Sommaire

	Introduction	**3**
I.	**Présentation du sujet et prise de connaissance du sujet**	**4**
II.	**Mise en place du projet**	**6**
III.	**Etude des différentes librairies**	**11**
IV.	**Le composant de base d'Allegro : le bitmap et la tuile**	**14**
V.	**Le scrolling et la gestion des sprites**	**19**
VI.	**La GUI (Graphic User Interface)**	**25**
VII.	**Bilan**	**28**
	Conclusion	**29**
	Cahier des charges	**30**
	Planning	**39**
	Manuel technique	**40**
	Annexes	**47**

Introduction

Nous avons choisi ce projet pour avoir une autre vue de la programmation. Nous ne voulions pas faire un logiciel dont nous connaissions la plupart des étapes à suivre, nous voulions apprendre à concevoir un programme ludique, un jeu vidéo qui ne suivrait pas forcément un développement pour lequel nous nous sentions bien formé.

Les buts étaient, tout de même, différents entre les membres du groupe de projet. Pour certains ce projet leur a permis de pouvoir approcher l'univers des jeux vidéo Open Source sous Linux. Pour les autres, ce projet a permis d'assouvir une curiosité envers la réalisation de jeu vidéo. Nous sommes conscients que ce projet englobe beaucoup de compétences dans lesquelles nous ne sommes tous au mieux que des débutants. Le système Linux bien qu'il ne nous soit pas étranger, n'était pour nous surtout qu'un système d'exploitation compatible avec nos enseignements de programmation à l' I.U.T. De même, le développement de logiciels libres Open Source nous était totalement inconnu. C'est pourquoi durant tout le projet nous nous sommes attachés à comprendre les mécanismes du fonctionnement de tels logiciels.

Nous allons vous présenter tout d'abord, les méthodes pour cerner ce sujet ainsi que notre point de vue quant à la démarche à suivre. Par la suite, nous expliquerons l'organisation et la mise en place de notre environnement de travail. Nous poserons dés lors, le système d'informations à développer en rapport avec la librairie graphique choisie. Ensuite, pour chaque module réalisé, nous décrirons l'analyse et les réalisations mises en oeuvre pour répondre aux objectifs. Enfin, nous finirons par un bilan rétrospectif pour valider ou non chaque élément du cahier des charges.

I. Présentation et prise de connaissance du sujet

1. Nature

Enoncé du sujet : 1)Développer un jeu d'arcade/action en C sous Linux (et Windows si possible).
2)La fluidité du jeu est primordiale.

Descriptif: XjAC est un shoot-em up avec un scrolling.

Un vaisseau jouable évolue dans une carte grâce à un scrolling. La carte est un élément statique tandis qu'un ennemi, par exemple, est un élément mouvant. Un tableau est une phase dans le jeu, il se compose de ces 2 types éléments et se termine lorsque le vaisseau arrive à la ligne d'arrivée de ce tableau. Ensuite, le tableau suivant s'offre au joueur.

Le vaisseau entrant en collision avec un élément statique ou mouvant est détruit, perd une vie et le joueur doit recommencer le tableau en cours au début. Si le vaisseau est détruit et qu'il ne reste plus de vie, la partie est terminée. Si le joueur recommence une nouvelle partie c'est a partir du 1er tableau.

Buts:

XjAC fonctionnera sous Linux et sous windows (si possible).
XjAC répondra dans chaque étape du développement à la philosophie OpenSource
Le jeu doit être fluide et rapide.

Limites:

Les graphismes du jeu sont en 2D (tableaux, éléments, menus)
La résolution minimale est 320x200 en 256 couleurs en mode fenêtré
ou en plein écran

2. Définitions

XjAC : X-window Jeu d'Arcade en C.
OpenGL: Regroupement de librairies graphiques dont le code source est disponible selon le critère Open Source.
OpenSource: Critère rendant disponible légalement le code du programme
Notice: Fichier-manuel expliquant un accés ou un procédé d'utilisation d'une ressource informatique.
Partie: Une session de jeu XjAC.
Tableau: Une phase dans une session de jeu. Cette phase a un départ et une arrivée.
X-window: Une interface graphique en standard sous Linux, sert de plate-forme pour les gestionnaires de fenêtres.

KDE, Debian: Différents gestionnaires de fenêtres utilisant X-window.
GUI: Graphic User Interface. Interface graphique pour l'utilisateur final.
Utilisateur final: Le joueur jouant à XjAC.
Bitmap : Carte de bits, tableau à double entrée, structure de données de base qu'offre Allegro.
Ecran physique : Est représenté par le bitmap vidéo affichable directement sur l'écran (BITMAP * screen).
Ecran logique : Il représente le bitmap stocké en mémoire dans le but d'être affiché (memory_bitmap ou video_bitmap).C'est un bitmap intermédiaire ou buffer, finalement il sera « collé » sur screen.
Couche : Implémentation en tableau d'un ensemble contenant tout les éléments de même nature.
Tuile ou tile : Elément graphique élémentaire de plus haut niveau et plus spécifique que les BITMAP d'Allegro.
blitter, blittage : Copier/Coller une bitmap (utilisation de la méthode blit())

3. Contraintes Générales

Recherche de libraire graphique OpenGL codée en C/C++ avec les ressources informatiques de l'iut.
Programmation en C/C++. Avec une nette préférence pour le C car il est de plus bas niveau.
Développement : Linux Mandrake 8.00 KDE (O.S personnels).
Bus d'information : Linux Debian (Bordeaux). Informations sur le projet consultable à l'iut.

4. Ressources

Internet (à l'IUT) :
Salle de discussion, site hébergeant des projets de librairies en cours (www.sourceforge.net, www.openGL.org, www.kde-developper.org)
Site conseil sur des librairies déjà testées (chat, faq, newsgroup)
Tutoriels et manuels au format HTML (pages internet)
Sites hébergeant des bibliothèques d'images

Utilitaires: StarOffice 5.1 pour les textes. KDE Pixmap2Bitmap. KDE Paint, The Gimp, ScreenCapture pour les images.
Livres de programmation : GTK, Applications portables en C/C++, Linux
Stockage : Lecteur Zip mis à notre disposition par M Dabancourt, Disquette 1.44 Mo

II.Mise en place du projet

1. Mise en place d'un système qualité et Open Source:

Pour le projet XjAC, un système qualité a été mis en place, afin de satisfaire au mieux aux exigences du cahier des charges et à la «philosophie» Open Source. Pour cela nous avons voulu mettre en place des outils de développement et de communication efficace et facile d'utilisation. De plus il fallait que ses outils soient à la disposition de tous afin de normaliser les méthodes de travail. Le programme XjAC doit pouvoir fonctionner sous Linux, par commodité tous nos outils de travail sont des outils fonctionnant sous Linux.

Outils de développement, de rédaction et de communication :

-*Allegro :* Outils de développement de XjAC. Les différents aspects d'Allegro seront décrit dans le chapitre « L'étude des différentes librairies graphiques »

-*StarOffice :* Outil de rédaction. StarOffice est un logiciel de traitement de texte gratuit, développé par SUN.

