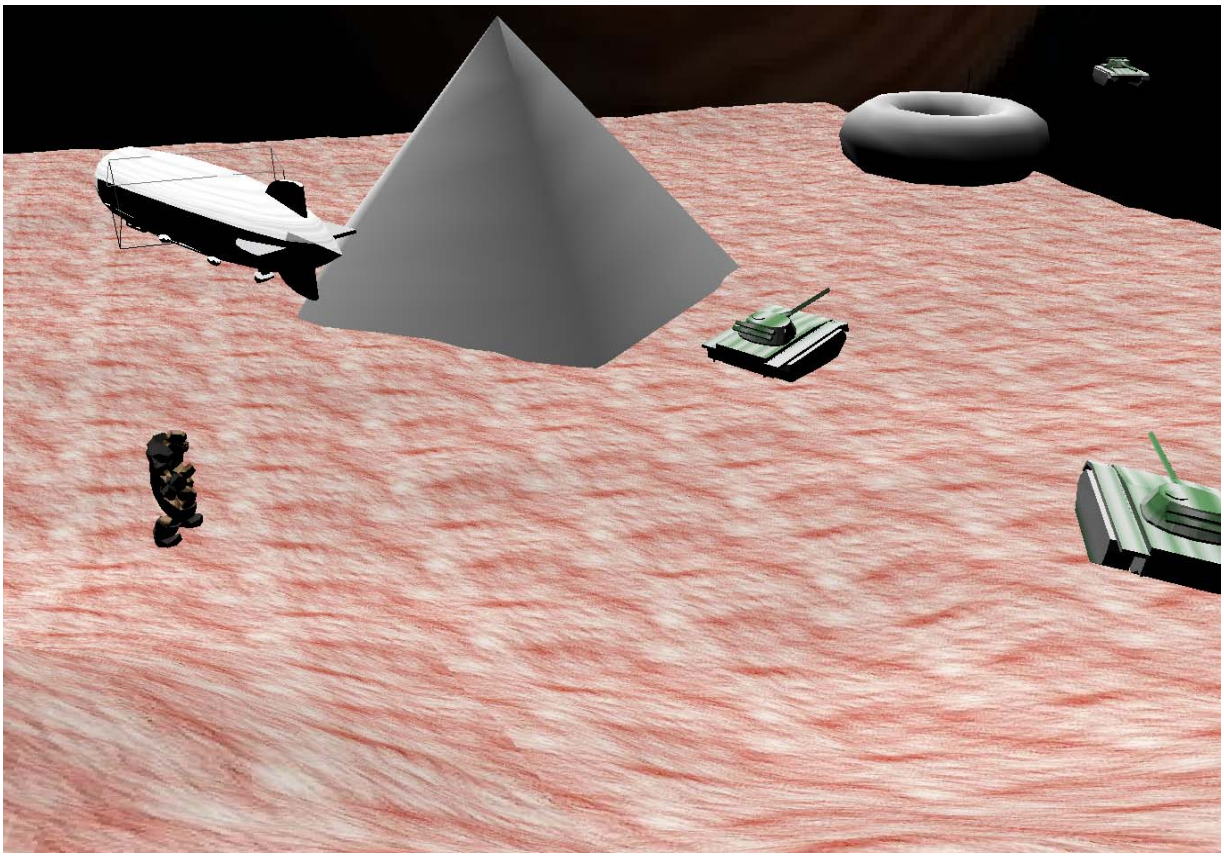
G3C
Generic Gnu Game Core



Document de référence du collaborateur

V 1.0 (02/04/2004)

Auteur : Quentin Ochem (simboy@users.sf.net)

Note sur le présent document

Ce document est destiné aux collaborateurs qui rejoignent le projet G3C, et à ceux qui souhaitent préciser leur connaissance du projet. Il s'agit, sous une forme condensée et élaguée, d'un résumé technique et stratégique des choix pris dans la conception du projet. Les points qui sont introduits ici sont appelés à être précisés dans d'autres documents qui leur sont dédiés. Il est bon pour le collaborateur de prendre connaissance de l'intégralité de ce document avant de se lancer dans le projet, afin d'avoir une vue globale de l'intégralité de ce que nous souhaitons faire.

Des informations complémentaires peuvent être trouvées sur Internet, à l'adresse du projet : http://www.sourceforge.net/projects/g3c/. D'autre part, toutes les remarques sur ce document sont les bienvenues. Merci de les adresser à g3c-developers@lists.sourceforge.net.

