relativement lent. Cela provient du fait que CMU Common Lisp ne semble pas posséder de fonction de remplissage rapide d'un tableau avec une valeur quelconque (de type `memset` en C) : nous sommes obligés d'effectuer un parcours classique de tableau à 2 dimensions pour la mise à zéro de chaque élément. La solution la plus appropriée serait de rajouter aux sources de CMU Common Lisp une fonction de type `memset`. A défaut, une autre solution envisageable serait de stocker ces deux buffers dans des tableaux C externes (alien, en terminologie CMU Common Lisp) pour bénéficier de la fonction `memset` de C, et accessible à Glos en Lisp.

- la routine de remplissage de triangle `fillpoly3zgour` (14%) utilise un algorithme bien optimisé et il est peu probable qu'il soit possible de faire beaucoup mieux.

Un rapide calcul sur la table 5.3 indique que le ramasse-miettes se déclenche toutes les 630 images affichées environ, ce qui est très acceptable. Toutefois cela pourrait être encore amélioré.

La table 5.1 indique que la taille des objets influe peu sur le rapport des performances qui reste constant (sauf pour `box.asc` qui est un objet très petit où MesaGL semble bien mieux faire)

De plus, La table 5.2 montre qu'en effectuant toutes les combinaisons des paramètres d'affichage éclairage et zbuffer, Glos est entre 3 et 4 fois plus lent que MesaGL en moyenne. Ce résultat est en accord avec ceux de la table 5.1

5.2 Avantages et inconvénients de Common Lisp avec Glos

5.2.1 Avantages

5.2.1.1 Gestion automatique de la mémoire

Même dans le cadre des optimisations effectuées, toute fuite de mémoire est impossible avec le ramasse-miettes. Cela ne nous empêche pas de gérer explicitement certaines allocations de mémoire pour éviter qu'elles ne se fassent de manière parasite (réservoir). Cela confère à notre bibliothèque un niveau de sécurité appréciable.

5.2.1.2 Un environnement dynamique

La boucle d'interaction de l'environnement Common Lisp permet d'interrompre l'exécution d'un programme utilisant Glos, pour changer certains paramètres d'affichage de la scène (désactiver certaines lumières ou le zbuffer par exemple), puis de reprendre l'exécution exactement à l'endroit ou l'on s'était arrêté.

Cet aspect dynamique rend très commode l'utilisation de bibliothèques en Common Lisp, et il contribue aussi à la sécurité des programmes. En effet, lorsqu'une erreur survient à l'exécution, il est possible de la corriger immédiatement (déboguage, " backtracing ") et de poursuivre l'exécution du programme sans perdre l'environnement courant. Dans la plupart des autres langages, il est hélas nécessaire de relancer le programme depuis le début, et donc il est moins naturel de récupérer son environnement d'exécution.

5.2.1.3 L'utilisation des macros

L'utilisation des macros permet de :

- factoriser le code pour une meilleure lisibilité et concision.
- dérouler les itérations de certaines fonctions critiques.
- vérifier plus tôt les arguments de certaines primitives : c'est le cas du bloc `Begin` dont la vérification de la syntaxe du corps intervient à la compilation et non à l'exécution comme c'est le cas avec MesaGL. En C, il est à la charge du programmeur de récupérer explicitement une erreur de syntaxe du bloc `Begin` avec `glGetError`. D'autre exemples de vérification à la compilation des arguments sont `Enable` et `Clear`.
- d'utiliser toute la puissance de Common Lisp pour générer du code Lisp.

L'usage des macros ne doit pas être systématique pour factoriser le code. Notre opinion est qu'il faut parfaitement maîtriser la sémantique du code pour être capable d'exprimer les idiomes du domaine. Cela est vraiment très difficile quand on traduit toute une série d'algorithmes très complexes et résultant de multiples améliorations et corrections.

5.2.2 Inconvénients

5.2.2.1 Gestion automatique de la mémoire

Eviter des allocations de mémoire indésirables avec Python n'a pas été chose facile (cette remarque tient seulement au ramasse-miettes de CMU Common Lisp).

On aurait apprécié que la documentation de CMU Common Lisp soit plus prolixe et pédagogue sur la manière dont Python gère les descripteurs de types de données. En effet, nous avons dû forcer certains types à adopter une représentation sans descripteur, en utilisant les techniques décrites dans cette documentation. Cependant, dans de nombreux cas particuliers, cette directive était ignorée car le compilateur réclamait en fait des garanties supplémentaires pour opérer. Le problème est soulevé dans la documentation, mais ses causes précises et les méthodes pour le résoudre font cruellement défaut.