-*Pine :* Outil de communication. Cette messagerie est simple d'utilisation et d'accès, puisqu 'elle est disponible sur Bordeaux, serveur debian mise en place pour les projets.

La mise en place de ce système nous permet de savoir l'avancement des tâches confiées aux membres du groupes. Ce qui nous a permis de pouvoir répartir les tâches de façon plus précise, pour correspondre aux fonctionnalités que nous devons réaliser.

Nous avons normalisé nos méthodes lors de notre recherche, afin de comparer nos résultats plus aisément particulièrement durant la période de test des libraires graphiques. De plus, la normalisation du code a été appliqué tout au long du projet. Le fait de normaliser selon les critères définis dans la documentation d'Allegro permet de faciliter la lecture du code.

La normalisation du code se manifeste par :

-une description courte des méthodes à l'en-tête de celles-ci.
-un commentaire sur chaque variable.
-un nom explicite pour chaque méthode.

Nous avons ainsi pu dégager les grandes phases lors de l'exécution d'un code à savoir :

Initialisation des ressources à engager dans le système de jeu
Manipulation des ressources
Destruction des ressources

2. Compte-rendu, Notice et Rapport

Pendant le projet, il a fallu se réunir pour pouvoir définir les objectifs, donner des directions de travail,etc.. . A chaque réunion une personne était chargée de prendre des notes qui ont resservis lors des réunions suivantes afin de valider par nous-même nos réalisations.

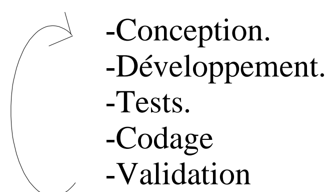
Les notices nous ont permis de faciliter l'installation des plate-formes de développement et de pouvoir retrouver facilement les erreurs commises lors de l'installation. Nous avons commencé à faire des notices dés le début du projet, lorsque nous avons eu des problèmes pour l'utilisation ou l'installation des différents logiciels. Les notices sont rédigées par l'utilisateur du logiciel de manière claire et précise pour être réutilisées par les développeurs. (voir Annexe/Notices)

La rédaction des rapports a toujours été faite selon un modèle normalisé :

-Définition des objectifs/besoins .
-Précision de l'environnement de travail.
-Conception pour produire un résultat incluant toutes les contraintes.
-Critique du résultat pour une amélioration.
-Synthèse des éléments.

3. Organisation du projet

Le développement du projet a suivi un modèle itératif. Chaque modules du projet suit une phase identique de développement :



-Conception.
-Développement.
-Tests.
-Codage
-Validation

Nous avons voulu détailler dans le cahier des charges chaque phase du développement logiciel dont voici les principales étapes :

Tout d'abord nous avons pris connaissance du sujet par M. Dabancourt. Il s'agissait de faire un scrolling avec une librairie graphique Open Source. Ensuite, nous avons effectué une recherche sur différentes librairies graphique Open Source, afin de déterminer laquelle correspondait le plus à nos besoins, cette tâche nous l'avons nommé recherche de l'existant. Il fallait s'assurer que les fonctions offertes par les librairies testées pouvaient permettre de coder un scrolling fluide.

Vu le grand nombre de paramètres à régler pour configurer les librairies graphiques, il était important de faire des tests et de configurer une plate-forme de travail commune pour le groupe de projet. C'est pendant cette tache que nous avons produit le plus de notices, il fallait équiper tous les postes personnels avec les différents outils liés au projet et les procédures indiquées dans des notices.

A)L' environnement de travail

Dans le système d'exploitation Linux, tous les logiciels installés utilisent des fonctions de base du noyau Linux et ajoutent leurs nouvelles fonctions.

L'ajout d'Allegro nous a permis d'avoir des outils évolués pour la programmation de XjAC, cependant nous avons utilisé des fonctions définies dans le noyau Linux pour la programmation de certaines fonctions. Nos plate-formes de travail sont constituées de trois couches logiciels, qui sont toutes indispensables pour pouvoir utiliser XjAC.

-Le noyau Linux. Ce noyau offre des routines (le systéme de fichiers, accés mémoire). Les fonctions définies dans ce noyau sont les plus rapides à s'exécuter. Ce noyau est lui-même composé de librairies notamment les librairies C/C++.

-Le serveur graphique Xfree 86 4.0.0. Ce serveur graphique est fournit avec la version de Linux Mandrake 8.0.0 que nous avons installé sur nos postes. Xfree86 est nécessaire pour utiliser le système X-Window. Ce dernier est l'interface graphique standard sous Unix. X-Window fournit les primitives graphiques pour dessiner des éléments de GUI. X-Window est un environnement puissant supportant différentes applications(jeu, outils de programmation, programmes graphiques, éditeur de texte,..). Xfree86 est la version libre du système X-Window pour Linux.

-La librairie Allegro. Cette librairie est la dernière couche logiciel ajoutée sur nos postes de développement. Cette libraire utilise des fonctions du noyau Unix et de Xfree86 et elle propose aussi des outils de développement dont nous avons besoin.