1. Généralités

1.1. Prologue

Le joueur se connecte à Internet, via l'interface du jeu. Il accède alors directement à sa planète d'origine, celle ou s'est posée pour la première fois l'âme qu'il contrôle. Cette âme, c'est une entité immatérielle capable de prendre possession des êtres sapiens et de les diriger. Sous ses ordres, il possède le maire d'un petit village, et de tous ses habitants. Il ne dirige pas directement la plus part d'entre eux, nombreux sont ceux qui ont un esprit contrôlé par une intelligence artificielle. Parmi les autres, il y a un architecte, un commerçant et deux soldats dont il a délégué le pouvoir à d'autres joueurs, à d'autres âmes. D'ailleurs, le maire et tous les habitants du village eux mêmes ne sont pas sa propriété exclusive, c'est l'âme qui dirige le gouverneur de la région qui les lui as assigné, ce gouverneur lui même étant en réalité la propriété d'un empereur local. Le joueur aurait pu, bien sûr, vouloir posséder ses propres créatures, mais il aurait fallu alors qu'il fonde une nouvelle civilisation à partir d'une des espèces extra-terrestres sauvages perdues de part la galaxie, et ça, il ne s'en était pas sentit capable. C'est pour cela que la gestion de ce village lui convient, pour l'instant. Bientôt, si son village grandit assez et devient une cité, il lèvera une armée. Il choisira des joueurs humains pour diriger ses commandants de guerre, pour animer ses plus puissants soldats et pour manoeuvrer ses chasseurs spatiaux. Il rejoindra les frontières de son monde et ira combattre aux côtés de son empereur les royaumes voisins. Il participera peut-être aussi à la recherche scientifique, deviendra diplomate, partira à l'aventure pour résoudre l'une des énigmes de l'univers, qui sait. Les possibilités de ce qu'il fera n'auront pour limites que ce qu'il pourra imaginer. Ces frontières ont été repoussées par un groupe de programmeurs, qui a créé le noyau du jeu auquel il est en train de jouer. Ce noyau, c'est G3C, « The Generic GNU Game Core ».

1.2. Présentation du projet

Le projet G3C a pour but la création d'un noyau de jeu. Ce noyau devra être au maximum réutilisable dans tous les jeux qui souhaitent gérer des mondes temps réel en trois dimensions. Les jeux du type *Doom* ou *Starcraft* devraient donc pourvoir utiliser les mêmes routines, avec simplement des options de configuration différentes.

G3C est avant tout un framework, c'est-à-dire une structure permettant de formater une conception. On cherche, au plus haut niveau, à faire le moins possible de choix techniques, afin que divers projets puissent utiliser différentes technologies. L'exemple type concerne l'affichage graphique. A priori, un utilisateur peut indifféremment choisir d'utiliser openGL, ou DirectX (ou même une librairie 2D type SDL s'il ne veut pas faire d'affichage 3D). Au plus haut niveau de G3C, on ne manipule que le concept d'objet graphique et on ne sait pas comment il est réalisé. Pour simplifier le travail du développeur, on propose cependant une implémentation standard dans un module annexe (G3L pour Generic Gnu Game Library) qui utilise openGL et qui permet à la plupart des utilisateur de ne pas avoir à se poser la question de la créations des outils 3D. De manière générale, lorsque l'on parle de G3C, il faudrait (mais dans ce document on abusera généralement du langage) différencier G3C stricto-sensus, et G3L qui propose une implémentation de G3C et qui en précise certaines caractéristiques.

G3C, c'est aussi un certain nombre d'outils permettant de créer facilement des jeux en réseau. La structure classique client serveur est bien évidemment disponible, ainsi qu'une structure univers persistant. Le point culminant de cette structure, et l'une des plus grandes difficultés techniques, c'est la volonté que nous avons de créer un univers persistant peer to peer, c'est-à-dire sans serveur centralisé. Cela peut paraître difficile, voir impossible à priori, nous verrons par la suite que certaines techniques et certains postulats nous permettent de prétendre à ce résultat.