Une bonne compréhension générale de ces mécanismes est nécessaire pour obtenir un bon niveau d'optimisation. C'est d'autant plus dommage que Python, pourtant issu du domaine public, est l'un des compilateurs Common Lisp les plus efficaces pour optimiser le code généré.

5.2.2.2 Un typage lourd à gérer

Toujours dans le chapitre des optimisations de compilation, forcer une représentation sans descripteurs pour supprimer l'incertitude de types a l'exécution alourdit sensiblement le code. En effet, il ne suffit pas de déclarer le type des arguments d'entrée-sortie, il faut aussi déclarer le type de retour de chaque opération arithmétique.

Le problème est que l'on est souvent obligé de gérer un typage explicite si on veut éviter le problème d'allocation. Donc, dans les sections critiques, on est obligé de mener de front ces deux optimisations.

5.2.2.3 Certaines primitives font défaut

Il nous manque une fonction `memset`, permettant d'initialiser un grand tableau en un temps très court. De ce fait, on perd énormément de temps à effectuer par nous-mêmes cette initialisation.

C'est d'autant plus dommage qu'il serait probablement très facile d'implémenter cette fonction au niveau de CMU Common Lisp (l'usage de fonctions étrangères ne devrait pas être envisagé pour quelque chose d'aussi simple).

5.2.3 Synthèse

Ensuite, dans le cas général de développement de programmes, les besoins en performances brutes ne sont pas aussi forts que ceux de notre bibliothèque. Nous pensons donc que dans de nombreux cas, on peut utiliser toute la richesse du langage (CLOS, MOP) sans avoir vraiment besoin de considérer des optimisations très poussées. D'ailleurs, la plupart de ces optimisations consistent à renoncer localement aux commodités du langage et à prendre en charge certains mécanismes que d'autres langages imposent systématiquement.

Outre l'application des optimisations suggérées, on peut bien entendu s'inspirer d'OpenGL pour trouver une multitude de fonctionnalités à ajouter à notre bibliothèque. En réalité, le mieux que l'on puisse souhaiter dans l'immédiat, c'est que les sources de notre bibliothèque expérimentale puissent être consultées par des personnes intéressées par le projet et pouvant l'améliorer ou à des fins éducatives.

Remerciements

Nous voudrions remercier tous les personnes qui ont eu la bienveillance de nous transmettre des versions récentes de leurs réalisations ou des documentations.

Nous voudrions spécialement remercier les participants de `comp.lang.lisp` pour l'attention qu'ils ont porté à nos questions sur Common Lisp, Python, voire aux choix de notre projet en général. Parmi tant d'autres, nous tenons à citer Messieurs Luca Pisati, Tunc Simsek, Pierre R. Mai et Raymond Toy.

Bibliographie

[1] *The Official Reference Document to OpenGL.*

[2] David B.Lamkins. *Successful Lisp : How to Understand and Use Common Lisp.* `http ://psg.com/~dlamkins/left/sl/sl.html`.

[3] Foley, Van Dam, Feiner, Hughes, and Phillips. *Introduction à l'infographie.* Addison-Wesley.

[4] Paul Graham. *ANSI Common Lisp.* Prentice Hall, 1996.

[5] Paul Graham. *On Lisp Advanced Techniques for Common Lisp.* Prentice Hall, 1996.

[6] Renate Kempf and Chris Frazier, editors. *OpenGL reference manual : the official reference document to OpenGL*, volume 2 of *OpenGL Architecture Review Board.* Addison-Wesley, 1996.

[7] Robert A. MacLachlan. *CMU Common Lisp User's Manual.* Number CMU-CS-92-161. 1992.

[8] Ken Pitman. *The Common Lisp Hyperspec.* Harlequin group ltd, :http ://www.harlequin.com/education/books/HyperSpec/FrontMatter/index.html, 1996.

[9] Robert W. Scheiffer and Lamett Oren. *CLX Common LISP X Interface.* 1988-1989.

[10] Mark Segal and Kurt Akeley. *The OpenGL Graphics System : A Specification.* Silicon Graphics, ftp ://sgi-gate.sgi.com/pub/opengl/doc/opengl1.2/opengl1.2.1.ps, 1992-1999.

[11] Guy Steele. *Common Lisp, the language.*

[12] Signum Support, editor. *Version Management with CVS.* 1992-1993.

[13] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL programming guide : the official guide to learning OpenGL.* Number ISBN 0-201-46138-2. 2nd edition.

[14] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*, volume 2. Addison-Wesley, 1997.