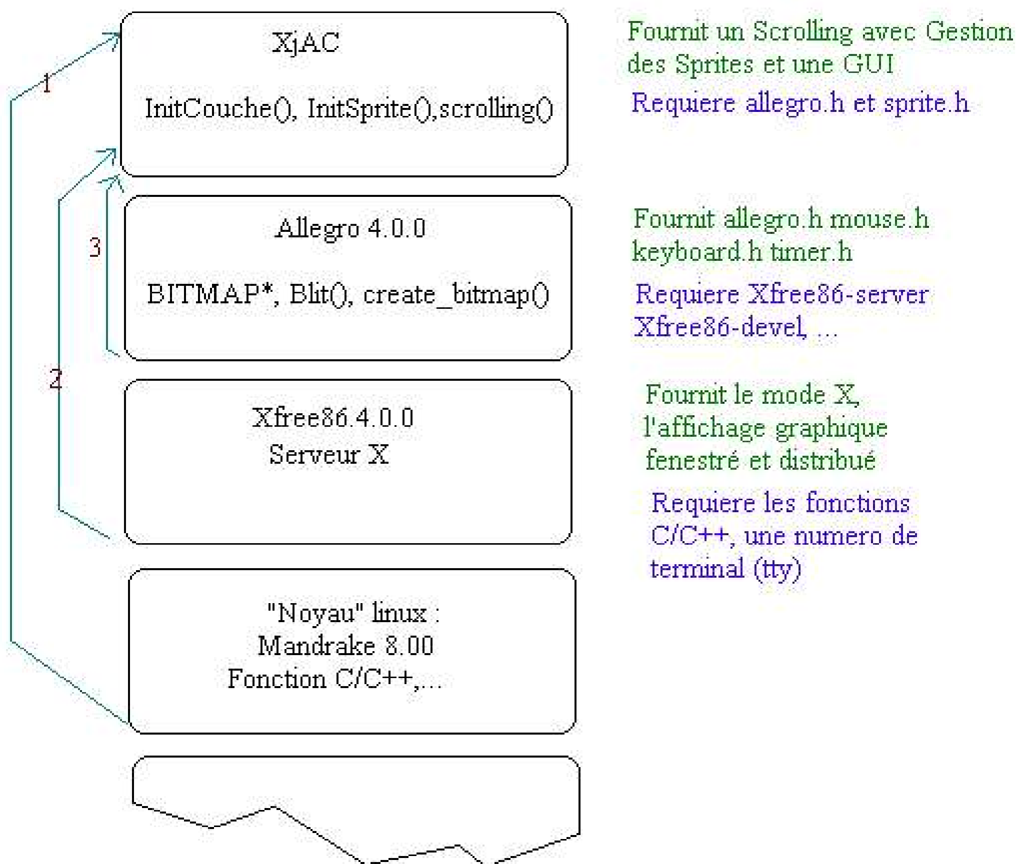


Schéma de l'environnement de développement : les couches logicielles

Des liens entre chaque couches existent. Elles sont représentées sur le schéma précédent par des flèches.

Flèche 1:

Dans le programme X-JAC nous utilisons des fonctions définies dans le noyau Unix. Le noyau offrait des méthodes de bas niveau donc trés rapide d'exécution, nous les utilisons pour la création de tableaux, l'allocation de la mémoire, renvoi de paramètres (new,delete,return). Le système des entrées-sorties du jeu X-JAC (gestion des niveaux) utilise aussi des fonctions du noyau.

Flèche 2:

Lors de la compilation de X-JAC, Allegro utilise le fichier *allegro.cfg* afin de configurer par exemple les paramètres d'affichage. Si ces paramètres ne sont pas définis, il utilise les paramétres de Xfree86 pour configurer cela automatiquement.

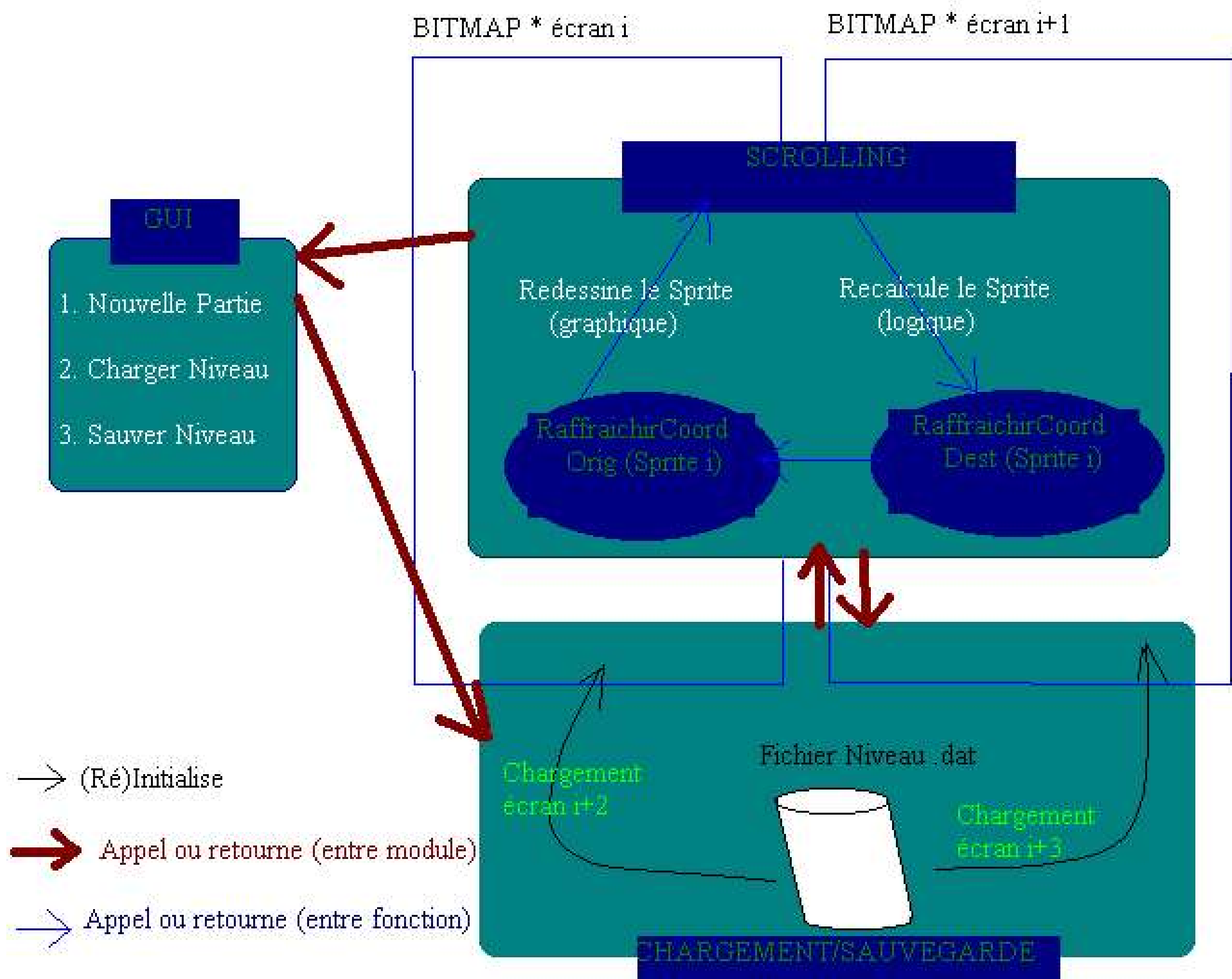
Flèche 3:

Les éléments affichés lors de l'exécution du jeu, le scrolling et la GUI, utilisent des méthodes d'Allegro. De plus, nous avons aussi besoin d'Allegro pour la définition et l'utilisation des objets que nous utilisons dans notre programme (BITMAP, blit() en autres...).

B)Le modèle du système

Le développement en lui-même a été décomposé en quatre modules indépendants : le scrolling, la gestion des sprites, le chargement/sauvegarde de fichier DAT du niveau, et l'interface utilisateur GUI (menu du jeu). L'utilisateur lance le jeu, accède d'abord à la GUI puis en choisissant dans le menu une nouvelle partie, décide de lancer le scrolling dans lequel est inclus la gestion des sprites et le chargement/sauvegarde du niveau.

Voici le modèle du système en cours d'exécution:



III.L'étude des différentes librairies graphiques

Durant ces recherches de librairies, nous avons étudier 4 grandes bibliothèques, elles semblent toutes répondre à Open Source et sont toutes utilisables sous Linux.

LE GTK

La bibliothèque GTK (Graphic ToolKit) fait partie des packages disponibles aux développeurs pour la réalisation d'une interface utilisateur plus sophistiquée.

D'autre part, l'environnement graphique GNOME présent sur les distributions Linux utilise GTK+ pour son interface graphique. Pour ces raisons, Arnaud a décidé de faire des recherches sur GTK.