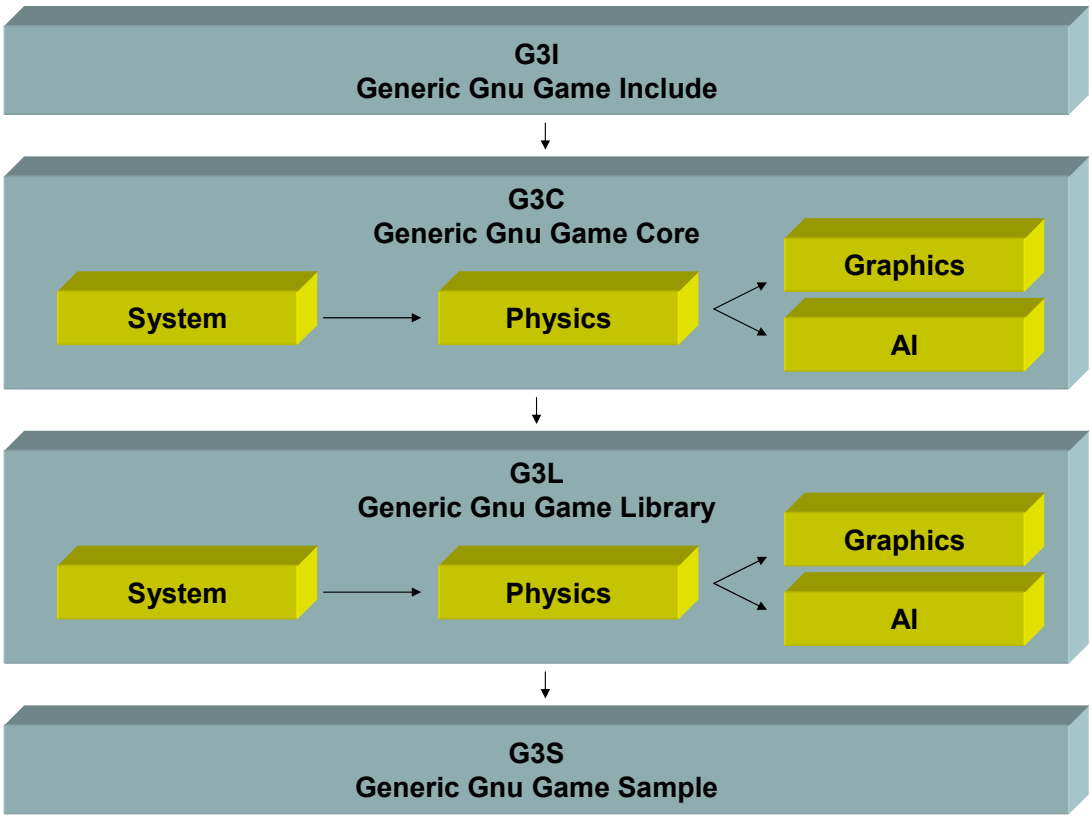
Il est évident que l'ambition de ce projet dépasse largement ce qui peut être espéré réalisé à court et moyen terme, avec les moyens dont nous disposons. Aussi, nous souhaitons rapidement être en mesure de proposer des versions intermédiaires, qui n'implémentent pas forcément toute l'idée que nous en avons. Nous travaillons sur une réalisation itérative.

En parallèle à ce travail de noyau, nous concevons un programme d'exemple sensé illustrer les fonctionnalités offertes par notre projet. Il n'est pas exclu que cet exemple devienne un jeu autonome, aussi toutes les aides au niveau du design graphique ou de la scénarisation sont les bienvenues.

Pour finir, G3C a pour ambition d'être intégralement portable linux et windows. En conséquence, les développements se déroulent en parallèle sur les deux plateformes. Les volontaires souhaitant s'intéresser au Macintosh seront largement appréciés.

1.3. Structure gros grains

Dans le schéma ci-dessous, représentant l'architecture générale du projet G3C, les flèches représentent les dépendances entre les différents sous projets :



Le sous projet G3I contient un certain nombre d'outils de bases utiles dans tout le programme. Ces outils sont tellement génériques qu'ils pourraient même être utilisés dans des programmes qui n'ont rien à voir avec les jeux vidéos. Il s'agit d'une liste chaînée, d'un ensemble de fonctions pour le calcul matriciel, etc.

Le sous projet G3C contient le cœur du programme, le framework ainsi qu'un certain nombre de fonctionnalités génériques. On fait ici le moins de choix d'implémentation possible, et on laisse la possibilité au programmeur de ne travailler qu'avec les outils disposés à ce niveau. Il y a quatre parties distinctes pour ce sous projet :

- System : Fournit des primitives utilisées pour des notions d'utilisateur, de connexion réseau, d'unité de mémoire, de synchronisation…
- Physics : Permet de coder un univers cohérent, possédant des objets et subissant des lois simulant les lois naturelles.
- Graphics : Propose un moyen d'interfacer l'utilisateur avec le mode via un affichage graphique ou une interface utilisateur.
- AI : Décrit les divers comportements associés aux entités de l'univers.

Le sous projet G3L contient une ou plusieurs implémentations des modules de G3C. Les quatre sous parties sont donc les mêmes que dans G3C. Il existe une vraie question que l'on se pose souvent et qui n'a pas toujours de réponse claire : vaut t'il mieux coder un module dans G3C ou dans G3L ? Est-ce générique ou doit t'on pouvoir choisir autre chose ? Cette question mérite d'être débattue dans beaucoup de cas, et les réponses peuvent changer selon l'état d'avancement du projet.

Enfin, le sous projet G3S n'est rien d'autre qu'un exemple d'utilisation de la libraire, lequel exemple est candidat à devenir le premier jeu utilisant G3C.

