Découverte de la
Programmation AmigaOS

Version 1.0 - 07/11/2003

http://www.guru-meditation.net

Avant-propos

Bonjour à tous,

A l'occasion de l'Alchimie, nous sommes très heureux de faire voir le jour à ce document en français sur la programmation Amiga. Il devrait représenter une précieuse source d'informations et il sera complété et corrigé au fil du temps.

Le but de ce livret est de vous offrir des clés, des pistes pour partir sur les chemins du développement en les balisant. Par exemple, vous ne trouverez pas ici de tutoriels sur l'utilisation des bibliothèques ou des devices du système. Nous donnons ici des principes et des conseils mais pas de code : pour des sources et des exemples, nous vous renvoyons à notre site (**http://www.guru-meditation.net**) et un chapitre est en plus réservé à la recherche d'informations.

Nous essaierons de prendre en compte un maximum de configurations possible, de signaler par exemple les spécificités de MorphOS, ... Quelque soit le système, on peut d'ors et déjà déconseiller à tous de "coder comme à la belle époque" comme on entend parfois, c'est à dire en outrepassant le système.

Nous souhaitons, par le développement, contribuer à un avenir plus serein de l'Amiga. C'est pourquoi, parfois avec un pincement, nous omettrons de parler d'outils ou de pratiques "révolus". On conseillera avant tout ceux qui sont maintenus et modernes ... ou encore, à défaut, anciens mais indispensables. Objectif : futur.

Ce livret est très porté vers le langage C mais donne malgré tout de nombreux éclairages sur la programmation en général. Le développement est une activité exigeante mais passionnante et chaque résultat, chaque progrès apporte un réel plaisir.

Il y a toujours quelque chose à créer : un petit outil DOS, une interface graphique pour un logiciel, un portage d'un programme Linux, ... L'important est de ne pas brûler les étapes. Nous souhaitons que ce livret en soit la première et pourra faciliter votre apprentissage de la programmation, vous amenant ainsi à vous passionner pour le développement sur l'Amiga, quel qu'il soit.

L'équipe guru-meditation

PS : Pour toute suggestion ou correction, veuillez nous contacter par email à l'adresse : corto@guru-meditation.net

Quel langage choisir ?

Quasiment tous les langages existent sous AmigaOS : Cobol, Fortran, Modula, Ada, Eiffel, Perl, ... mais aussi les plus typiques Rebol et ARexx. Nous allons seulement parcourir ici les grandes familles, histoire de débuter la programmation système en choisissant une valeur sûre. Les connaissances acquises suffiront à s'attaquer à des langages plus particuliers si vous le souhaitez.

Nous avons retenu cinq familles, à savoir : C/C++, assembleur, Basic, Java, inclassables. Les langages ont tous leurs particularités (langage objet, difficulté d'apprentissage, disponibilité, efficacité, ...). Après une brève description de chacun, vous devriez pouvoir vous faire un avis sur **le plus adapté à vos besoins et à vos compétences**.

Nous citons à chaque fois des outils avec la mention "pour mémoire". Il s'agit de rappeler leur existence mais bien que pourvus d'avantages, nous vous conseillons de ne plus les utiliser. Ils sont soit indisponibles, soit limités, ... un développeur se doit de choisir et d'utiliser des outils efficaces et pour lesquels il pourra trouver assistance.

Langage C/C++

Le langage C est éprouvé et toujours vaillant. Considéré comme langage de haut niveau, il permet toutefois de "taper dans le métal". La rigueur qu'il demande prépare à n'importe quel autre langage et sa connaissance est indéniablement un plus.
Le C++ contient de riches bibliothèques de base et apporte les facilités de la programmation orientée objets mais ajoute des notions parfois complexes (qu'on peut cependant ne pas utiliser).
Avantages : langages de référence, efficacité, polyvalence
Inconvénients : difficulté d'apprentissage, demande de la rigueur
Conseillé pour : les projets évolutifs et pour des programmeurs ayant des notions de base

SAS/C : Le compilateur mythique de l'Amiga est toujours très utilisé mais n'est plus développé depuis des années. Il est très rapide à la compilation et génère du code de qualité. En revanche, il ne supporte que partiellement le C++ et possède une extension pour la compilation PPC restée à l'état expérimental. Ce produit commercial n'est plus en vente.
URL : non disponible

GCC : LA référence multi-plateforme est très utilisée, même si son installation est redoutée et requiert la mise en place de l'environnement Geek Gadgets. Assez long à la compilation, il reste performant et digère sans problème le C et le C++. Ce compilateur gratuit est fourni avec un tas d'autres outils GNU.
URL : http://www.geekgadgets.org ou http://mdc.morphos.net

VBCC : Cet outsider pourrait bien se faire une bonne place sur votre système. Ce compilateur gratuit s'installe sans soucis et bénéficie d'un bon support. Il présente en plus certains des atouts

de GCC : cross-compilation, profiling, ... Par contre, il ne comprend que le langage C.
URL : http://devnull.owl.de/~frank/vbcc_e.html ou http://www.compilers.de/vbcc

StormC : Le seul compilateur commercial toujours plus ou moins supporté (verra-t-on une version OS4 et/ou MorphOS ?). Il permet la compilation 68k et PPC (WarpOS) et sa dernière version utilise GCC comme compilateur, l'intérêt réside donc dans l'environnement de développement intégré pour gérer ses projets. Il bénéficie aussi d'un débogueur intégré puissant.
URL : http://www.haage-partner.com/storm/sc_e.html

Pour mémoire : DICE, HiSoftC++, MaxonC++, ...

Assembleurs

Langage de bas niveau, l'assembleur qui manipule directement le matériel et les registres du processeur, est toujours utilisé même si des polémiques voudraient qu'on ne lui trouve plus d'utilité. Les maîtres de l'assembleur font figure de héros du code et ce langage a toujours fait fantasmer les programmeurs. Pour l'heure, le problème sur l'Amiga réside dans la migration actuelle des processeurs 68k au PowerPC.
Avantages : maîtrise totale du processeur, performances
Inconvénients : travail fastidieux, connaissances attachées à un seul processeur
Conseillé pour : les intrépides et les gens rodés au C

Devpac 3 : A l'image de SAS/C, l'illustre Devpac de l'éditeur Hisoft et ses outils s'étaient fait une excellente réputation et il comprenait un support et toutes les includes systèmes. Réservé aux codeurs 68k, ce logiciel commercial n'est plus vendu depuis longtemps.
URL : non disponible

PhxAss : Sans doute le meilleur actuellement, l'assembleur de Frank Wille est utilisé par le compilateur VBCC pour sa génération de code 68k.
URL : http://devnull.owl.de/~frank/phxass_e.html

pasm : Développé par Frank Wille (encore lui !), c'est l'équivalent PPC de PhxAss et il sert donc aux versions PowerUP, WarpOS et MorphOS de VBCC. Gratuit et toujours développé, il supporte les instructions standards mais aussi AltiVec.
URL : http://devnull.owl.de/~frank/pasm_e.html

PowerASM : Uniquement disponible pour WarpOS, c'est l'assembleur de l'environnement StormC dans lequel il s'intègre parfaitement. Une excellente documentation (ppctut.lha) lui est consacrée et téléchargeable sur Aminet.
URL : http://www.haage-partner.com/storm/pa_e.html

Barfly : Avec ses outils et son fameux débogueur (soit disant le meilleur), le bébé de Ralph Schmidt n'a sans doute pas le succès qu'il mériterait. Barfly est désormais aussi livré dans le SDK MorphOS.
URL : Aminet et MDC

AsmOne : Il nous revient d'outre-tombe et le développement a repris. Il supporte désormais le PowerPC en plus de la famille 68k.
URL : http://www.euronet.nl/users/jdm/homepage.html

Pour mémoire : a68k, AsmPro (désormais développé en OpenSource), K-Seka, ...

Basic et assimilés

Ses représentants sont principalement destinés à ceux qui veulent obtenir un résultat sans trop attendre, à ceux qui préfèrent réfléchir à l'organisation du programme plutôt qu'aux détails de sa conception. Vous n'aurez pas besoin de vous perdre dans des considérations techniques tortueuses pour créer simplement des applications étonnantes !
Loin d'être péjoratif, le mot Basic représente avant tout une facilité d'apprentissage et une puissance intrinsèque à chaque commande.
Avantages : la puissance et les résultats en quelques commandes
Inconvénients : difficulté d'évolution possible
Conseillé pour : les débutants, les programmeurs occasionnels

Blitz Basic : Il s'appelle aujourd'hui AmiBlitz et évolue régulièrement depuis que les sources ont été donnés. Le nombre de bibliothèques de fonctions additionnelles (TCP/IP, MUI, chunky, ...) lui donne toute sa puissance. Plus respectueux du système, moins bogué, AmiBlitz tient la route ! Il ne compile toujours qu'en 680x0.
URL : http://blitz-2000.david-mcminn.co.uk/home.html

Pure Basic : Nous devons ce successeur du Blitz au français Frédéric Laboureur. Il est en standard bien fourni et intégré au système, il produit du code optimisé et est même multi-plateforme (Amiga, Windows, Linux). Il a manqué quelque chose pour qu'il s'impose et la version Amiga est aujourd'hui en retrait ...
URL : http://www.purebasic.com

PowerD : Un langage à découvrir pour zéro Euro ! Il plaira à tous ceux qui ont aimé le langage E puisqu'il s'en inspire. Il semble assez peu utilisé par rapport à la puissance qu'il offre. Bien qu'assez récent, il propose de nombreux modules tels que MUI, AHI, Warp3D, ...
URL : http://www.tbs-software.com/powerd/index.html

Pour mémoire : Amos et AmosPro

Java

Sa présence dans cette liste peut paraître anecdotique puisqu'il manque des bibliothèques indispensables (AWT et Swing pour le graphisme et les interfaces), mais peut convenir à une étude de ce sympathique langage (notion d'objet, héritage, exceptions, machine virtuelle). Il faudra réunir, sur une machine pourvue de bonnes ressources, le compilateur Jikes d'IBM et la JVM Kaffe avec ses packages de base. Sinon, deux alternatives se présentent.
Avantages : langage multiplateforme, orienté objet et simple à apprendre

Inconvénients : implémentations à l'état expérimental pour l'instant
Conseillé pour : ceux qui veulent goûter à la programmation orientée objets

Jamiga : Ce projet initié par Peter Werno a été démarré dans un but d'apprentissage. Mais cette JVM qui souhaite atteindre la compatibilité avec la version officielle 1.4 est utilisable avec les classes de base. Pour bénéficier de classes supplémentaires sans tout réécrire, Jamiga s'appuiera sur le projet GNU Classpath.
URL : http://sourceforge.net/projects/jamiga

JavaMos : Cette solution n'arrivera peut-être pas de si tôt mais apparemment, quelqu'un de confirmé dans le domaine se serait attelé à l'adaptation ou à l'écriture d'une JVM pour MorphOS.
URL : non disponible

Pour mémoire : Java est encore trop jeune sur Amiga pour avoir de la mémoire !