La glib est un ensemble de fonctions, parmi lesquelles on trouvera des fonctions de gestion mémoire, des chaînes de caractères ou des threads. Les primitives de dessin sont le point, le cercle, la droite. Le GTK permet de gérer les fenêtres (au sens X-Window du terme), les couleurs, le clavier, la souris....

Le GTK a été écrit en C ce qui est avantageux pour réaliser un défilement fluide. Cette librairie est assez portable (présente sur les distributions Linux et adaptable à Windows). Malheureusement, elle manque d'exemples de programmes.

De plus, en ce qui concerne la manipulation de ces fonctions, cela reste assez difficile, le nombre de paramètres à renvoyer est important. En effet, il faut pour chaque fenêtre par exemple renvoyer le contexte graphique, le contexte evenementiel, etc.. .Nous désirons une bibliothèque de plus haut niveau, plus simple à manipuler.

Par ces motifs, nous n'avons pas utilisé GTK pour développer notre jeu.

SDL

Nous nous sommes intéressé à SDL (Simple Direct Media) car de nombreux sites lui sont consacré. Pour cette raison, Jean-Claude a décidé de faire des recherches sur la librairie SDL.

SDL a pour champ d'utilisation principal la création de jeux en 3D. La librairie est codée en C/C++ et utilisable sous Linux.

Malgré la diversité des sites lui étant consacrés, il fût impossible de trouver une documentation détaillée en rapport avec son installation, ce qui nous empêcha d'achever sa mise en place. De plus, SDL semble plus adapté pour les developpements 3D.

Les tests ne furent pas exécutés sur machine, car nous avons pas reussi à installer cette librairie graphique.

EZWGL

L'étude de la bibliothèque graphique EZWGL (the EZ Widget and Graphic Library) fût confiée à Gwénaël.

EZWGL est utilisée principalement lié à la programmation graphique sous le système X Window. Elle a été testée sur de nombreuses plates-formes (SunSparc/X11R5, SGI-INDIGO/X11R5, Pentium- Linux/X11R6,...)

Cette bibliothèque est un ensemble de fonctions en C pour développer principalement l'interface utilisateur graphique (GUI) et écrire des programmes sous Madrake 8.0.

Les « gadjets » (widget en Anglais) actuellement mis en oeuvre sont très nombreux (environ 40). On peut y trouver le bouton et tous ses dérivés, des frames, des histogrammes etc..

Les inconvénients :

- Pour créer un bouton et son étiquette associée, on emploie une méthode composée d'un trop grand nombre d'arguments (7 pour afficher une phrase sur un font rouge!).
- Installation impossible (RPM défectueux).
- Cette bibliothèque ne sembple pas adaptée pour les jeux.

Aucun test ne fût réalisé.

EZWGL est trop axé sur la GUI.

Allegro

Cette librairie fût confiée à Thomas, elle est importante pour nous parce que tout d'abord, au cours de nos recherches internet nous avons souvent rencontré des liens vers son site hébergeur. De plus, comme toutes les librairies étudiées juqu'à maintenant ne nous ayant pas satisfaites et que Allegro avait été spontanément choisie par l'autre équipe de projet, nous nous sommes alors vivement intéressés à ses possibilités.

Description :

Allegro est une librairie graphique en C et en assembleur destinée à l 'écriture de jeux vidéo et autre types de programmes multimédia. Allegro est un acronyme récursif: Allegro Low LEvel Game ROutine. « Routine de bas niveau pour les jeux »

Cette librairie fonctionne sous différentes plateforme:

-Unix (Linux,FreeBSD,Solaris,Iris)
-Windows (MSVC, MinGW,Bordland,Cygwin)
-Dos (DJGPP, Watcom)
-BeOS
-MacOS(alpha)
-QNX(alpha)

Cependant il faut faire quelques modifications de code pour passer d'une plateforme à une autre.

Fonctions graphiques:

-Possède toutes les routines de dessins(pixel, ligne, rectangle, etc..).
-Pour les sprites, Allegro permet de faire différentes animations et supporte différents formats tels que les BMP,PCX et autres formats par librairies d'extension.
-Manipulation de palette(lecture, écriture, conversion, fondu), et la conversion du format de couleur RGB<->HSL.
-On peut faire des dessins directement sur l'écran ou sur des images-mémoire de grande taille.
-Allegro permet de faire un scrolling hardware et triple buffer et des écrans en mode X splittés.

Drivers graphiques:

Pour Unix :
Allegro utilise X-Windows, DGA, fbcon, SVGAlib, VGA, mode-X, VBE/AF.
Pour Windows;
Direct-X(plein écran et fenêtré), GDI.

Fonctions mathématiques:

Allegro permet d'utiliser des fonctions mathématiques d'arithmétiques et trigonométriques. Elle possède les tables de trigonométrie précalculées, et permet la manipulation de matrices et de vecteurs 2D et 3D.

Lorsque l'on utilise Allegro sous Linux on peut compiler le code avec gcc. Lors de la compilation du code, Allegro configure le programme selon la machine en utilisant le fichier allegro.cfg. Si rien n'est défini dans allegro.cfg, Allegro va remplir lui-même ce fichier grâce aux méthodes d'auto-détection qui lisent les paramètres réglés par Xfree86.

Avantages :

-gère le clavier, la souris et le joystick.
-Allegro possède des méthodes bas niveau ce qui permet d'avoir un code rapide à l'exécution.
-Conçu pour les jeux vidéo.
-Peu de difficultés pour trouver de la documentation sur Allegro.
-On trouve facilement des codes sources de jeu fait avec Allegro sur internet.

Inconvénients :

-Les réglages graphiques sont différents selon les machines.
-La portabilité du code entre différents systèmes d'exploitation n'est pas parfaite.
-Pour l'utilisation de certaines méthodes il est précisé dans la documentation de posséder des versions de logiciel spécifiques.

Exemple pour l'utilisation de l'accélération graphique matérielle, il faut utiliser la méthode GFX_XDVA2 mais il faut la version de XFREE86 4.0.0.

Nous avons choisi la librairie graphique Allegro 4.0.0 car elle répond à tous les besoins définis dans le cahier des charges et au critère qualité aussi. Avec cette librairie graphique tous les outils dont nous avions besoin pour implementer le projet de ces modules étaient fournis. Nous voulons faire un scrolling fluide et l'atout principal de cette librairie, c'est la facilité d'utilisation et la rapidité d'exécution de ses méthodes.