2. Objets physiques

2.1. Principe d'un objet physique

On considère que tout ce qui est matériel est un objet physique. Dans l'idéal, même le sol est un objet physique. Ces objets sont composites, c'est-à-dire qu'ils sont l'union de plusieurs morceaux d'objets plus petits. Chaque objet plus petit a ses propres propriétés de résistance, de poids, éventuellement d'interactivité (par exemple, un canon peut tirer, des jambes peuvent avancer, etc.). Le joueur doit pouvoir d'une part réutiliser des objets déjà tout faits, d'autre part être capable, via un langage de script et un petit éditeur, de créer ses propres objets.

On ne sait pas encore jusqu'à quel niveau sont décrit les objets élémentaires, va-t-on jusqu'à en décrire chaque pièce (ici un panneau de fer, là une vitre en plastique), ou propose t'on d'office une bibliothèque d'objets plus complexes ? Ce qui est certain, c'est que pendant l'exécution du programme, les objets complexes nécessiteront de très nombreux calculs. C'est pour cette raison qu'il faudra créer en parallèle un espèce de « compilateur d'entités physiques », capable d'effectuer des calculs à l'avance pour simplifier la structure d'un objet une fois assemblé.

Afin de standardiser la représentation des objets, et de les rendre accessibles dans les différents modules de G3L qui ne sont pas forcément codés dans le même langage que le noyau, les différents fichiers qui leur sont relatifs seront codés en XML.

2.2. Principe des règle physiques

On applique sur les objets un certain nombre de règles physiques. Parmi elles, la plus connue et peut-être la plus difficile à écrire est la règle de détection de collisions. L'objectif est de ne pas fixer le nombre ni l'implémentation de ces règles (bien qu'un grand nombre sera fourni en standard dans G3L). On peut penser à des règles comme la gravitation, le vent, l'usure de certains matériaux, le feu… Le programmeur doit pouvoir en ajouter, en retirer ou en créer à loisir, selon les performances des machines dont il dispose. Les règles seront à même, et ce dynamiquement, d'ajouter des variables aux objets de l'univers.

3. Affichage graphique

3.1. Liaison entre objet physique et objet graphique

Lorsque l'on crée un objet physique, on ne spécifie pas sa représentation graphique. L'objet peut tout à fait évoluer dans un mode sans aucune sortie écran. Il est bien évidemment nécessaire pour le joueur de le lier à un ou plusieurs objets graphique, mais aucune contrainte n'est posée. Ces objets graphiques pourront être réalisés via openGL ou DirectX, mais aussi par des moteurs 2D comme SDL. On pourra ainsi par exemple afficher directement une carte en implémentant un objet graphique comme étant un point de couleur, différent selon le possesseur de ce point.

3.2. Interface avec l'utilisateur

Il faut considérer G3C comme étant une sorte de boîte noire, un univers auquel on peut accéder et que l'on peut modifier de différentes façons, via des interfaces différentes. Une interface, c'est deux parties distinctes : une scène (l'affichage) et un outil d'interaction (via le clavier, la souris, le joystick…). On peut interagir sur beaucoup de choses différentes, et de manières aussi variées. Par exemple, un soldat interagira sur un seul personnage, en vue subjectif, et sera capable de lui donner de nombreux ordres (à gauche, à droite, tire, saute, etc.) alors qu'un général (vue de haut) possèdera un très grand nombre de soldats mais ne pourra leur donner que des ordres plus généraux (attaque, déplace, stop, etc.).

4. Interactions entre les joueurs

4.1. Notion d'âme

Contrairement aux autres jeux on-line, on n'associe pas nécessairement le joueur à une unité, ou à un groupe d'unité. On peut par exemple imaginer qu'un général laisse temporairement le contrôle d'un de ses soldats à un joueur en doom-like, ce même général s'est d'ailleurs vu octroyer temporairement son armée par un empereur qui dirige une partie d'une espèce qu'à créé un dieu. On peut admettre donc que l'unité appartient au dieu, mais qu'elle est gérée successivement par un empereur, un général et un soldat. Lorsque le soldat aura rempli sa mission, il est possible que le général le récupère pour autre chose. Dans un jeu classique, le joueur disparaîtra complètement.

On va donc associer un joueur non pas à une unité, ou à un groupe d'unité, mais à une identité que l'on appelle âme. Cette identité n'a pas nécessairement de lien physique, même si elle peut être temporairement rattachée à une ou plusieurs unités. Cette identité est rattachée à des informations la concernant, on peut savoir quel a été son passif, et c'est en fonction de ce passif qu'elle aura accès à différentes ressources de la part de différents joueurs.

Bien sûr, certain joueur pourront avoir leur personnage tout au long du jeu. Les règles peuvent différer d'un jeu à l'autre, mais dans la version la plus générale, on peut imaginer qu'il est possible de l' « acheter ». Tout dépend du type de jeu, et du type de joueur.