Inclassables

Pourquoi inclassables ? Parceque cette partie contient des langages à part, et radicalement différents les uns par rapport aux autres. Sur le déclin (langage E), futuriste (Rebol) ou spécifiques, ces langages ont une identité propre et sont à découvrir.
Avantages : langages accessibles aux débutants, résultats rapides
Inconvénients : hyper spécialisation, manque de perennité possible
Conseillé pour : des développements bien ciblés

Langage E : Il n'existe que sur Amiga ! Destiné à écrire de véritables applications (comme Photogenics à ses débuts), il privilégie la programmation système à la manière du C, mais avec une approche moins ardue. Peu implanté en France, vous pouvez l'acquérir gratuitement et Aminet recèle de nombreux exemples et bibliothèques.
URL : Aminet

ARexx : Il ne possède aucun équivalent pour piloter des applications (et désormais le Workbench !) ou les faire communiquer entre elles. Le principe est simple : les fonctions de base internes à une application peuvent être appelées de l'extérieur et ça permet d'automatiser de nombreuses tâches. ARexx est un langage de script, interprété mais peut s'utiliser de manière totalement indépendante, notamment avec son extension MUI. En plus, son manuel est disponible en français (sur le site Boing Attitude) !
URL : Aucune, langage fourni avec l'OS

Rebol : Complètement innovant et multi-plateforme, c'est plus qu'un langage, c'est un environnement communicant ! Il a été conçu en s'affranchissant de tous les concepts académiques, condition essentielle pour franchir un grand pas technologique. On ne sera pas surpris que son créateur, Carl Sassenrath, soit un des pères de l'Amiga.
URL : http://www.rebol.com

Karate : Ce langage interprété utilise un système de balises simples, comme en HTML. Sauf qu'ici, le résultat est plus attrayant puisque Karate est conçu pour créer simplement des démos

comme les plus grands codeurs. Merci Krabob ! On s'écarte des langages de programmation classiques mais l'important est de créer et d'apprendre à ordonner ses idées.
URL : http://www.k-fighter.net

Hollywood : Ce logiciel de présentation multimédia présente lui-aussi un langage permettant de décrire des séquences et des actions à partir de scripts, comme l'illustre Scala dont il sait traiter les fichiers. Il est possible de compiler sa présentation sous forme d'exécutable ! Très spécialisé mais permet d'obtenir du concret rapidement.
URL : http://www.softwarefailure.de/en/prod_hollywood.html

Que vous soyez débutant en programmation ou bien amateur chevronné, ce tour d'horizon vous aura sans doute permis de mieux choisir une langage qui vous est adapté. On a terminé ce tour d'horizon par des outils spécialisés qui n'ouvre pas forcément à la programmation pure mais leur pouvoir créatif peut vous y amener. Ils font d'ailleurs intervenir des concepts importants : boucles, comparaisons, etc. c'est à dire tout ce qui compose un algorithme.

En définitive, le choix du langage vous appartient et doit se faire en fonction :
- des OS cibles : unique et ciblé ou bien aussi nombreux que possible ?
- du but à atteindre : apprentissage, besoin de performances, rapidité de développement, ...
- type de programme : démos (karate), présentations (Hollywood), jeux, calcul, ...

Recherche d’informations

Une fois le choix du langage effectué, les premiers tests peuvent commencer. Il est tout à fait possible de se débrouiller seul avec la documentation fournie et avec beaucoup de volonté. Mais l’évolution sera bien sûr plus rapide si vous savez vous entourer de précieux conseils et d’un maximum d’informations.
Plusieurs pistes existent, qu’elles soient généralistes ou spécialisées dans un langage ou une technologie. Dans un premier temps, le débutant devra se familiariser avec les principes de base, la logique, ... Pour le C et autres langages universels et largement couverts, l’idéal est d’acquérir un livre du commerce pour apprendre le langage, en suivant de nombreux conseils illustrés d’exemples.

Le papier toujours et encore

Les anciens supports, sur papier, peuvent toujours servir. Il fut une époque où l’Amiga avait une presse spécialisée et toute sa place dans les rayons informatiques de librairies.

L’éditeur Micro Applications a été très prolifique, même dans la spécialité programmation en traitant des langages BASIC, assembleur et C. Mais la référence technique en matière d’architecture de l’Amiga et de développement était sans conteste le pavé qu’est la “**Bible de l’Amiga**”. De nos jours, la plupart des informations de cette bible n’est plus pertinente.

Les informations sur les processeurs **Motorola** se trouvent dans des ouvrages d’autres éditeurs. Eyrolles propose toujours 2 titres bien qu’en Anglais. On les retrouve d’ailleurs sur le site Amazon qui propose en plus 3 livres en français. Motorola (http://e-www.motorola.com) lui-même envoie gratuitement des titres sur simple demande.
Au passage, pour retrouver ces titres, j’avais d’abord cherché les mots-clés “motorola 68000” dans les pages francophones de Google et j’ai réalisé qu’il existait beaucoup de tutoriels dédié à cette famille de processeurs.

Le **CManual** a été vendu imprimé en français à une époque. Désormais il est uniquement disponible sur Aminet. Son contenu est ancien mais il présente les différentes parties du système. En cela, il constitue une bonne approche.

Dans le même style et beaucoup plus complet, il est possible de trouver aux enchères sur Internet les fameux RKRM (**ROM Kernel Reference Manuals**) qui ne sont autres que les documentations officielles rédigées à l’époque de Commodore. On les trouve aussi au format AmigaGuide sur Internet ou sur le CD Developer 2.1.

Voilà pour les livres dédiées à l’Amiga. Pour les langages les plus populaires, nous avons sélectionné quelques titres pour leur richesse et leur clarté.
C : “Le langage C” par Brian Kernighan & Dennis Ritchie, éditions Dunod
C : “Méthodologie de la programmation en C” par J-P Braquelaire, éditions Dunod

JAVA : “A coeur de JAVA 2” (2 volumes) par Horstmann & Cornell, Editions Campus Press

La révolution Internet

On a vu que des livres et tutoriels, anciens ou pas, sont présents sur Internet. C’est en effet un outil à exploiter à fond quand on recherche des informations, présentes sur des sites mais pas seulement.

Le site **guru-meditation** est incontournable. Avec ses articles, ses forums, ses liens et toute l’actualité en français, il est complètement dédié à la programmation Amiga.

Pour les programmeurs MorphOS, il existe un site officiel, le MorphOS Developer Connection (**http://mdc.morphos.net**) qui propose lui aussi des forums et articles (en anglais) mais également un accès au SDK et à d’autres fichiers utiles au développement.

Concernant OS4, on imagine qu’il existera un équivalent mais on n’a pas encore d’informations dessus.

Pour trouver les sites qui traitent d’un sujet précis, il y a bien sûr les moteurs de recherche, comme Google. Mais ce dernier est possède un avantage important puisqu’il propose aussi une recherche sur les archives des newsgroups : tant qu’on a pas essayé, on n’imagine pas toutes les informations dispensées puis oubliées.

Enfin, abonnez-vous aux mailing-lists et forums. C’est le meilleur moyen d’entamer un dialogue avec des homologues. En français, il y a la **liste AmigaImpact dédiée au développement** : ml-dev. On s’inscrit via le site guru-meditation.

Le choix s’étoffe quand on s’ouvre à la langue anglaise :

- **amiga-c**, puis plus spécialisées comme **MUI** ou **SDL** sur Yahoogroups (http://groups.yahoo.com) et pour souscrire directement par email : “nomdelaliste-subscribe@yahoogroups.com”
- **AmiBlitz** : souscription par email à “blitz-list-subscribe@netsoc.ucd.ie”
- **comp.sys.amiga.programmer** ou **fr.comp.lang.c** ou bien d’autres sur les newsgroups
- **pure basic** : souscription par email à “mailing-request@purebasic.com” avec “subscribe” comme sujet

Plus on est préparé et formé, plus la diversité des sources est grande, plus la curiosité est aiguisée, plus les progrès seront rapides et concluants.

Includes et stub libs

La principale différence entre le “bête” C ANSI/ISO et le C sur Amiga est l’utilisation de bibliothèques partagées. La quasi intégralité de l’AmigaOS se trouve sous cette forme (exec.library, dos.library, intuition.library...) et il existe également de nombreuses bibliothèques additionnelles créées par des auteurs indépendants.