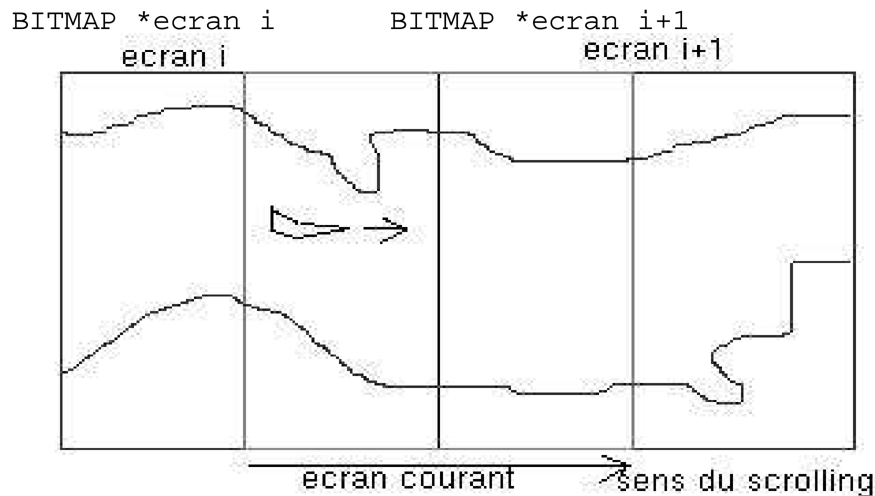
IV. Le composant de base d'Allegro : le bitmap et la tuile

1. Le Bitmap (BITMAP*)

Un niveau de jeu se décompose en plusieurs écrans. Nous avons besoin pour réaliser le scrolling de connaître l'état mémoire de l'écran courant i à afficher et de l'écran suivant i+1.

Il faut donc créer une structure de données de type BITMAP* car ce type est compatible avec toutes les fonctions d'Allegro. Cette structure représentera d'abord toutes les images à afficher sur un écran de niveau. Pendant le cycle principal, seulement deux de ces structures seront chargées

en mémoire vive pour effectuer l'affichage et rendre l'effet de défilement.



La structure d'image graphique BITMAP qu'offre Allegro est la suivante :

```
typedef struct BITMAP
{
 int w,h                    //longueur et largeur de l'image en pixels
 int clip                   //booléen d'activation du mode clipping
 int cl,cr,ct,cb            //coordonnées du rectangle « clip »
 int seg                    //segment mémoire
 unsigned char* line[]              //ligne de pixel de l'image
}
```

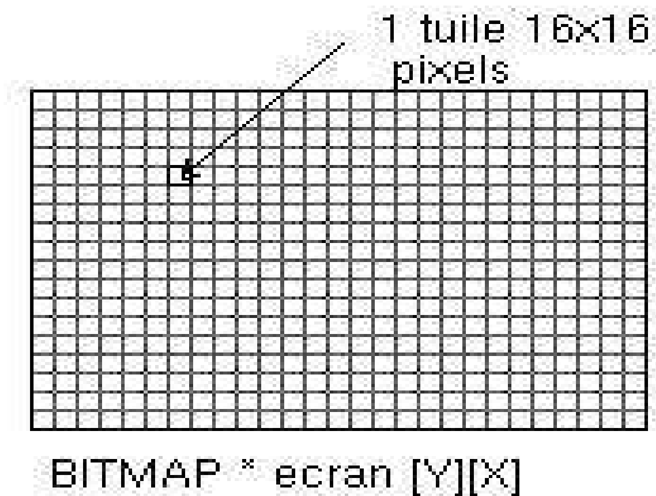
Nous pourrions utiliser alors un BITMAP* écran i et un BITMAP* écran i+1 puis réaliser le scrolling avec ces deux BITMAP de dimension de l'écran (ex:640x480). Mais cela a un inconvénient majeur, il faudrait redessiner BITMAP* écran i de 640x480 à chaque modification de l'image, par exemple, suite à l'explosion d'une petite partie du décor.

Cette conception n'est même pas envisageable, nous voulons pouvoir redessinner que quelques tuiles de l'écran s'il le faut. Mais aussi, en un moment donné et sans subir de ralentissement, pouvoir toutes les redessinner.

2. La tuile ou « tile » (BITMAP* [Y_TILE][X_TILE])

Aprés analyse de codes source de jeux existants, nous avons remarqué que la solution existe : une conception par tuiles. L'écran est composé en fait d'un quadrillage de petite tuiles de taille standard 16x16 pixels. L'objet tuile etant aussi un BITMAP*, chaque tuile peut donc stocker une image.

Ainsi notre probléme est résolu : il suffit simplement de redessinner la tuile modifiée par l'explosion.



Une tuile ou tile en anglais sera notre élément graphique de base. Le schéma ci-dessus représente un écran graphique par exemple en 640x480 composé de tuile de 16x16.

X= 640 / 16 = 40
Y= 480 / 16 = 30
écran=XxY=40X30=1200 tuiles.

Par ailleurs, il faut distinguer les éléments de natures différentes qui composent une image d'écran du jeu. En effet, nous aimerions qu'une méthode unique soit compatible avec chaque type d'élément.

Ainsi, nous fixons que les éléments graphiques d'un certain type appartiennent à une couche. Nous distinguons cinq types de couches au total par écran graphique.

Par la suite, toutes les couches seront appliquées les unes sur les autres en utilisant la méthodes de « blitage » suivantes:

copier/coller : blit()

copier/coller masqué (par la couleur « transparente ») : masked_blit()

Ces méthodes élémentaires de blit sont très rapides et simples à manipuler, aussi essayerons nous d'utiliser un maximun de ces méthodes.

Ces opérations se feront sur des BITMAP* variables internes de travail, et des BITMAP* intermédiaires ou buffer. Ces buffers seront ensuite « blités » à l'écran (extern BITMAP* screen) d'Allegro.

L'affichage final n'est donc qu'un empilement de couches « blitée » à l'écran. Ceci est rendu possible grâce au fait qu'un bitmap lors de sa création mémoire est initialisé par défaut à la couleur blanche (entier de valeur 0 en profondeur de couleur 8 bit), l'élément neutre (c'est la coulruer « transparente ») qui permet les actions de « blitage ».

En ajoutant 3 dimensions à notre tableau de BITMAP*, voici notre structure de données en couches qui à elle seule représente tout les éléments graphiques d'un écran de jeu.

3. La structure en couches : l'écran logique

BITMAP* ecran [Nombre_de_couches] [Y] [X]

- couche 0 : Objets
  Les Objets sont tous les éléments graphiques qui sont immobiles à l'écran et dont nous n'aurons donc pas à (re)calculer les déplacements. Cela implique aussi qu'ils ne seront pas redessinés en général.

- couche 1 : Sprites
  Les Sprites sont tous les éléments graphiques qui sont mobiles à l'écran et dont il faudra (re)calculer les déplacements et par conséquent seront redessinés plus souvent.

- couche 2 : Propriétés
  Cette couche n'est pas une couche graphique. Elle paraît être un peu intruse dans cette structure mais, c'est une couche qui permet de coder des états logiques pour chaque tuile. Les états logiques seront codés grâce à la couleur de certains pixels et à leur emplacement dans une tuile. Le but est de coder des états que l'on peut changer à la survenue d'un ou plusieurs événements.

  Pour chaque couche, on a pour l'instant besoin de :
  est_DESTRUCTIBLE,est_PASSABLE en lecture seule
  à_DESSINER,est_DEPASSE en lecture/ecriture

Exemples d'utilisation:

IMPLEMENTER LES REGLES/CONTRAINTES DE JEU

- La tuile de la couche objet est elle passable par des sprites, si non il y aura collision.