5. Fonctionnalités système

5.1. Propositon peer to peer

On veut pouvoir monter un univers persistant à l'aide d'une architecture peer to peer. Le principe de base est que chaque machine possède et gère un petit bout de l'univers. On espère donc pouvoir obtenir un univers de taille proportionnel au nombre de machines sur le réseau. Au-delà de la question des performances d'un tel choix (qui sont assez difficiles à prévoir à ce niveau), de nombreuses questions de correction sont posées. Voici les principales :

- Comment garantir le non piratage des données ? Etant donné que le programme est ouvert, n'importe qui peut virtuellement faire n'importe quoi. Cette question est d'ailleurs plus large : une machine peut ne pas tenir la cadence, avoir une mauvaise version du jeu, s'arrêter brusquement. Pour toutes ces raisons, on introduit une grande redondance des données. Chaque section de l'univers est dupliquée autant de fois que nécessaire, et les machines se contrôlent mutuellement. Si l'une d'entre elles tombe en panne, les autres sont encore là pour assurer la persistance de l'univers. Si l'une d'entre elle donne des résultats erronés, toutes les autres vont pouvoir s'en apercevoir et la mettre de côté.
- Comment continuer à faire exister l'univers si il n'y a plus une machine connectée ? On admet qu'il existe toujours un nombre n de machines connectées, et ce nombre est posé grand. La question est plutôt comment parvenir à atteindre ce nombre. Plusieurs solutions complémentaires sont possibles. La première idée est d'internationaliser le jeu. Les joueurs des États-Unis ne seront pas connectés en même temps que les joueurs européens. Mais il y a une solution plus forte : rémunérer la participation des joueurs, du genre « plus vous laissez vos machines tourner, plus vous gagnez de points, plus vous pourrez faire de choses dans l'univers, avoir beaucoup d'unités, etc. ». On incite donc les joueurs qui possèdent des liaisons type câble/ADSL de laisser tourner leurs machines toute la journée, même lorsqu'ils ne jouent pas. On peut même surenchérir ce dernier point en offrant un service payant de serveurs dédiés au jeu, dont les joueurs achèteraient des tranches de temps qui leur rapporterait des points.

Ces solutions ne doivent pas détourner le lecteur de la vraie question : celle de la complexité liée à la surcharge de la redondance et aux limitations réseau. Si on augmente l'univers de n unités, faut t'il ajouter n machines ? x*n machines ? x^2 machines ? Cette valeur est bien évidement dépendante des solutions qui seront trouvées, et on aura comme premier objectif de la diminuer.

6. Gestion des comportements

6.1. Les quatre niveaux

On dénombre quatre possibilités de gestion des comportements, qui pourront être utilisés en simultanée dans un jeu :

- Le pantin : Ce sont les comportements les plus basiques du jeu. Ils suivent des procédures pré écrites et immuables. C'est aussi le comportement habituel des unités dans les jeux vidéo, mais nous allons tenter de faire autrement.
- L'automate : Il est identique à l'automate, en ce sens qu'il suit « bêtement » un code qui détermine son fonctionnement. Cependant, l'automate peut être programmé via un langage de script pour réagir différemment.
- L'androïde : L'androïde est une intelligence artificielle évolutive, capable de s'adapter, d'apprendre, de réagir… Plusieurs implémentations sont envisageables, réseaux de neurones, algorithmes génétiques, logique formelle… Elles peuvent d'ailleurs être implémentées en même temps pour différentes unités.
- Le cyborg : Un cyborg est directement connecté à un vrai joueur. C'est la manière la plus efficace de gérer une unité, mais également la plus coûteuse en termes de ressources (le nombre de joueur n'étant pas illimitée).

On note que chaque nouvelle classe est un sur ensemble de la classe précédente. Ainsi, un automate peut avoir des parties programmées de type pantin, un androïde peut faire appel à des procédures automates et un cyborg peut utiliser des ordres androïdes.

7. Cadre de développement

7.1. Un projet GNU

La licence qui a été choisie pour G3C est la licence GPL (General Public Licence), rédigée par la GNU Software Foundation. Il s'agit sans doute du choix stratégique le plus important dans la création du projet, et ce à cause des nombreuses contraintes de cette licence, mais également des nombreuses possibilités qu'elle permet. En voici la substance :

- Un programme GNU est libre, en ce sens que son code source doit être fourni en même temps que son exécutable.
- Un programme GNU est libre dans le sens ou il est impossible d'interdire la copie de ce code source ou sa modification par un tiers.
- Tout programme qui possède une partie de son code GPL, ou qui utilise directement ou indirectement (par un appel de librairie par exemple) du code GPL doit être lui-même GPL.

Il existe une licence amoindrie, la LGPL (Lesser General Public Licence), qui autorise des programmes non-GNU à utiliser du code GNU. Nous sommes sincèrement convaincu que le GNU peut être une alternative véritable aux produits classiques, et c'est dans cette optique que nous avons choisi de garder la version forte de la licence, imposant à des projets professionnels et commerciaux qui souhaiteraient utiliser G3C d'être GPS eux même. Nous verrons plus tard comment faire de l'argent même avec ces contraintes.