Le langage C utilise la pile pour passer les paramètres aux fonctions, tandis que ce sont les registres du 68k qui sont utilisés pour les fonctions de l’AmigaOS et des autres bibliothèques. Pour utiliser ces dernières, un programmeur doit donc faire appel à des fichiers d’includes ou à des link-libs particuliers. Ces fichiers ne sont pas les mêmes selon le compilateur que vous utilisez et les auteurs de bibliothèques ne fournissent pas toujours ceux dont vous avez besoin.

Cet article vise à clarifier le sujet des inlines, pragmas, protos, fd, sfd, clib et autres stub-libs utilisés sur Amiga, et à vous expliquer comment les générer s’il vous en manque pour utiliser une bibliothèque particulière. Il abordera également brièvement le sujet de l’amiga.lib.

Le répertoire clib

Ce répertoire contient les prototypes des fonctions des bibliothèques partagées. C’est du C standard : le compilateur doit connaître le prototype d’une fonction avant que celle-ci ne soit appelée dans un source. Le fichier clib d’une bibliothèque doit donc être inclus (indirectement, nous l’expliquons ensuite) dans tout programme souhaitant utiliser cette dernière.

Extrait du fichier clib/amigaguide_protos.h:

```
LONG LockAmigaGuideBase( APTR handle );
VOID UnlockAmigaGuideBase( LONG key );
APTR OpenAmigaGuideA( struct NewAmigaGuide *nag, struct TagItem * );
APTR OpenAmigaGuide( struct NewAmigaGuide *nag, Tag tag1, ... );
```

Les prototypes sont les mêmes quels que soient le compilateur et le processeur/système cible (68k, PPC, x86, AmigaOS, PowerUp, WarpOS, MorphOS ou
Amithlon).

Les pragmas

Les pragmas se trouvent généralement dans un répertoire “pragmas”, ou parfois “pragma” (dans le cas d’Aztec C, et aussi lorsqu’on utilise fd2pragma (voir la fin de l’article)). Ils sont par définition spécifiques à chaque compilateur. Ils sont utilisés pour l’accès aux bibliothèques partagées par les compilateurs SAS/C, StormC 3, DCC, MaxonC, Lattice et Aztec.

Ces fichiers contiennent la description des registres du processeur utilisés pour transmettre les paramètres aux fonctions des bibliothèques, ainsi que la base de bibliothèque à utiliser.

Voici un exemple de pragma dans le format utilisé par SAS/C, DCC et Lattice:

```
#pragma libcall AmigaGuideBase LockAmigaGuideBase 24 801
#pragma libcall AmigaGuideBase UnlockAmigaGuideBase 2a 001
#pragma libcall AmigaGuideBase OpenAmigaGuideA 36 9802
#pragma tagcall AmigaGuideBase OpenAmigaGuide 36 9802
```

Vous n'y comprenez rien ? Moi non plus. Notez que le pragma tagcall n'est valable que pour SAS/C ; les deux autres compilateurs ne peuvent apparemment pas appeler de fonctions à nombre variable d'arguments (aussi appelées "tag functions") par cette méthode.

Voici la description des mêmes fonctions mais pour StormC, Maxon et Aztec C cette fois:

```
#pragma amicall(AmigaGuideBase, 0x24, LockAmigaGuideBase(a0))
#pragma amicall(AmigaGuideBase, 0x2a, UnlockAmigaGuideBase(d0))
#pragma amicall(AmigaGuideBase, 0x36, OpenAmigaGuideA(a0,a1))
#pragma tagcall(AmigaGuideBase, 0x36, OpenAmigaGuide(a0,a1))
```

C'est déjà un petit peu plus compréhensible à l'oeil nu, puisqu'on y remarque la liste des registres utilisés. Comme précédemment, le pragma tagcall ne marche qu'avec StormC et pas avec Maxon ni Aztec C.

Les inlines

Les inlines se trouvent dans le répertoire "inline". Ce sont des macros qui remplissent exactement la même fonction que les pragmas mentionnés ci-dessus, mais pour les compilateurs GCC et VBCC (ainsi que StormC 4 puisqu'il utilise GCC).

Exemple pour GCC 68k :

```
#define LockAmigaGuideBase(handle) \
  LP1(0x24, LONG, LockAmigaGuideBase, APTR, handle, a0, \
  , AMIGAGUIDE_BASE_NAME)
#define UnlockAmigaGuideBase(key) \
  LP1NR(0x2a, UnlockAmigaGuideBase, long, key, d0, \
  , AMIGAGUIDE_BASE_NAME)
#define OpenAmigaGuideA(nag, attrs) \
  LP2(0x36, APTR, OpenAmigaGuideA, struct NewAmigaGuide *, nag, a0, struct TagItem *, attrs,
a1, \
  , AMIGAGUIDE_BASE_NAME)
#define OpenAmigaGuide(a0, tags...) \
  ({ULONG _tags[] = { tags }; OpenAmigaGuideA((a0), (struct TagItem *)_tags);})
```

C’est plus confus mais tout de même un peu plus compréhensible que les pragmas. Notez que la syntaxe “({ ... })” n’est pas du C standard mais une notation spécifique à GCC qui sert à assigner au bloc la valeur de la dernière expression évaluée en son sein.

Et pour VBCC 68k, ça ressemble à ça:

```
LONG __LockAmigaGuideBase(__reg(”a0”) APTR handle, __reg(”a6”) struct Library *)=”\tjsr\t-36(a6)”;
#define LockAmigaGuideBase(handle) __LockAmigaGuideBase((handle), AmigaGuideBase)

void __UnlockAmigaGuideBase(__reg(”d0”) long key, __reg(”a6”) struct Library *)=”\tjsr\t-42(a6)”;
#define UnlockAmigaGuideBase(key) __UnlockAmigaGuideBase((key), AmigaGuideBase)

APTR __OpenAmigaGuideA(__reg(”a0”) struct NewAmigaGuide * nag, __reg(”a1”) struct TagItem **,
                       __reg(”a6”) struct Library *)=”\tjsr\t-54(a6)”;
#define OpenAmigaGuideA(nag, *) __OpenAmigaGuideA((nag), (*), AmigaGuideBase)
```

L’auteur de VBCC a donc opté pour l’assembleur inline (ce qui n’est pas du C standard non plus). Pour MorphOS (systèmes PowerPC plus globalement), les inlines ont encore une autre allure et celles de GCC sont placées dans le répertoire distinct “ppcinline”.

Ici aussi, OpenAmigaGuide(), qui a un nombre variable d’arguments, n’est pas présente. Mais cela ne veut pas dire qu’on ne peut pas l’utiliser... Il existe en effet, pour tous les compilateurs, la possibilité d’utiliser des stub libs.

Les stub libs

Les stub libs sont des bibliothèques statiques (ou link libraries) remplissant la même tâche que les fichiers pragmas et inlines. Elles sont utilisables par tous les compilateurs, mais chacun a bien sûr un format de libs différent (sinon, ça ne serait pas drôle).

Elles contiennent des stubs, qui sont des fonctions C standard se chargeant de prendre les paramètres sur la pile pour les placer dans les registres, avant d’appeler la fonction correspondante de la bibliothèque.

Vous avez généralement le choix entre utiliser des inlines ou pragmas et utiliser une stub lib.

Une chose à noter est que pour les compilateurs dont les inlines/pragmas ne gèrent pas les fonctions à nombre d’arguments variable, vous devez utiliser une stub-lib pour pouvoir appeler ces fonctions. Vous n’aurez normalement pas de souci avec les fonctions de l’OS car des stubs devraient être inclus dans l’amiga.lib de votre
compilateur, et donc linkées à vos exécutables par défaut.

Pour utiliser une stub lib, procédez de la même façon que pour n’importe-quelle link-lib: en rajoutant “-lnom_de_la_lib” à la ligne de commande, ou en rentrant son nom à l’endroit adéquat si

vous utilisez une GUI comme celle de StormC.
Attention, si le fichier s'appelle par exemple "socket.lib", il faut taper "-lsocket" sans le ".lib". GCC ainsi que VBCC PowerUP/MorphOS utilisent la notation "libsocket.a", mais ça ne change rien: il faut également taper "-lsocket".

Le répertoire proto

Dans ce répertoire, des fichiers comme par exemple "proto/amigaguide.h" qui incluent eux-mêmes:
- le fichier de prototypes du répertoire clib
- le fichier d'inlines ou de pragmas qui correspond au compilateur utilisé (ou rien si une stub lib doit être utilisée).

Vous ne devriez jamais inclure directement les pragmas ou inlines dans vos sources. Ces fichiers sont trop dépendants du compilateur. De même, à l'exception de certains fichiers (comme clib/alib_protos.h qui contient les prototypes de l'amiga.lib), vous ne devriez jamais y inclure directement les fichiers clib.

Utilisez les fichiers du répertoire "proto" au lieu d'inclure explicitement les fichiers clib/pragmas/inlines de votre compilateur; de cette façon, vos sources seront compilables avec tous les compilateurs sans aucune modification. Et en plus, ça évite de surcharger le source.

Par exemple, ne tapez PAS ça:

```
#include <clib/amigaguide_protos.h>   // c'est MAL
#include <inline/amigaguide_protos.h> // c'est MAAAL
```

Mais au contraire tapez ça:

```
#include <proto/amigaguide.h>
```

Notez que les protos fournis dans les includes officielles de l'AmigaOS (dans le NDK 3.9 et sur les CD Dev) n'incluent pas les clib, pour une raison qui m'échappe. Mais peu importe : ils ne sont jamais utilisés ; votre compilateur est normalement livré avec ses propres protos pour toutes les bibliothèques de l'OS.