- La tuile de la couche objet est elle destructible par des sprites, si oui penser à la redessinner.
- La tuile de la couche objet est elle à dessiner à l'écran.
- etc, etc...

CONTROLE EVENMENTIEL

Les déclencheurs permettent de spécifier un déclenchement d'action du jeu lors de sa traversée.

- La tuile représente le point de départ/fin du niveau.
- La tuile a la propriété d'accélerer la vitesse d'un sprite la traversant
- etc, etc...

Le nombre de propriétés codables se calcule pour une tuile de 16x16 (se décomposant de 256 pixels) ainsi : Mettons que l'on fixe que chaque ligne de pixels de cette tuile code les propriétés des tuiles de chaque couche, on peut coder 9 propriétés de couches différentes. De plus, si on code la propriété selon la couleur de ce pixel, par exemple avec une profondeur de couleur de 8 bits, on obtient déjà 255 propriétés d'action/événements différentes pour 1 propriété de couche !

- couche 3 : Fond
  Cette couche représente le fond ou l'univers graphique dans lequel les objets et sprites évoluent par exemple dans l'espace.

- couche 4 : Fond2
  Cette couche facultative est reservée pour un usage ultérieur.

Les éléments logiques subissent plus de manipulation de lecture/écriture que les éléments graphiques. Par ailleurs, ils peuvent avoir un « lien de parenté » (héritage) étroit les liant, ainsi pouvoir appliquer sans distinction des méthodes à la structure mère comme aux structures filles est attrayant. Il semble donc qu'une implémentation avec une structure de données en C++ soit préférable.

Les éléments graphiques s'ils subissent moins de modification que les éléments logiques, il n'en reste pas moins qu'ils sont l'objet d'un traitement de copier/coller sélectif pour avoir une charge mémoire controlée dont nous pourrons surveiller l'ampleur tout au long du développement des modules.

V. Le scrolling et la gestion des sprites

1. Le Scrolling

Le scrolling est l'élément principal du déroulement du jeu. Appelé aussi défilement, ce système permet de faire défiler à l'écran les différentes couches que nous avons définies auparavant. Tous les événements du jeu doivent intervenir pendant le jeu et donc pendant le scrolling. Grâce à notre système en couches, il devient aisé de faire défiler les couches à l'écran sans pour autant modifier l'emplacement des tuiles en mémoire.

Notre scrolling consiste donc en un défilement graphique puisque les tuiles contenues en mémoire gardent leur position et doivent être remises à jour seulement lors de la survenue d'un événement.

a) Etude de l'existant

Une recherche sur différents codes sources à été réalisée. Une analyse a été faite ce qui a conduit à faire plusieurs remarques. En général, dans les jeux de type 'shoot them up' on rencontre deux types de scrolling. Le premier système le plus simple génère un décor de fond aléatoirement. Le deuxième type élabore des cartes de tuiles qui determinent le niveau chargé à l'avance.

b) Réalisation

D'une part nous génèrerons aléatoirement des étoiles pour le fond sur notre couche fond. D'autre part, nous répartirons des tuiles sur notre couche objet. Enfin le scrolling fait défiler les couches Objet et Fond.

Le but de notre jeu est de mettre l'accent sur la fluidité, par conséquent d'assurer un défilement continu de notre scrolling. Nous avons donc conçu trois types de scrolling pour

expérimenter la continuité du défilement.

Prenons comme exemple un niveau de 10 écrans. Nous voulons faire défiler ces dix écrans les uns à la suite des autres. Plusieurs solutions s'offrent à nous :

- Scrolling de tout le niveau.

  Le plus simple à priori c'est de construire une seule grande bitmap de la taille de nos 10 écrans et de n'en faire afficher qu'une partie, en la décalant au fur et à mesure.

- Scrolling en chargeant les 10 écrans pendant le scrolling.

  Il est possible de « découper » le niveau en plusieurs bitmaps. Par exemple une bitmap de 640X480 pour chaque écran et il suffira de « charger » ces bitmaps au fur et à mesure.

- Scrolling des écrans déjà chargés en mémoire

  Cette dernière méthode reprend la précédente sauf que la mémoire pour les bitmaps est déjà allouée, ce qui est plus rapide car il n'y a aucun chargement pendant le scrolling.

Chaque scrolling a ses caratéristiques....

Pour le premier, le problème se situe au niveau de la fluidité car faire afficher à chaque fois une partie de la map de 6400X480 pixels entraine un ralentissement dû à la taille de la bitmap.

Pour le second, la fluidité est bonne mais le temps passé au chargement des bitmaps pendant le scrolling peut engendrer des saccades. Dans certains cas, ces saccades sont légèrement perceptibles.

Pour le dernier, apparement aucun problème, le seul inconvénient est qu'il faut allouer de la mémoire à l'avance prporotionnelle à la taille des bitmaps pour que l'exécution se fasse avec une bonne optimisation.

c) Fonctionnement

Notre scrolling est constitué des différents éléments suivants:

- Des couches (fond, objet).
- Des buffers pour le décalage (scrolling).

Les couches

L'image de fond est une bitmap sur laquelle nous avons choisi de dessiner aléatoirement des petits cercles pleins pour représenter les étoiles. Cette image est ensuite chargée dans la couche fond.

Les tuiles de décor sont générées à des emplacements aléatoires dans la couche objet. Les tuiles des sprites(vaisseaux) sont générées elles aussi de façon aléatoire dans la couche sprite.

Les buffers

Les couches sont superposées dans les buffers, ce qui permet alors de les décaler pour créer l'effet de scrolling. En fait, nous nous servons de trois buffers: le premier pour la partie droite, un second pour la partie gauche et un buffer final qui se compose des deux parties.

Les deux buffers sont affichés selon la position courante dans le buffer final. Ce dernier est affiché puis effacé à chaque pas du scrolling.