On remarque facilement le problème majeur du GPL : il est impossible de protéger un logiciel contre la copie pirate. Cependant, on peut considérer les avantages suivants :

- Il existe un nombre incalculable de produits GNU qui peuvent être réutilisés à volontés, et parmi eux on en compte beaucoup de qualité professionnelle. C'est un support facile à utiliser pour développer une application GNU.
- Étant donné qu'il n'y a pas de notion de propriété, ni de gain direct, il est assez facile de trouver des collaborateurs prêts à travailler gratuitement. Leur travail ne sera pas volé, et servira dans un domaine qui correspond à un certain idéal. Il est gratifiant (et c'est peut-être aussi un peu pour cela que nous l'avons choisi) de travailler pour le logiciel libre.
- Au niveau de l'utilisateur, on peut s'intéresser de l'image qu'à le GNU. Même si elle reste encore médiocre auprès des professionnels, le GNU véhicule auprès d'un large public des valeurs de liberté qui le rend naturellement enclin à se tourner vers ce genre de solutions.
- La licence GPL offre une vraie protection pour le code. Interdire son utilisation sous peine d'être soi même GPL rend par exemple inintéressant pour un concurrent de récupérer le code et de l'améliorer pour son compte, étant donné que nous pourrons nous aussi bénéficier des améliorations.

Ce choix de GNU ne nous interdit cependant pas de travailler avec des outils professionnels si le besoin s'en fait sentir (compilateurs, environnement de développements, outils de créations graphique). Il nous est même possible d'utiliser des librairies propriétaires (DirectX par exemple) dans notre code. De manière générale, la licence GPL nous autorise à utiliser pratiquement tout ce qui existe sur le marché.

7.2. Un code en Ada

Choisir Ada comme principal langage de programmation peut paraître étrange à première vue. Etrange, parce que la plus part des gens programment aujourd'hui en C/C++, et qu'imposer Ada comme langage de base en forcera beaucoup à apprendre un nouveau langage, dans le meilleur des cas, et à ne pas participer au projet, dans le pire. Alors pourquoi imposer Ada ? Un langage qui, comme beaucoup le disent, est nettement plus lent que le C++ ?

Tout d'abord, il convient de démentir ce dernier point. Ada n'est pas plus lent que le C++, ses performances sont largement comparables, voire même meilleures dans certains cas. Le mythe selon lequel Ada est moins rapide que le C++ vient de deux faits : tout d'abord, Ada effectue énormément de tests à l'exécution, pour prévenir d'éventuelles erreur de programmation, ensuite ses performances sont énormément dépendantes des compilateur, le langage étant beaucoup plus éloigné de la machine que le C++. Les premiers compilateurs Ada étaient effectivement extraordinairement mauvais, et construisaient des programmes très peu optimisés. Aujourd'hui, un compilateur tel que GNAT (que nous utiliserons) bénéficie exactement de la même technologie que C (via gcc) pour passer de l'arbre syntaxique au code assembleur. D'autre part, le nombre et la position des tests sont réduits au strict nécessaire, ce qui évite les nombreuses redondances dont souffraient les premiers compilateurs. Le seul reproche qu'on peut faire, et que l'on pourra toujours faire, à Ada concerne la taille des exécutables, sensiblement plus importante que celles de ses homologues en C/C++. Dans un monde où le prix du Mégaoctet est en perpétuelle décroissance, et où finalement seule comptent les Mégahertz, une telle limitation n'entre pas en ligne de compte.

Sachant maintenant que du point de vue des performances, ces deux langages se valent, pourquoi choisir Ada ? Revenons en arrière, concernant le C dont est issu le C++. Ce sont des étudiants d'une université américaine qui l'on mis au point, pour pouvoir développer ensuite leur système d'exploitation, une version gratuite d'Unix (qui donnera Linux plus tard). Le langage n'a pas été l'objet d'une vraie réflexion en termes d'intelligence de la syntaxe, mais juste pour pallier à un assembleur ingérable. Il est donc largement permissif, et une programmation propre réduit au minimum l'utilisation des facilités syntaxiques qu'il propose. Le problème est qu'il est fréquent, à la suite d'une erreur d'inattention, de tomber dans l'une de ces facilitée non attendues et de créer ainsi un programme qui plante. Le débugger peut alors prendre de longues heures. Ada a été pensé par un groupe de personnes qui recherchait un langage capable de résoudre à la compilation le maximum d'erreurs d'inattention, et de déceler à l'exécution avec le plus de précision possible les bugs. Ce qui rend les programmes Ada particulièrement fiables et fonctionnels par rapport à leurs homologues C.