L'amiga.lib

L'amiga.lib est une link lib à double emploi. Elle contient d'une part des fonctions utilitaires comme DoMethod(), HookEntry() ou RangeRand(), et d'autre part des stubs pour toutes les fonctions de l'OS, voire même celles de bibliothèques externes selon votre compilateur.
C'est pourquoi même s'il vous manque le pragma ou l'inline pour une fonction donnée de l'OS, vous pourrez quand même l'utiliser grâce au stub présent dans l'amiga.lib (en général, cette lib est linkée par défaut dans tout exécutable par votre compilateur).

Pour utiliser des fonctions de l'amiga.lib dans un programme, vous devez inclure son clib:
#include <clib/alib_protos.h>

Ceci n'est pas nécessaire si vous n'avez besoin que des stubs et pas des fonctions utilitaires.

FD, SFD et fd2pragma

Maintenant que vous avez vu la composition de tous ces fichiers, vous devez vous dire que ça va prendre du temps pour tous les taper à la main ;-). Heureusement, il existe des outils pour les générer automatiquement :-).
Ces outils utilisent généralement les fichiers FD comme point de départ. Les fichiers SFD sont apparus au public assez récemment dans le NDK3.9 et permettent eux-mêmes de générer des FD, clib et autres fichiers.

Voici un petit bout d'un fichier FD:

```
LockAmigaGuideBase(handle)(a0)
UnlockAmigaGuideBase(key)(d0)
##private
amigaguidePrivate2()()
##public
OpenAmigaGuideA(nag,*)(a0/a1)
```

Ce fichier contient les informations nécessaires pour que le compilateur sache comment appeler une fonction: son adresse par rapport à la base de la
bibliothèque et les registres utilisés.

Il existe notamment fd2inline pour générer des inlines pour GCC à partir des FD, mais le plus puissant (et le seul que j'utilise) s'appelle fd2pragma et est disponible sur Aminet. Contrairement à ce que son nom pourrait laisser croire, il permet non seulement de générer des pragmas à partir de FD, mais également le contraire, ainsi que des inlines, stub libs, etc. ; et ce pour tous les compilateurs et systèmes (AOS 68k, WOS, PUP, MOS - désolé, il manque Amithlon).

Débogage

Ce riche sujet mérite un chapitre fourni. Bien que le but soit toujours le même, fournir une application saine, le débogage s'envisage de nombreuses manières. Depuis des actions élémentaires jusqu'à l'utilisation d'outils spécialisés, il y a toute une panoplie de solutions pour créer un logiciel performant et sûr. C'est une étape délicate et qui demande de réels efforts pour trouver et retirer les erreurs de programmation (fonctionnement anormal, mauvaise gestion de la mémoire, ...). Mais la satisfaction d'obtenir une application propre est si agréable.

Prévenir les bugs

Ca n'est pas nouveau, le préventif est préférable au correctif. Et il existe des solutions pour trouver des bogues avant même la première exécution du programme, c'est à dire par étude du source.

Tout d'abord par la manière la plus naturelle qui soit : la **relecture** par des yeux bien humains. Et c'est encore mieux si elle est réalisée par une personne tierce qui pourra porter un regard différent. Encore une bonne raison pour appliquer quelques règles simples : rendre le code lisible, utiliser des noms de variables explicites, définir des constantes quand c'est possible, indenter correctement (notamment à l'aide de l'outil indent), ... Un source confus devient vite un bouillon de culture de bogues.
On n'y pense que rarement, ou on n'y tient pas, mais la lecture de son code par quelqu'un d'autre est des plus bénéfique ! Cela permet de revenir sur l'architecture, la lisibilité, les techniques employées, ... avant qu'il ne soit trop tard !!

Le **compilateur**, avec ses yeux à lui, vérifie lui aussi votre code, puisque s'il contient des erreurs, il refuse de le compiler ou indique des warnings. Il est donc votre relecteur systématique ! Avec GCC, des options comme -Wall, -ansi ou -pedantic peuvent vous en dire plus et même ... vous amener à revoir votre jugement sur la qualité de vos travaux. VBCC possède quant à lui l'option -c99 qui signale quelques warnings en plus, comme par exemple l'utilisation d'une fonction sans prototype. SAS/C vous remonterait encore des alarmes différentes, c'est pourquoi il est très positif d'utiliser plusieurs compilateurs : ils ne laissent pas passer les mêmes choses.

On peut amener le parser d'un compilateur C à devenir plus restrictif mais il ne le sera sans doute jamais autant qu'un outil dédié comme splint. D'abord connu sous le nom de lint puis **lclint** (http://lclint.cs.virginia.edu/), on en est arrivé à **splint** (http://www.splint.org).
Splint veille sur les erreurs classiques d'utilisation de la mémoire désallouée ou de non-libération (fuite mémoire). Il se défend aussi bien avec les problèmes de déréférencement de pointeur, les macros douteuses, les boucles infinies, les valeurs de retour de fonctions ignorées, les risques d'incohérences sur les types, ... et autres joyeusetés ! Il est configurable à l'extrême et dispose de plusieurs modes plus ou moins stricts. Pour donner un ordre d'idée, sur un source, splint indiquait 8 erreurs en mode weak, 28 en standard et ... 123 en strict ! Assez vexant ...
L'installation de lclint (à récupérer sur Aminet) n'est pas évidente mais heureusement décrite sur le site guru-meditation. splint, plus conséquent encore, est uniquement utilisable sous MorphOS

pour l'instant (à télécharger sur MDC). De plus, il ne gère pas les varargs du standard C99.

Tracer l'exécution du programme

Contrôler les valeurs et la bonne exécution des fonctions, comprendre le parcours, etc. C'est en traçant ainsi le déroulement du programme que l'on peut se rendre compte de problèmes. Qui n'a pas commencé instinctivement à truffer son code avec d'innombrables appels à **ce bon vieux printf** ? On ne peut blâmer personne mais on évitera d'utiliser cette unique approche des plus basiques, même si elle rend de bons services très simplement.
La fonction **kprintf**, implémentée dans la bibliothque libdebug.a ou debug.lib, est une évolution de printf mais s'utilise de la même manière. Sa spécificité réside dans le fait que l'écriture s'effectue sur le port série. Le texte émis peut donc soit être lu à l'autre bout du câble série sur un terminal soit redirigé vers une fenêtre grâce à l'outil sashimi décrit par la suite.
Des programmes Linux comme strace ou cflow sont chargés de dessiner l'arborescence des appels de fonctions et représentent en ça un moyen de contrôle du bon déroulement de l'exécution.

Une cause très fréquente de plantage dans les programmes est l'accès à de la mémoire située à l'adresse 0 (pointeur NULL), par suite de l'utilisation d'un pointeur non initialisé. Dans d'autres cas, la continuation de l'exécution d'un programme n'a aucun sens si on rencontre une valeur totalement inattendue et inacceptable. C'est exactement dans ces situations où intervient la fonction **assert** qui nous vient de la bibliothèque standard du langage C. Elle permet d'interrompre le programme si une condition (un test de validité) échoue, par exemple :
assert(date_naissance < date_du_jour);
Si le programme avait ici permis la saisie d'une date de naissance future, il s'agit d'un bug et cet assert bien placé quitterait le programme avec un message d'erreur indiquant le lieu de l'échec dans le code.
Le programme final, pour peu qu'on le compile avec le define NDEBUG, ne souffre d'aucun ralentissement puisque le code concerné par les appels à assert ne sera pas inclus.
Note : La fonction assert fait l'objet d'un article entier sur le site guru-meditation.

Quoi de plus évident d'un **débogueur** pour traquer les défaillances d'un programme ? **GDB** est le plus célèbre d'entre eux. Pas forcément simple d'emploi et austère en lignes de commandes, cet outil GNU s'est imposé partout et se retrouve parfois encapsulé pour donner naissance à un débogueur graphique, comme celui de l'environnement StormC 4. Il est aussi intégré dans OS4 qui propose d'appeler GDB quand un programme subit une défaillance.
GDB a besoin que le programme étudié ait été compilé avec l'option -g. Ensuite, il peut en retirer toutes sortes d'informations (valeurs de variables, position courante, ...), placer des points d'arrêt, avancer pas à pas, etc.
SAS/C a son propre débogueur (CPR) et ses options de débogage comme DEBUG avec les valeurs LINE ou FULL. Quant à VBCC, il propose les options -g et -hunkdebug ainsi qu'un débogueur en ligne de commande.

Trouver le hit

Le hit est un terme apparu avec l'outil **Enforcer** de Michael Sinz. Il désigne tout accès illégal en mémoire. Il s'agit véritablement d'un bogue. Aucun programme ne devrait lever de hits. Mais commençons par apprendre comment les révéler. La partie suivante détaillera les outils qui les détectent.

SegTracker fera partie des programme à utiliser très tôt dans la startup-sequence. Il trace les chargements en mémoire et permet d'établir une correspondance entre une adresse mémoire et ce qui s'y trouve. Il permettra ainsi à d'autres outils de savoir quel programme et plus précisément quelle partie du code a produit un hit. Mais rien n'est exploitable en l'état, il est impétarif de mettre en place un dispositif pour recevoir physiquement les informations de débogage, soit par affichage à l'écran, soit par accumulation dans un fichier.
Note : les fonctions de SegTracker sont directement intégrées dans le système MorphOS.

C'est **Sashimi** qui est responsable de cette tâche. Comme pour le kprintf, il va détourner les données envoyées au port série vers un fichier ou une fenêtre, suivant les réglages. Sashimi utilise un buffer circulaire pour stocker et afficher les informations interceptées. Ce qui veut dire qu'arrivé en fin de buffer, les plus anciens messages seront écrasés. Des options permettent de régler la taille du buffer et bien d'autres choses.
Sashimi d'utilise en général assez tôt dans la startup-sequence afin de permettre éventuellement de déceler des problèmes au démarrage.