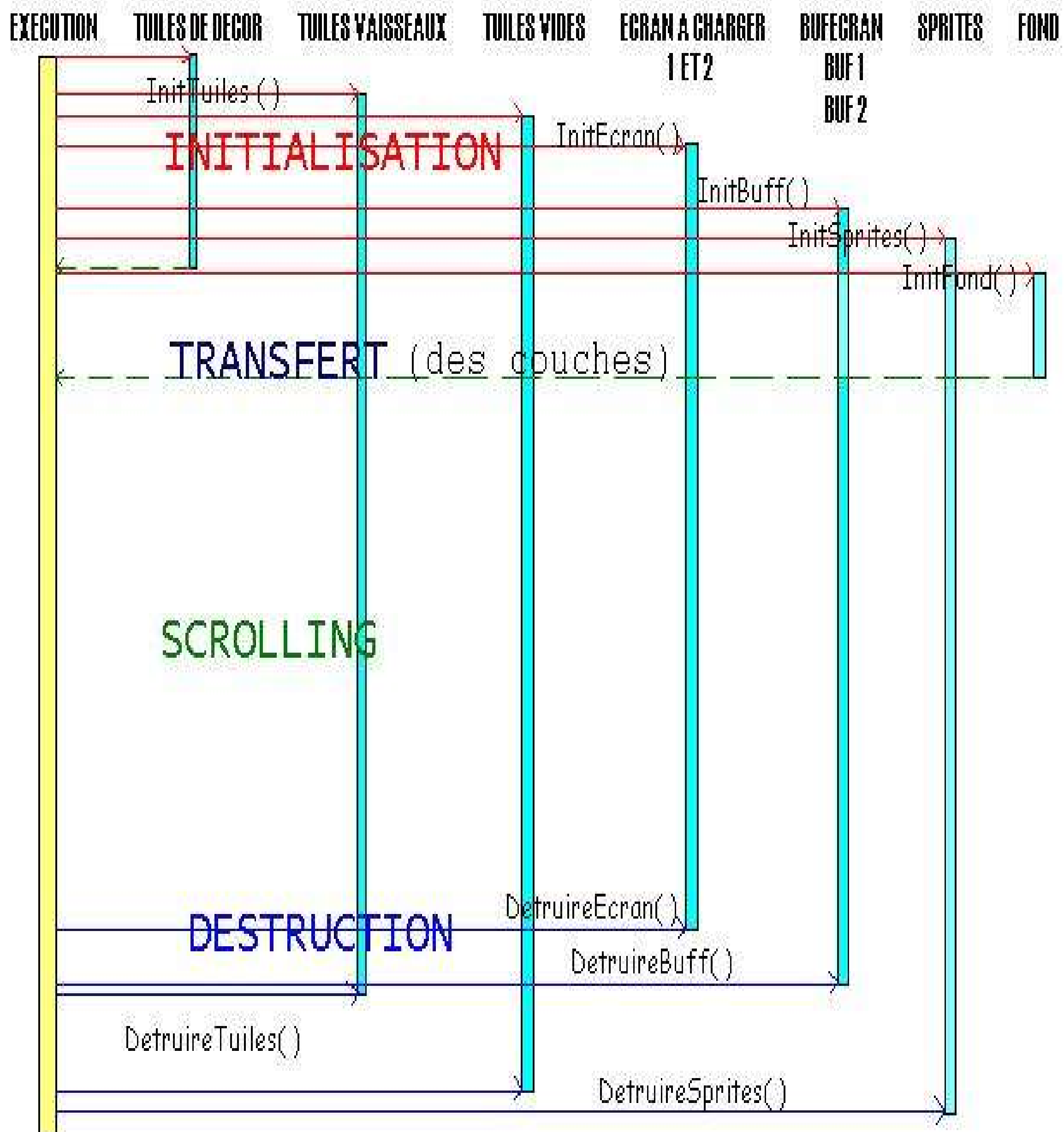
Le scrolling

Nous avons choisi de faire défiler les couches objet et fond mais la couche sprite ne défile pas. En effet, les Sprites doivent naviguer librement. Donc, lors du scrolling, on décale l'affichage des couches du décor et du fond et on décale en mémoire les sprites (objets dynamiques).

Le résultat du décalage des couches et du rafraîchissement des sprites est mémorisé dans le buffer final puis affiché à l'écran. Tous ces choix ont pour conséquence de produire un scrolling fluide. A chaque pas de ce scrolling, toutes les modifications se font en mémoire mais un seul affichage est exécuté: celui du buffer final sur l'écran.

Ce qu'on voit à l'écran et qui donne l'impression d'un niveau qui défile n'est que l'assemblage de plusieurs couches sur lesquelles les tuiles ne se décalent pas à la même vitesse.

Voici le diagramme des séquences lors de l'execution du module de Scrolling :



2. La Gestion des Sprites et le Calcul de leur Coordonnées

La gestion des sprites est active dans la boucle du scrolling, et comme nous avons pu le constater dans le diagramme ci-dessus, est initialisé dans la méthode Init_Sprite() et enfin détruite. L'objet C++ Sprite a été défini ainsi :

```
classe Sprite
{

        private:

        int x1, y1;                 //Coordonnées  dans le repére de tuiles
                                              x :[0 ... X]  et  y :[0 ... Y]
        int numeroDeCamp;           //Camp d'appartenance
        char etat;                  //Indique le comportement courant
        int bouclier;               //Points de protection
        int puissanceReacteur;      //Coefficient de déplacement
        int puissanceDeFeu;         //Coefficient de dommage
        int Arme;                   //Type d'arme

        int capacitex               //Points de capacité des moteurs arrière
        int capacitey               //Point  de capacité des moteurs latéraux
        int maxcapacitex            //Poussée maximale arrière
        int maxcapacitey            //Poussée maximale latérale

        float reelx,reely;          //Coordonnées dans le repére de pixels
                                             x :[0 ... SCR_X]  et  y :[0 ... SCR_Y]
}
```

Un double repère est necessaire pour réaliser le lien entre l'emplacement logique du sprite (coordonnée logique) et son emplacement graphique. La conversion se résume a une simple division par les dimensions de la tuile (Y_TILE, X_TILE).

Seulement, nous avons remarqué un bug lors des essais, les sprites « sortent » de l'écran en bas et à droite. En fait, lorsqu'une coordonnée du sprites avoisine SCR_X ou SCR_Y, c'est à dire la résolution de l'écran, le sprite disparaît, il n'est plus « blité » à la tuile correspondante. L'origine de ce
bug est localisé dans la fonction RaffraichirCoordOrig(int). Il doit s'agir d'une erreur d'annulation ou de troncature lors du passage d'un repère à l'autre que nous avons malheureusement pas eu le temps de régler.

Voici la boucle de scrolling pendant laquelle sont raffraîchis les sprites:

```
while ((bord-PAS < SCR_X) && ((sort!=27)))
{               //Affichage des couches objet et fond
                masked_blit(ecr1,bufEcran,bord,0,0,0,(SCR_X)-bord,SCR_Y);
                masked_blit(ecr2,bufEcran,0,0,(SCR_X)-bord,0,bord,SCR_Y);

                //Mise à jour des sprites
                        sort = SaisieClavier();
                for (i=0;i<numSprite;i++)
                {
                    RaffraichirCoordDest(sort, i);
                    RaffraichirCoordOrig(i);

                }

                //Affichage de l'ensemble des couches a l'ecran
                blit(bufEcran,screen,0,0,0,0,SCR_X,SCR_Y);
                clear_bitmap(bufEcran);
                rest(1);
                bord=bord+PAS;
}
```

Les coordonnées de destination sont calculées selon le déplacement possible du sprite :

cx, cy est la variable *capacité du moteur* et *frottement* représente la force de résistance à la poussée des moteurs dans un certain milieu. Le *frottement* est défini pour un niveau.