D'autre part, et même après l'avènement des technologies objet, partager du code en C++ reste une complexe affaire de normalisation. Les fichiers d'entête sont souvent incompréhensibles et l'utilisation antédiluvienne des macros rend énormément de codes illisibles. Ada est fondé sur une séparation stricte des spécifications et de l'implémentation, ce qui rend les fichiers d'entête tellement clairs qu'il est, la plus part du temps, inutile d'y joindre la moindre documentation.

Enfin, la norme C++ évolue perpétuellement, et aucun compilateur n'est tenu de la respecter, ce qui entraîne que chacun y lit à peu près ce qu'il y veut. Un programme

parfaitement fonctionnel avec un compilateur ne passera pas la phase de compilation pour un autre. Concernant Ada, un compilateur n'est pas autorisé à ne pas respecter la norme. Cela rend, entre autres, la portabilité Linux / Windows quasi sans douleurs. Cette portabilité est l'un des points du cahier des charges du jeu Âmes.

Pour ces raisons, il nous a semblé préférable d'innover, et d'imposer un langage habituellement cantonné à des applications critiques, comme les calculs embarqués ou les logiciels militaires. Ada n'est pas un langage intrasèquement difficile à apprendre. Sa syntaxe a beaucoup de points communs avec Pascal, et implémente tous les concepts objets classiques. Il existe un grand nombre d'outils gratuits permettant de l'utiliser, environnement de développement, débuggeurs, compilateurs, et son intégration avec d'autres langages est parfaitement possible. Nous espérons que les programmeurs intéressés ne nous tiendront pas trop rigueur de notre choix, dans les premiers temps, et sommes persuadés qu'ils l'approuveront dès qu'ils se seront essayés à ce langage.

7.3. Buisness model

Il peut paraître déplacé de parler ici de buisness model, puisque nous ne sommes pas en train de créer une activité rémunératrice au sens propre du terme. Cependant, GNU n'est absolument pas incompatible avec vente ou gain d'argent, et la question de monter une activité professionnelle autour de G3C ayant été posée, l'éventualité a été envisagée.

Il faut considérer deux cas de figure : soit on vend une librairie (on s'adresse donc à des professionnels du jeu vidéo), soit c'est un jeu final dont il s'agit (donc vendu à un utilisateur grand public). Le second point doit de toute manière être abordé car, même si l'on ne s'intéresse qu'aux éditeurs de jeux vidéos, le problème demeure : comment pourront t'il vendre leurs produits, et par quels moyens les protéger contre une utilisation pirate ?

Concernant le premier point, on ne peut pas vendre une bibliothèque GNU seule. Elle peut être gratuitement obtenue auprès de n'importe quel support de distribution (Internet dans notre cas). Mais l'on peut vendre des services autour de cette librairie. Cela implique que l'entreprise possède l'expertise du programme, et dans son utilisation, et dans sa structure interne. On imagine par exemple vendre cinq choses différentes :

- Du support technique : s'il y a un bug, il est corrigé par l'entreprise. S'il y a une incompréhension, ou des difficultés dans l'utilisation du programme, l'entreprise est là pour aider le client.
- Des garanties : le programme marche. Si ce n'est pas le cas et que cela coûte de l'argent au client, il y a prise en charge par l'entreprise soit au niveau humain (réparation des dégâts matériels, correction de bugs) soit au niveau financier (réparation des pertes), selon les modalités du contrat.
- De la formation : le personnel du client qui doit être formé pour l'utilisation des outils de l'entreprise peut l'être par une équippe d'experts détachés de cette dernière.
- Du travail de régie : Le client qui souhaite disposer d'une partie de son programme développé par un sous-traitant peut choisir l'entreprise. Si il souhaite avoir dans son équipe un développeur ou un ingénieur spécialiste du programme, il peut également louer ses services.

- Du service conseil : Le client qui se trouve devant un problème impliquant l'utilisation du programme peut demander conseil à l'entreprise.

On peut penser encore à d'autres services (certification, label de qualité, etc). La liste n'est pas close, l'important est ici de démontrer qu'il y a des solutions.

Concernant la question de l'utilisateur final, là encore, impossible de faire payer le programme directement. En utilisant G3C, qui est sous GPL, le jeu final doit lui aussi être sous GPL, c'est-à-dire que l'on ne peut interdire sa reproduction et que l'on doit en fournir les sources. Le piratage deviendrait en quelque sorte « légal ». Deux solutions à ce problème :