Grâce à Sashimi on peut visualiser le log des hits et des messages de debug lancés via kprintf. La provenance de chaque hit localisé grâce au travail de fond de SegTracker est décrite avec 3 paramètres : le nom du module en cause (exécutable, bibliothèques, ...) ainsi que le hunk et l'offset fautifs. Ces deux derniers sont relatifs à la structure des programmes sous AmigaOS : avec les informations de débogage, ils permettent de retrouver le fichier source et la ligne de code ayant provoquée l'erreur. Un outil est chargé de ce travail : **FindHit**, développé pour le développement avec SAS/C. Il s'utilise très simplement et se décline en plusieurs variantes à utiliser en fonction du compilateur : **GccFindHit** pour GCC et **FindHunkOffset** pour VBCC. Le SDK de MorphOS propose quant à lui l'utilitaire objdump (ppc-morphos-objdump plus précisément) à manier un peu différemment puisqu'il permet d'obtenir le source désassemblé dans lequel on localise l'offset incriminé et on constate à quel partie du code cela correspond.

Tout est désormais en place pour accueillir un outil qui saura générer le flot d'informations que l'on vient d'apprendre à contrôler et à exploiter. On est maintenant préparé à entrer dans la jungle de la gestion mémoire.

Mémoire : outils système

Les bogues liés à des problèmes de manipulation de la mémoire sont très communs. Ils provoquent des plantages aléatoires et développent par là un côté insaisissable. Mais rien n'est perdu si l'on sait s'entourer. Le sujet est difficile et sensible ; les outils varient en fonction de nombreux paramètres : compilateur utilisé, système, absence de MMU, ...

Fuites mémoires, dépassement de tampon ou écriture dans une zone non allouée, perte de pointeur, ... autant de termes qui illustrent les possibilités de bugs concernant la délicate gestion de la mémoire. Ces derniers ont tendance à se traduire pas des plantages systèmes aléatoires, sous notre système dépourvu de protection mémoire.

Il existe globalement deux orientations possibles, deux bases sur lesquelles compter en matière de débogage mémoire :

- **Enforcer**, l'outil historique
Il capture les accès illégaux, comme la lecture ou écriture dans les zones mémoires inexistantes et non autorisées (les premiers kilos-octets de la RAM). Il connaît plusieurs variantes comme **CyberGuard** (pour les Amiga équipés de cartes Blizzard) et MurForce.
Pour déboguer les allocations mémoire et lecture/écriture dans les zones non allouées, il s'utilise avec un complément comme **MungWall** ou **WipeOut** qui semble plus complet puisqu'il permet aussi de gérer les pools et de détecter quand un programme ne libère pas toute la mémoire allouée. Si MungWall est choisi, il faudra l'épauler de **PoolWatch**.

- **MuTools**, la gamme complète
Il s'agit d'une gamme complète d'outils architecturés autour de la "mmu.library". MuForce représente l'outil de base et joue le même rôle qu'Enforcer. Il détecte les accès à des régions mémoire non disponibles ou illégales.
MuGuardianAngel, utilisé conjointement à MuForce, remplace les solutions comme WipeOut ou MungWall/PoolWatch, garantissant une surveillance des accès aux zones mémoire non allouées. Il effectue également une vérification périodique de la mémoire et permet un désassemblage à la volée du code 68k lors d'un hit.

D'autres outils de débogage viendront en complément :
- **PatchWork** complètera la solution retenue, signalant une mauvaise utilisation d'une fonction système (appel non conforme à la définition des autodocs). Il vérifie aussi l'alignement de structures DOS par exemple.
- **BlowUp** mettra en évidence des utilisations non conforme comme l'emploi d'instructions processeur erronnées, les divisions par zéro, l'accès à de la mémoire non valide, etc. Couplé avec WipeOut, c'est LA solution pour les Amiga non équipés de MMU.

Note : la mise en place de telles stratégies (principalement les outils comme MungWall, WipeOut, MuGuardianAngel, ...) cause des ralentissements.

Mémoire : bibliothèques dédiées

Il existe une approche différente pour traquer les erreurs liées à la gestion de la mémoire. Il s'agit de bibliothèques qui permettent de faire appel à leurs fonctions plutôt qu'aux habituelles. Un précurseur est APurify qui lie l'exécutable avec une bibliothèque statique de débogage. Cet outil n'est malheureusement pas adapté aux compilateurs actuels.

Le salut provient du fantastique **mpatrol** qui recèle une richesse incroyable ! Développé sur Amiga à la base, il existe sous de nombreux autres OS. Il signale toutes les erreurs classiques

(accès dans une zone non allouée, libération incorrecte, ...) mais fournit aussi des statistiques comme le total maximum des allocations atteint pendant l'exécution du programme. Un inconvénient est que les fonctions à utiliser ne sont pas spécifiques à l'Amiga mais appartiennent au C standard. Par contre, mpatrol ne se limite pas aux allocations et libérations. Il surveille aussi les actions des fonctions comme realloc, memcpy, ...

Avec GCC, il faut juste compiler son programme avec les options -g et -pg puis ajouter les flags suivant à l'édition de liens : -lmpatrol -lbfd -liberty
Ce n'est pas obligatoire mais pour que mpatrol soit plus bavard, on aura pris soin d'ajouter l'appel à l'include "mpatrol.h" dans nos sources.

L'exécution contrôlé du programme se terminera par la création d'un fichier de log nommé "mpatrol.log". Si l'exécution a été commandée par "mpatrol monexecutable", alors le log est plus détaillé. On y trouve toutes les erreurs qui sont répertoriées de manière assez infaillibles et chacune pointe vers le module et la ligne en cause !

D'autres outils sont fournis dans l'archive. On trouve par exemple **mleak** qui travaille sur le log obtenu et indique très clairement les endroits où ont lieu des fuites mémoire. Citons aussi mprof qui ouvre au profiling, c'est à dire la détection des fonctions les plus fréquemment appelées et où le programme passe le plus de temps.

Optimisation

C’est le fantasme de tout programmeur ! Les progrès effectués par les compilateurs laissent de moins en moins de place à l’optimisation manuelle au niveau où beaucoup la situe, celui du jeu d’instructions du processeur. Nous allons voir que l’optimisation ne se résume pas à cela.

En quoi cela consiste ?

Optimiser, c’est améliorer le rendement, la productivité d’un programme en tirant le meilleur des ressources mises à contribution : matériel, système d’exploitation, outils de développement, ... Il faut en avoir une excellente connaissance de chaque élément mis en oeuvre. L’art de l’optimisation est tout sauf de la précipitation.

Une source principale d’optimisation réside dans le choix des algorithmes et des structures de données ; on appelle ça la macro-optimisation. La tentative de gain de quelques cycles processeur est vaine s’il est possible de faire de faire 10 fois mieux en réorganisant un algorithme qui aurait été codé à la hâte. Par exemple, il ne sert à rien d’optimiser une fonction de tri bulles si l’on passe à côté du quicksort, par méconnaissance ou par manque d’analyse.

L’important est d’optimiser les traitements lourds et appelés très souvent pour que l’effet soit plus visible. La responsabilité de chutes de performances revient majoritairement aux boucles mal agencées. Ou alors aux structures de données qui peuvent être organisées de telle manière qu’elles seront plus exploitable dans un but précis avec plus de rapidité. Un bon exemple est décrit dans l’ouvrage “Programmation graphique C/C++ et assembleur” de Michael Abrash, un des concepteurs de Quake, je vous renvois à cette lecture passionnante.

Il faut aussi penser à l’évolution. Pour les bases de données, il est inutile de vouloir par exemple limiter le nombre de tables pour simplifier l’architecture si chaque modification fait appaître par la suite une “verrue” qui pénalisera les performances. De la même façon, on peut trouver une astuce valable au début mais qui supporte mal la montée en charge en fonctionnement.

Enfin, **l’optimisation ne doit pas avoir lieu au détriment de la lisibilité du code et du respect des délais prévus**. Le temps de développement est souvent compté. Or, l’optimisation consomme du temps, tout comme la recherche d’un bug dans un source confus car soit-disant optimisé. Dans cette optique, dérouler une boucle ou utiliser un décalage pour multiplier par 2 n’est pas forcément utile ...

L’optimisation intervient dans une étape finale. Ce qui ne veut pas dire qu’on ne doit y penser qu’à la fin ; il y a une nuance. Il est suffisament difficile de faire un programme fiable et fonctionnel qui répond efficacement à un besoin. A quoi bon doubler les performances s’il n’est ni pratique ni exploitable (bogué) ?

Outils

Après ces grands principes, évidents mais pas forcément suivis, il est nécessaire de voir concrètement ce qui est réalisable.

Déjà, sachez que **le compilateur est notre ami**. Il est souvent plein de ressources insoupçonnables ! Il propose notamment des options de compilation célèbres comme -O2 et -O3 (cette dernière option n'est pas toujours bien supportée par nos compilateurs). Le mot-clé "inline" attribué à une fonction peut aussi rendre l'exécution plus rapide : au lieu d'être appelée comme une fonction traditionnelle, elle sera dupliquée dans le code, on évite ainsi de perdre du temps dans les passages de paramètres. Consultez la documentation de votre compilateur : comme on le voit, l'ajout d'une option peut remplacer l'optimisation recherchée pendant des journées entières.
Une autre approche possible est de cibler l'exécutable pour un processeur donné (060 par exemple) ou pour une gamme (020-060) et profiter ainsi de certaines particularités : pipelines du 68060, AltiVec du G4, ...
Les fonctions mathématiques, contenues dans une bibliothèque statique, peuvent aussi être choisies judicieusement lors de l'édition de liens (-lm, -lm040, -lmieee, ...). La sollicitation du FPU peut aussi modifier la donne. On regardera aussi les avantages que l'on peut tirer de "small code" et "small data".
Mais rien ne sert de compliquer à outrance, ayez juste la curiosité de vous intéresser aux options d'optimisation en temps voulu. Voici quelques réglages classiques qui peuvent contribuer à l'obtention de code plus rapide :

68k GCC : -O2 -m68020-60 -fomit-frame-pointer
68k VBCC : -O2 -cpu=680x0 -speed (-maxoptpasses=n)
68k SAS/C : CPU=680x0 OPTIMIZE
MorphOS GCC : -O2 -mmultiple -fschedule-insns2 -mfused-madd -ffast-math
MorphOS VBCC : -O2 -speed –madd (-maxoptpasses=n)

L'optimisation n'est pas toujours une fin en soi. Pour certains programmes, on privilégiera la compatibilité sur toute la gamme Amiga en omettant par exemple une option comme -m68020-60 ou -cpu=680x0.