Si le joueur à joué en appuyant sur une touche de déplacement :

```
cx = cx - puissanceReacteur  -  frottement;
cy = cy + puissanceReacteur + frottement;
```

Si le joueur n'a pas joué :

```
Si ( cx < 0 ) cx+= frottement;
Si ( cx > 0 ) cx -= frottement;
Si ( cy < 0 ) cy+= frottement;
Si ( cy > 0 ) cy-= frottement;
```

Si le joueur joue il fait donc évoluer la capacité de son moteur; s'il se déplace vers l'avant le frottement le contraint; s'il se déplace vers l'arrière il inverse la poussée et le frottement dû à la vitesse du scrolling l'aide (cela permet des dégagement rapides trés utiles parfois !). Si le joueur ne joue pas, la poussée est réglée automatiquement à hauteur de la vitesse du scrolling (cx =0 & cy =0).

VI. La GUI (Graphic User Interface)

C'est une interface graphique mise en place dans le but de faciliter la communication entre le joueur en tant qu'utilisateur et la machine exécutant le jeu.

1. L'historique de la conception de la GUI

Nous avons consacré une attention particulière au développement de la GUI, nous sommes passé par plusieurs versions avant de réaliser une GUI finale, qui est plus intuitive pour l'utilisateur.

Dans un premier temps, on a fait une GUI classique en mode texte. On a numéroté les choix que le système offrait au joueur. Pour valider l'utilisateur doit appuyer sur la touche qui correspond au numéro de son choix.

Dans un second temps, nous est venue l'idée d'améliorer notre GUI en adaptant l'utilisation des images BITMAP pour construire une interface purement graphique. On a donc eu recours au logiciel The Gimp et KdePaint afin de dessiner diverses images dont nous avions besoin.

Enfin, après la réalisation de notre GUI graphique, on a décidé de la perfectionner, en la rendant modulaire afin de pouvoir fabriquer en même temps le sous-menu options.

2. La structure de la GUI

La GUI contient plusieurs menus, à titre d'exemple le menu principal qui est le plus important au point de vue communication avec l'utilisateur, ce menu se déclenche au début du lancement du jeu, et aussi juste après la fin de la partie pour savoir si le joueur a envie de recommencer une nouvelle partie ou de quitter le jeu.

Menu principal du jeu

Nous avons mis en place un sous-menu OPTIONS dans notre menu principal. En le

validant, le joueur pourra paramétrer les options de jeu, car un autre menu d'options sera affiché à la place du menu principal. Un écran de score a été aussi implémenté.

Menu Option

Menu Score

3. Les problèmes rencontrés et leurs solutions

Durant le développement de la partie GUI, nous avons rencontré quelques problèmes qui ont parfois bloqué l'avancement du codage. Citons par exemple : le problème de l'animation de notre interface, et aussi celui de la gestion des scores des joueurs.

Tout d'abord le problème de l'animation consistait à trouver un outil pour créer celle-ci, tout en gérant les touches de directions, nous n'avons pas trouvé de structure d'animation tout faite proposée par la librairie Allegro.

De plus il a fallu gérer le passage d'une animation à une autre par les touches fléchées. Pour résoudre ce problème, nous avons utilisé au point de vue programmation la structure du tableau de type chaîne de caractères afin de mémoriser dans ses cases les noms de toutes les images dont nous avons besoin.

Avec cette structure , il suffit d'appeler le tableau utilisé en désignant le numéro de la case de l'image à charger, ce qui déclenche systématiquement l'appel de celle-ci. Alors que l'animation est affichée en analysant la pression d'une touche de directions, et ce, en chargeant les images convenables.

En ce qui concerne la gestion des scores, le problème était de pouvoir charger/sauver le fichier des scores des joueurs c'est à dire leur noms ainsi que leurs scores, et finalement afficher à chaque fois les six premiers joueurs classées par ordre décroissant de score.

Pour résoudre ce problème, nous a utilisé la structure FILE en langage C pour pouvoir construire un fichier texte et consulter ces données et effectuer la mise à jour si nécessaire tout en effectuant le tri des scores de ces joueurs. Puis on affiche le fichier ainsi ordonné sur l'écran si l'option SCORES est sélectionnée.

VII. Le bilan

1. « Ce qu'on a fait »

Le scrolling réalisé est fluide et rapide. Le jeu fonctionne de façon opérationnelle jusqu'en résolution 1024x780 en profondeur de couleur 8 bits. Le code est portable sur un système de type Windows (voire autre), nous n'avons utilisé que les fonctions de bases d'Allegro et du C/C++ compatibles entre systèmes hétérogénes. Les sprites se déplacent indépendamment du scrolling et ont un comportement commun initialisé au hasard. Le décor (couche objet) est lui aussi généré au hasard.

2. « Ce qu'on a pas fait »

La gestion des collisions qui aurait du être implémentée pour faire sortir de la course les concurrents qui se percutent et qui pour l'instant s'annulent (disparaît graphiquement).

La gestion des fichiers de données .DAT (datafile) pour charger effectivement l'écran i+2 lorque l'écran i est dépassé par le traceur (bord) et pour sauvegarder des écrans qui auraient composés les niveaux du jeu.

3. « Ce qu'on aurait voulu faire en plus et qui est faisable»

Tout d'abord, nous aurions voulu rendre chaque couche indépendante liable les unes aux autres par une table de correspondance. Ainsi le scrolling ferai défiler indépendamment chaque couche selon une vitesse donnée ce qui permettrait des effets visuels de chocs et autres...

Ensuite, nous voulions aussi implémenter dans la couche propriéte, la gestion des tuiles déclencheuses d'actions incluse dans le scrolling. Pour ce faire, il fallait définir dans ces tuiles (« au but non-graphique ») des propriétés d'interactions entre couches. Ensuite, nous coderions le départ et l'arrivée du niveau, plutôt que de forcer l'arrêt utilisateur via la touche « escape », ainsi que des effets d'accélération de scrolling, de zone d'écran « dangereuses » et d'autres de « bonus ».

Enfin, nous aurions aimé initialiser le gestionnaire de sprites et de décor autrement qu'au hasard. C'est à dire, pour les sprites, une interface (classe amie) d'intelligence artificielle simple qui aurait permis de faire des concurrents pour la course qu'est XjAC, et pour les décors un interpréteur (classe amie) qui lirait la carte du niveau en cours de chargement depuis le disque dur.

Conclusion

Ce projet ambitieux qu'est XjAC n'a pas pu être implémenté totalement, mais il nous aura permis d'avoir une compréhension désormais bien plus nette de Linux, comme système d'exploitation certes, mais avant tout comme d' un système de développemment libre. Et nous savons désormais que le développemment de jeu graphique ou de programme ludique, se révéle être un métier où l'expérience est primordiale au moins autant que la créativité. Le critére Open Source contraint à une rigueur et un respect de l'existant et des normes essentielles, pour qu'à chaque développement s'ajoutant à un programme Open Source, la transparence et la clareté de la réalisation soit toujours garantie. C'est dans cette optique que nous avons réalisé XjAC avec la volonté de comprendre l'origine et l'utilisation des ressources que nous mettions en oeuvre. C'est aussi la raison de l'astreinte à notre objectif de qualité pour permettre, à un autre projet de s'inspirer de nos travaux ce qui permettra, peut-être, de continuer à faire évoluer le logiciel libre.