- Fournir des binaires de qualité : Si on ne peut interdire la copie d'une source, on peut par contre interdire celle d'un programme binaire donné (à vérifier), ou bien celle de librairies non libres que ce binaire utiliserait. On peu par exemple proposer un programme compilé avec un compilateur plus puissant que les outils de base, ou bien utilisant des librairies payantes. La copie de l'exécutable devient alors interdite, bien que n'importe quel utilisateur peut toujours compiler lui-même le programme, ou obtenir une version de base qui sera moins performante mais tout aussi fonctionelle.
- Faire payer un système de serveur : Dans le cas d'un jeu on-line, il est possible de faire payer toute ou partie des serveurs Internet, sachant que les utilisateurs auront normalement la possibilité de créer leurs propres serveurs. Là encore, c'est une question de performances : les serveurs créés par l'entreprise seront plus performants que ceux mis en place par les joueurs eux-mêmes.
- Utiliser des images et des textes protégés : Bien que le code soit libre, l'environnement artistique qui est autour (textes, images, vidéos, musiques) peut, lui, être protégé. Il devient donc possible de faire des copies légales du code, mais sans ce qui est finalement le plus important : l'ambiance du jeu.
- Offrir une belle boite avec de la documentation : C'est une chose capitale que l'on oublie facilement, mais bien que le piratage soit devenu une généralité, les gens continuent à acheter des jeux, parfois même lorsqu'ils en possèdent une version pirate. Pourquoi, sinon que leur acte d'achat est psychologiquement réellement important ?

On remarquera que le capital image de l'entreprise est absolument fondamental. En effet, rien n'empêche à priori un concurrent de faire la même chose sur le même code. Il est indispensable que les joueurs se reconnaissent suffisamment dans l'entreprise pour considérer l'offre concurrente inintéressante.

Pour achever la justification de ce buisness model, on peut affirmer que le GNU ne parviendra à s'imposer qu'en sachant s'introduire dans le model commercial déjà en place. Les plus idéalistes pourront regretter que la notion de liberté initialement contenue dans la licence y est peut-être moins nette. Il s'agit cependant d'une progression sans conteste par rapport à des stratégies propriétaires, parfois et écrasantes, qu'il sera inutile de nommer.

7.4. Utilisation du système

Une question importante à se poser concerne le moyen d'utiliser le système. Comment le programmeur pourra t'il appeler les fonctionnalités de G3C ? A cette question, il y a quatre solutions, et nous nous efforcerons de permettre les quatre.

Premièrement, le programmeur souhaite coder en Ada, et il peut inclure directement le code de G3C dans son exécutable. C'est là qu'il aura les meilleures performances, et la plus grande souplesse. Ceci dit, deux problèmes se posent : tout d'abord, on a tendance à préférer l'utilisation de libraires partagées, ensuite le langage Ada est souvent méconnu au profit du C, et peu de programmeurs seront prêt à faire la migration.

Si le programmeur souhaite quand même programmer en Ada, il peut utiliser une librairie interfacée en Ada. Il perds alors certaines fonctionnalités, comme celles liées à la généricité.

Si il veut absolument coder dans un autre langage, il passera obligatoirement par une interface C++ (qui est peu ou prou le dénominateur commun de tous les langages). A l'aide de cette interface, il peut compiler un code mixte ada/C++.

Enfin, et c'est la moins bonne solution sur le plan technique mais la plus pratique (et sans doute celle qui sera la plus utilisée), il peut récupérer les fonctionnalités de la librairie partagée via une interface C.

Ces informations tendent à démontrer qu'il existe deux niveaux de spécification : les spécifications internes et les spécifications externes. Les premières sont celles des programmes Ada, et intègrent toutes les fonctionnalités objets, les secondes sont celles de programmes en C, terriblement limitées. Peut-être même trop limitées (exit la généricité, l'héritage, les exception…). On peut se demander si on ne pourrait pas mettre en place un système de description orienté objet, du type des interfaces COM utilisées dans Windows. On pourrait aller jusqu'à se permettre une modification du compilateur que nous utilisons, pour introduire une façon non standard d'interfacer des objets (ce qui aurait comme défaut de nous rendre plus ou moins dépendant d'un compilateur). Toutes les idées sont les bienvenues.

7.5. État d'avancement

Un premier prototype de plus de 15 000 lignes en C++ a été terminé fin janvier. Il a permis de boucler une première phase de conception, et il est disponible sur le CVS de Sourceforge en consultation.

Un environnement de développement linux et Windows est disponible, permettant de compiler et d'exécuter des programmes sur les deux systèmes. Cependant, l'accélération graphique 3D ne semble pas fonctionner sous Windows avec les outils que nous utilisons, c'est un problème à investiguer.

Le nouveau code fait maintenant quelques 10 000 lignes en Ada, qui sont pour un très grand nombre d'entre elles uniquement des spécifications. Concrètement, le programme affiche un terrain plat et un cube au milieu de ce terrain. Le succès de ce petit programme, c'est d'avoir fait cohabiter étroitement du code C++ et du code Ada. En effet, pour une raison à ce jour inconnue, il est impossible sous Windows de faire les appels à openGL directement en Ada (à investiguer). Ces appels sont donc faits via du C++. De la même façon, il manque

un certain nombre de caractéristiques à GtkAda, qui conduisent au fait que le code des fenêtres est lui aussi réalisé en C++.