Le **profiling** est une spécialité qui consiste à traquer les ralentissements (goulots d'étranglements) dans un programme. Les compilateurs possèdent une option qui leur permet d'ajouter du code utile au profiling : -prof pour VBCC (dans sa version 68k uniquement), -p pour GCC, PROFILE pour SAS/C, ... Ensuite, un outil externe intervient pour exécuter le programme ainsi compilé de manière contrôlée. Cet outil est propre à chaque compilateur : vprof, gprof et sprof si on garde l'ordre précédent. On connaît ainsi combien de fois chaque fonction a été appelée et combien de temps a duré son exécution. L'outil mpatrol, spécialisé dans la chasse au bogues liés à la mémoire, possède aussi une fonction de profiling.

D'autres outils procèdent différemment mais peuvent se révéler utiles. La commande **time** par exemple, qui nous vient du monde UNIX, lance l'exécutable à tester et établit un compte-rendu en affichant les ressources utilisées par le programme, dont la durée d'exécution.

Le langage assembleur va-t-il être notre dernier recours ? Je ne souhaite pas décourager son apprentissage car ça peut être formateur et c'est un domaine passionnant. Une fois que le programme fonctionne et qu'on a isolé une fonction à améliorer, on peut envisager la convertir en assembleur pour gagner encore en vitesse quand toutes les solutions faciles ont été écumées. Le mieux est d'isoler la fonction cible et de demander au compilateur de fournir le source assembleur correspondant, avec l'option -S de gcc et vbcc. Avec un oeil exercé, on peut se rendre compte de la qualité du code et des améliorations à apporter ... à moins d'envisager une réécriture complète. Parfois des instructions performantes ne sont pas forcément utilisées par le compilateur.

En début de chapitre, nous avons déjà évoqué la prudence à adopter avant de se ruer sur l'optimisation du code par de savantes écritures. Vous comprendrez d'autant mieux cette réserve quand vous aurez testé votre programme sur plusieurs configurations différentes. Face à une grande diversité du parc Amiga, on a autant de bus mémoire, de processeurs, de versions d'OS, de systèmes graphiques, de patchs, ... Voici un exemple qui me provient de mon expérience personnelle : pour TinyGL, je voulais accélérer des opérations en mémoire. J'ai donc effectué un programme de test avec de nombreuses manières de procéder et je l'ai fait tester sur plusieurs configurations. Résultat : les routines que je croyais optimisées ne l'étaient pas plus que d'autres ... puisque par exemple le bus mémoire ne permettait pas d'en bénéficier ou alors certaines optimisations constatées sur ma machine provoquait des ralentissements sur d'autres.

Utilité de l'optimisation

Avec la montée en puissance des machines, est-il encore nécessaire d'optimiser ? Oui ! Car c'est lié au respect des ressources. On aime le travail bien fait sur Amiga ... et on a suffisament crié à la honte quand, sur d'autres plates-formes, l'augmentation de puissance des processeurs était un argument facile pour bâcler le travail. Bien sûr il y a des limites et rien ne sert d'optimiser éternellement si le programme doit arriver après la bataille ... d'autant que la diversité des machines rend les choses difficiles. Et les résultats de certaines optimisations sont peut-être moins flagrantes du fait de la rapidité décuplée des Amiga PowerPC.

Quand on pèse tout ça, on s'aperçoit que l'optimisation n'est pas juste une affaire de jeu d'instruction et d'économie de cycles processeur. Optimiser, c'est reconsidérer son travail et tenter de l'observer avec un regard différent. Mais développer c'est aussi gérer son temps ... et la recherche d'optimisation à outrance ne servirait qu'à le gaspiller.

Restez simples et logiques, ne perdez jamais de vue le contexte de développement et utiliser les conseils génériques trouvés ça et là mais méfiez-vous toujours. Et surtout testez si les modifications sont efficaces. Un gain, ça se mesure !

Les portages

Un portage représente l'adaptation d'un programme à un OS pour lequel il n'a pas été conçu à l'origine. Il y a bien sûr la conversion de jeux qui caractérise l'activité de certaines sociétés mais également la "recompilation" d'outils provenant principalement de Linux. Ou du monde "libre" en général. On peut en effet obtenir les sources d'un nombre incroyable de projets (via des sites comme sourceforge, freshmeat, ...) qui font d'ailleurs parfois preuve d'une réelle ouverture par leur côté multi-plateforme. Pour qu'un logiciel soit facilement portable, soit il consiste uniquement en traitements de données et affichage texte, soit il doit être conçu dans une politique de migration facilitée (conception modulaire, choix du langage C ou C++, ...).

Les portages concernent donc des migrations de logiciels qui proviennent d'autres OS mais pas uniquement. On peut le voir dans le sens plus large d'adaptation ... au sein d'une même plateforme. Et concernant l'Amiga, certains programmes sont à retoucher pour qu'ils utilisent :

- les appels systèmes (AHI, timer.device, ...) plutôt que le matériel (Paula, CIA, ...)
- le langage C plutôt que l'assembleur pour faciliter l'évolution vers le PowerPC
- les compilateurs GCC et VBCC plutôt que SAS/C trop lié à des spécificités de l'Amiga (ce qui était un avantage en son temps)

Un portage peut se révéler très facile mais il peut aussi tendre vers le cauchemar ! Les choses se gâtent alors et il faudra mettre en place une véritable stratégie. Ce qui demandera notamment du temps, du réalisme, de solides compétences en programmation, ... L'important est d'y aller progressivement, c'est come celà qu'on apprend. Pour se faire la main nous vous conseillons de commencer par des outils texte Linux, des démos SDL puis des jeux simples, ...

Pour vous y aider, nous détaillons dans ce chapitre 3 catégories distinctes de portage : les adaptations de logiciels UNIX, la compilation avec la bibliothèque SDL, le passage d'AmigaOS à MorphOS.

Porter un programme UNIX

Pour les outils Linux ou déjà orientés multi-plateformes (MySQL, zlib, libpng, ...) on bénéficie d'un base commune, à savoir : langage C standard, conformité POSIX, ... Dans ce cas, il y a souvent très peu de changements à effectuer (quelques fonctions, chemins du filesystem, ...) : la recompilation s'avère très simple, pour le peu que l'on possède un environnement GeekGadget correctement installé. Configure ... et puis voila ! GCC, le standard sous Linux, est plus pratique mais certains projets open-source se compilent parfaitement avec d'autres compilateurs, comme VBCC.

Sinon, si cela ne fonctionne pas, c'est qu'il y a sans doute une barrière importante pour porter le logiciel en question : threads, API non supportée (GTK, libusb, ...). Pas facile de faire un doc là dessus: soit tout marche tout seul, et taper deux lignes suffisent (pas rare), soit il y a des problèmes, et là ça se règle au cas par cas.

Enfin, un résumé:

Ce qu'il faut: un environnement GeekGadget assez complet, avec les archives: (ftp://ftp.ninemoons.com/pub/geekgadgets/amiga/m68k/snapshots/current/)

Au coeur de GG se trouve GCC, le compilateur multi-plateforme par excellence.

Indispensables:
- ixemul-#?
- gcc/egcs
- binutils
- make
- pdksh
- sh-utils
- file-utils
- sed
- grep
- textutils

Conseillées, souvent utilisées:
- gawk
- diffutils
- patch
- perl
- findutils
- flex
- bison
- groff
- texinfo
- automake
- autoconf

Et pour compiler des applications X11, il faut les diverses archives X-#?.

La procédure:

1) Récuperer l'archive des sources de ce qu'on veut compiler.
2) lire les README/INSTALL, ... qu'il peut y avoir dedans
3) lancer un shell 'sh' et se placer dans le répertoire principal de l'archive source
4) taper './configure –help' pour voir la liste des options de configuration. Il y en a en général beaucoup, mais la plupart sont standard et en principe il n'y a pas trop lieu d'y toucher. Seule exception: –prefix pour spécifier l'endroit où on voudra installer l'executable. Les options spécifiques au logiciel qu'on veut compiler, s'il y en a, sont à la fin de la ligne.
5) lancer la configuration: par exemple './configure –prefix=/gg'. Ça dure en général un bout de temps, nos Amigas n'étant pas très rapides. En principe, cette opération n'échoue pas, à moins qu'il ne manque un package GG de base. Une fois l'opération terminée, des Makefiles et

eventuellement d'autres fichiers (config.h, ...) ont été créés.
6) lancer la compilation: 'make'. C'est là le point délicat. Si tout ce passe bien, on obtient l'executable. S'il y a des erreurs, c'est à gérer au cas par cas. Les erreurs les plus fréquentes sont un GG manquant (flex, ...), une fonction manquante dans les includes, ou une fonction manquante lors du linkage.
7) on installe, par 'make install' en général.

Quelques commentaires de plus:

- Compiler une application X11 est en général plus compliqué, car suivant les toolkits utilisés, on peut facilement tomber sur des bibliothèques qui ne sont pas dans GG. Il faut alors les recompiler soi-même.

- Les applications X11 n'utilisent pas toujours 'configure'. Parfois on tombe sur 'imake'. Ça existe aussi sur Amiga, et en principe, ça marche. Ça fait pas mal de temps que je ne l'ai pas rencontré, cependant.

- On peut compiler dans un autre répertoire que celui contenant les sources. C'est même recommandé, car ça permet d'avoir plusieurs répertoires de compilation, avec des options différentes. Supposons qu'on ait les sources dans dh0:sources, et qu'on veuille compiler dans dh0:build. On remplace l'étape 5 par:

```
cd /dh0/build
/dh0/sources/configure –prefix=/gg ...autres options...
```

- Il vaut mieux spécifier les chemins d'accès au format Unix au lieu du format Amiga (/dh0 au lieu de dh0:), que ce soit dans les options de configure, que dans les makefiles ou autre.

L'alternative SDL

Un portage plus complexe demande de convertir entièrement des groupes de fonctions touchant à des domaines très divers : affichage, son, timers, gestion des périphériques, ... De quoi en décourager plus d'un ! Heureusement, vous pouvez bénéficier d'une bibliothèque de fonctions prêtes à l'emploi et qui encapsule tout ça. Il s'agit évidemment de la SDL.

Cette bibliothèque a désormais conquis un grand nombre de développeurs et a acquis une bonne maturité sur notre système puisque nous sommes représentés sur le site officiel :
http://www.libsdl.org

Beaucoup de projets l'utilisent. Bien que son implémentation soit parfois critiquée, elle a le mérite de présenter une API simple et d'ouvrir une logithèque conséquente. Sur l'Amiga, la SDL est maintenue depuis longtemps maintenant par Gabriele Greco. On attend également le SDK de ChaoZer pour son implémentation nommée WarpSDL qui donne de bien meilleurs résultats (réécriture complète) et qui supporte l'AGA en plus du RTG.

Présentation

La SDL (pour Simple DirectMedia Layer) est donc une bibliothèque multimédia simple à mettre en oeuvre et destinée à fonctionner sur de nombreux systèmes d'exploitation, s'affranchissant ainsi de leur API. A la base, le but était de faciliter le portage de jeux Windows vers Linux. En effet, cette bibliothèque fournit toutes les primitives nécessaires pour accéder au matériel vidéo et audio, aux périphériques standards (joystick, lecteur de CDROM), aux événements système (actions de la souris, du clavier, ...) ainsi qu'aux timers.

Tous ces domaines sont exploités avec un degré d'abstraction appréciable, même si on subit une perte de performances. Cette dernière est toutefois limitée puisque dans le domaine de la video, la SDL exploite les capacités des cartes
graphiques et supporte OpenGL. Les résultats dépendent bien sûr de l'implémentation suivant les OS supportés (une demie douzaine officielle à ce jour). Cette bibliothèque multimedia est une aubaine pour l'Amiga et nous vous encourageons à tenter l'aventure et à adapter un maximum d'outils et de jeux, même modestes. De nombreux sources sont accessibles depuis le site officiel.

Recommandations

L'archive de la dernière version en date est à récupérer sur le site de Gabriele Greco qui maintient la version Amiga : http://ggreco.interfree.it/sdl.html
A ce jour, la version 1.26 propose une bibliothèque statique (pour 68k et MorphOS) et une dynamique (68k seulement). Les compilateurs supportés sont SAS/C, VBCC et GCC.

L'installation de SDL est détaillée sur le site guru-meditation donc on ne s'étendra pas mais nous vous conseillons vivement de lire avec attention à la fois la page de Gabriele Greco et le readme de l'archive. Ces documents contiennent normalement suffisament d'informations.

Lorsque vous recompilerez des programmes, vous devrez veillez à vérifier que l'include de la SDL répond bien à l'écriture <SDL/SDL.h> et non <SDL.h> ou pire "SDL.h". En effet, les includes doivent être placées dans un répertoire nommé SDL (tout comme les includes OpenGL se trouvent dans "GL"), lui même placé dans un répertoire d'includes commun à tous vos compilateurs (sous peine de copies inutiles).

La bibliothèque est prévue pour effectuer des sorties de debug, c'est à dire communiquer des informations sur le déroulement du programme et sur les appels SDL. Ces lignes de texte sont générées par la commande kprintf (contenue dans libdebug.a ou debug.lib à lier à l'exécutable) qui les envoie sur le port série ou dans une fenêtre si on les redirige avec l'outil Sashimi (voir le chapitre dédié au débogage). Si vous ne souhaitez pas les voir apparaître, il faudra inclure dans votre code la ligne suivante pour ne pas être ennuyé à l'édition de liens :
void kprintf(char *a, ...){}

Plusieurs commandes pilotant des opérations graphiques proposent un mode "software" et "hardware". Dans la mesure du possible, on évitera d'utiliser ce dernier mode, il semble responsable de la majorité des plantages avec les adaptations Amiga. Une autre source d'instabilité pourrait provenir d'un bug dans la gestion du son qu'on essaiera de désactiver pour constater son éventuelle responsabilité.

Pour un portage SDL comme pour n'importe quel autre développement, vous apporterez un soin particulier aux modifications effectuées, en utilisant par exemple des "#if defined(AMIGA)" pour isoler les parties spécifiques. Enfin, chaque changement devrait être signalé à l'auteur d'origine pour qu'il puisse intégrer votre travail dans les sources communes. Ceci garantit un support durable et éventuellement une meilleure représentation de l'Amiga parmi les autres systèmes.

On peut dire que pour SDL, le travail de portage a déjà été réalisé. Il ne reste bien souvent qu'à recompiler. Dommage qu'il n'y ait pour l'instant aucune solution simple pour utilisation la bibliothèque partagée sous MorphOS ... système pour lequel l'adaptation de programmes requiert quelques conseils.

Le cas MorphOS

Avec l'arrivée de machines uniquement équipées de PowerPC, on observe un regain de plaisir à utiliser son système, que l'on soit utilisateur ou développeur (la compilation avec GCC ne représente plus une pénible attente). Pour en tirer le meilleur parti, il est important que les logiciels évoluent pour exploiter au mieux les ressources matérielles. L'AmigaOS 4 n'est pas encore sorti mais de nombreux conseils que nous donnons pour MorphOS seront utiles quand le système et un SDK seront disponibles publiquement.
MorphOS offre une compatibilité des sources et des exécutables. L'émulateur intégré au système permet en effet de faire fonctionner les programmes 68k respectueux du système et avec un gain de vitesse appréciable ! L'émulation est tellement bonne qu'on pourrait se demander quel est l'intérêt de passer du temps à recompiler pour MorphOS (exécutabes ELF). Mais sur Amiga, nous choisissons de toujours oeuvrer pour plus d'efficacité, sans se reposer sur la montée en puissance des processeurs.
Quant aux sources, sous réserve de quelques modifications qui font l'objet de cette section, ils devraient se compiler plutôt aisément. A la base, MorphOS reprend l'API AmigaOS mais il a quand même la volonté d'aller au-dela des limites imposées et figées pendant des années.

Tous les programmes 68k ne fonctionnent pas et il faut dans ce cas apporter des corrections, ce qui représente réellement un portage. Nous allons voir les points sensibles pour s'y préparer.

Si le projet contient de l'assembleur, il faudra le repasser en C. A l'époque, l'assembleur était utilisé dans un soucis d'optimisation et cela était justifié. Désormais, avec le besoin de portabilité, le progrès des compilateur et la puissance processeur, ça donne lieu à de vastes débats ... Quoiqu'il en soit, le C devrait être la base, il est ensuite toujours possible de passer des fonctions critiques en assembleur PPC et d'effectuer une compilation astucieuse pour bénéficier à loisir de l'un ou de l'autre.

La première étape consiste donc à **obtenir un source en C recompilable sur un Amiga classique**. On s'attachera ensuite à retirer tous les accès directs au matériel : coprocesseurs (Paula, ...), CIA (timers, état de la souris, filtre audio, ...), etc. Si on souhaite continuer de supporter une version Amiga, il faudra user habilement des directives de compilation, du genre "#ifdef __MORPHOS__".

Bien souvent, le compilateur roi sur Amiga était le fameux SAS/C. A l'heure actuelle, on lui trouve

quelques inconvénients car trop spécifique (propres mots-clés, appels non-standard aux bibliothèques, passage de paramètres par registres, support particulier de “vararg”, ...). Le but est donc d’adapter ses projets en compilant avec GCC qui est le plus universel et de surcroît la solution officielle adoptée par MorphOS et OS4 mais on n’oubliera pas VBCC. L’essentiel est d’obtenir du code standard. Le fichier “SDI_Compiler.h” est intéressant pour définir des macros qui encapsule les mots-clés spécifiques de chaque compilateur : SAVEDS, ASM, REG, ... Il se trouve dans l’archive “CLib-SDI.lha” disponible sur Aminet.

Bien, désormais on se trouve face à un projet respectueux du systèmes et qu’il est possible de compiler avec GCC ou VBCC uniquement, sans l’aide d’un assembleur externe. On peut dès lors le considérer comme “portable”.

On peut enfin s’attaquer aux **spécificités dues au PowerPC et à MorphOS**. La plupart des concepts est expliqué dans une documentation à consulter sur le MDC (http://mdc.morphos.net), dans la rubrique “Porting”. L’inscription et la consultation de MDC sont indispensables à qui veut sérieusement développer sur MorphOS et bénéficier du support nécessaire. D’autant que le site s’étoffe et devient une base de connaissances appréciable.

Pour le PowerPC, par défaut, les données d’une structure sont alignées à une adresse multiple de 4. C’est à dire que le compilateur va naturellement faire suivre une variable de taille inférieure à 4 par une variable fantôme qui sert juste à obtenir un complément à 4 octets. En 68k, l’**alignement** s’effectuant sur 2 octets, il faut indiquer au compilateur que l’on doit conserver cette valeur pour garantir la compatibilité. C’est fait au moyen de la directive “#pragma pack(2)”. Toutes les includes système fournies avec le SDK sont patchées.

La documentation évoque aussi le passage de paramètres de type “vararg” (nombre de paramètres variable) et le FPU. Mais un point qui touche beaucoup plus de monde concerne les hooks. On évitera de les employer mais il n’y a parfois pas d’autre alternative (notamment dans MUI).

Un soucis récurrent dans l’adaptation de logiciels à MorphOS concerne MUI et touche à la manière dont on doit encapsuler les **hooks** et leurs variantes, comme les **dispatchers**. Déjà, de manière générale et dans la mesure du possible, il faut éviter les hooks. Leur problème est qu’ils utilisent en paramères des variables associées à des regitres imposés et propres au modèle 68k. Sous MorphOS, il est donc indispensable d’encapsuler ce passage de paramètres. Des macros comme M_HOOK ou DISPATCHER existent pour cela. On en trouve plusieurs implémentations, dont une respectivement sur MDC et dans le SDK.

Dans l’environnement GCC de MorphOS, les inlines se trouvent dans le répertoire particulier “ppcinline”. Ces fichiers ainsi que les protos et stubs doivent d’ailleurs être générés de préférence avec le script Perl “cvinclude.pl”, plus adapté que fd2pragma.

La fonction **CreateNewProcess** d’Exec prend une suite d’arguments au moyen des fameux tags ou par passage d’une taglist. Sous MorphOS, elle nécessite l’ajout du tag suivant à encadrer d’un “#ifdef __MORPHOS__” :
NP_CodeType, MACHINE_PPC

Enfin, le GCC de MorphOS considère que le char est unsigned alors que pour les autres

compilateurs (vbcc, gcc 68k, sas/c), char est signed par défaut. Mais la norme n'indique rien, la liberté appartient au compilateur. On peut forcer ce dernier à interpréter "char" comme entier signé avec l'option -fsigned-char. Il faudrait sinon corriger chaque occurence dans les sources.

Et OS4 ?

Même si les systèmes OS4 et MorphOS ont choisi deux voies techniquement différentes pour leur évolution, leur programmation présente des similitudes : passage du 68k au PPC, API AmigaOS 3.1 comme base commune, utilisation de GCC par défaut, ...
Le SDK sera fourni gratuitement et disponible à partir de la sortie d'OS4. Il est déjà fonctionnel et en partie nettoyé (intégration des includes pour la programmation réseau par exemple). GCC 3, le compilateur qui s'impose, produit du code soit 68k soit PPC. Pour générer des exécutables OS4 via le jeu de cross-compilation, il est possible de développer sur une multitude d'hôtes : OS3.9 68k, OS4 68k, Linux x86, Linux PPC, Win32 et MacOSX ! Une version de VBCC est également disponible.

Les autodocs ont été mises à jour pour intégrer des correctifs mais aussi des nouveautés puisque de nouvelles API voient le jour, notamment AmigaInput pour gérer les périphériques.

Le débogage semble simplifié avec l'outil GrimReaper qui laisse complètement libre en cas de hits et d'erreurs. Il propose : informations générales (registres, type d'exception, module et adresse du crash), désassemblage, fermeture de l'application, redémarrage de l'ordinateur, prise en main du débogage sous GDB, ...

Le SDK doit tout de même être finalisé (ce qui devrait être fait d'ici à la sortie de l'OS) et rien est arrêté sur l'organisation du support aux développeurs.

Les portages peuvent être un bon moyen de produire quelque chose et de progresser avant de se lancer soi-même dans un projet qui serait à mener de bout en bout.
Alors vous savez ce qu'il vous reste à faire : **trois, deux, un ... portez !**

Technologies 3D

Encore une fois, dans une optique de pérénité, on conseillera d'opter pour un apprentissage OpenGL. C'est une référence dans la 3D, qu'on le veuille ou non, au même titre que le langage C dans la programmation généraliste.

Comme pour la question de l'assembleur, la création de son propre moteur 3D n'est aujourd'hui envisageable que dans quelques cas précis : apprentissage, démos, ... Pour un jeu par exemple, il y a déjà tellement à faire ... à quoi bon créer une bibliothèque de rendu de scènes 3D si le travail que ça représente fait trainer le projet et que ça finisse par ne pas aboutir ?

Faisons le point sur les solutions disponibles, leurs caractéristiques principales et leur disponibilité.

Implémentations OpenGL

OpenGL est une API de haut niveau consacrée à la création de scènes et d'animations en 3D. Mais qui peut le plus peut le moins : son utilisation en 2D peut être valable pour peu que les opérations soient câblées, c'est à dire effectuées par la carte graphique. Les bases d'OpenGL furent définies par un maître en la matière : le constructeur Silicon Graphics. Pour contourner le système de licence, le **projet MESA** a été mis en route à l'initiative de Brian Paul. MESA est un équivalent open source d'OpenGL, proposant à la base uniquement un rendu logiciel mais un système de pilotes autorise les accélérations matérielles.

Le supplément GLUT propose une interface commune à toutes les plateformes pour la gestion des périphériques : affichage fenêtré, souris, clavier, timer, ... Celà hisse à un niveau encore plus haut la portabilité d'applications OpenGL.

StormMesa
Sans doute la plus ancienne implémentation. On la doit à l'éditeur Haage & Partner. La dernière version (3.0, basée sur MESA 2.5) date de décembre 1998 et supporte Warp3D. StormMesa devrait satisfaire un large panel d'utilisateurs puisqu'il est accessible aux possesseurs d'une machine 68k ou PPC WarpOS, en AGA ou équipée d'une carte graphique.

TinyGL
Cette bibliothèque ne contient que les fonctions principales d'OpenGL mais permet un apprentissage facilité. L'avantage réside dans sa simplicité de mise en oeuvre et sa petite taille. Il existe deux versions bien distinctes de TinyGL :
- une de type static, disponible sur Aminet, utilisable sous 68k (AGA/CGFX) et MorphOS avec plusieurs compilateurs, mais qui ne propose que du rendu logiciel (lent).
- une intégrée à MorphOS en tant que bibliothèque partagée du système, qui utilise pour l'instant uniquement les pilotes profitant de l'accélération matérielle.

MiniGL
Cette bibliothèque bénéficie de pilotes matériels par le biais de la couche bas niveau Warp3D (voir ci-dessous). Elle fonctionne donc en 68k ou WarpOS. Par contre, elle ne propose pas de rendu logiciel, excepté grâce à un projet externe de Stéphane Guillard qui nécessite WarpOS (ne fonctionne pas sous MorphOS). MiniGL fonctionne sous MorphOS mais sa lience interdit sa recompilation sous autre chose qu'AmigaOS.

JunGL
Cette adaptation de MESA ne fonctionnera que sur MorphOS (bibliothèque OpenGL officielle pour ce système). Elle bénéficiera d'une implémentation complète et à jour des fonctions OpenGL 1.4 et supportera à terme l'accélération graphique.

Autres solutions

Il est tout à fait possible d'écrire une API de rendu 3D en partant de rien. On peut aussi, et c'est préférable de nos jours, opter pour OpenGL. Mais coincé entre temps de développement ahurissants et lourdeurs éventuelles, vous pourriez souhaiter une alternative intermédiaire. Et c'est possible ! Deux solutions existent :

Warp3D
C'est la solution la plus communément rencontrée dans les démos et les jeux. Elle est émulée sous MorphOS et donne de bons résultats. Une nouvelle version appelée Nova est prévue pour OS4. C'est donc sensé fonctionner sur toute machine accélérée (Permedia, Voodoo, ...).

Rave3D
Citée uniquement par soucis d'exhausitivité puisque Rave3D ne possède pas d'API publique. Plus ou moins standardisé par Apple, cette API n'a jamais vraiment eu de succès auprès du grand public. Elle utilise exclusivement des pilotes matérielles pour bénéficier de performances optimales. Sous MorphOS, eller sert à TinyGL et à l'émulation de Warp3D.

Woof3D
C'est la solution originale qui sort du lot. Elle propose une approche haut niveau (certaines fonctions facilitent la vie) sans sacrifier les performances et elle vous permettra d'effectuer des développements multi-plateformes : 68k RTG (pas d'AGA, attention) et Warp3D si présent, Linux x86 et Windows.
Sur Amiga, Woof3D propose un rendu logiciel mais aussi matériel en encapsulant Warp3D. La prochaine version pourra faire appel aux fonctions OpenGL, c'est à dire par exemple TinyGL sous MorphOS (d'où support 3D par la carte graphique).

Au final, le choix d’une solution 3D dépend du système d’exploitation, en préférant bien sûr une solution bénéficiant d’accélération matérielle ou au moins facile à mettre en oeuvre, c’est pourquoi on déconseille les bibliothèques bas-niveau telles que Warp3D.

- MorphOS : à moins d’avoir besoin de toute l’étendue de l’API OpenGL, on choisira TinyGL en se rabattant sur la version soft en dernier recourt
- Amiga 68k : TinyGL 68k ou StormMesa ou MiniGL
- WarpOS : MiniGL, en supposant que la PPC est équipée d’une carte graphique et que Warp3D est installé
- AmigaOS 4 : aucune solution officielle annoncée mais elle devrait reposer sur Nova, le nouveau Warp3D