Licence 3 informatique — C avancé
Université de Marne la Vallée — Année 2015-2016

Projet de programmation C : Shoot 'em up

L'objectif de ce projet est de développer une application graphique d'un jeu d'avion de type Shoot'em up (descendez-les tous) en temps réel et donc en gérant le taux de rafraîchissement de l'écran F.P.S. (Frames per second).

Capture d'écran du Chromium-bsu (Shoot'em up 2D de vaisseau)

Description et déroulement du programme

Outre quelques possibles écrans d'accueil (complètement optionnels), une fois le jeu lancé, une fenêtre graphique s'ouvre. Sur cette fenêtre, un avion (ou équivalent) doit apparaître, le joueur peut diriger cet avion avec les touches directionnelles du clavier. Votre avion doit disposer d'une arme de base qui tire en tir tendu uni-directionnel qui devra être le sens de défilement du décor du jeu. L'arme de base devra être actionnée via l'appuie prolongé sur la touche de tir ou bien en appuyant par à-coups (la touche control gauche du clavier ou bien la touche espace constitue un bon choix pour une touche de tir).

Les ennemis devront apparaître toujours sur le même bord (le bord vers lequel sont dirigés les tirs de l'avion). Les ennemis doivent défiler progressivement vers le bord opposé de l'écran. Le joueur doit tirer sur les ennemis afin de les empêcher d'atteindre l'autre bord. Les ennemis peuvent aussi tirer. Lorsque qu'un tir ennemi touche l'avion du joueur ou bien lorsqu'un ennemi traverse l'écran ou encore si un ennemi touche l'avion du joueur, alors le joueur perd une quantité de vie. Lorsque sa quantité de vie arrive à zéro, la partie est perdue.

Gérer le rafraîchissement de l'écran

Pour gérer le taux de rafraîchissement, deux choses sont fondamentales :

- Votre programme ne demande pas trop de ressources.
- Vous limitez le nombre d'appels à la fonction `MLV_actualize_window`.

Votre fonction `main` ou bien une autre fonction gérant globalement votre jeu devra comporter une boucle maîtresse qui gère le jeu frame par frame. Tant que le joueur ne veut pas quitter, on affiche la frame, on résout tous les événements sur la frame, puis on calcule le temps passé et on ajuste la vitesse du jeu en attendant quelques millisecondes.

```
1   /* Main loop over the frames... */
2   while(!quit){
3     /* Get the time in nanosecond at the frame begining */
4     clock_gettime(CLOCK_REALTIME, &last);
5     /* Display of the current frame */
6     /* THIS FUNCTION CALL A SINGLE TIME MLV_actualize_window */
7     draw_window(&param, &grid);
8
9     /* We get here at most one keyboard event each frame */
10    event = MLV_get_event(&key_sym, NULL, NULL, NULL, NULL,
11                          NULL, NULL, NULL, &state);
12
13    /* Event resolution here.... */
14    ...
15    /* Moves of the entities on the board */
16    ...
17    /* Collision resolutions */
18    ...
19
20    /* Get the time in nanosecond at the frame ending */
21    clock_gettime(CLOCK_REALTIME, &new);
22    /* We compute here the time spent for the current frame */
23    accum = (new.tv_sec-last.tv_sec)+((new.tv_nsec-last.tv_nsec)/BILLION);
24
25    /* We force here to wait if the frame was too short */
26    if(accum < (1.0/48.0)){
27      MLV_wait_milliseconds((int)(((1.0/48.0) - accum)*1000));
28    }
29  }
```

Donner l'impression de vitesse, le défilement

Deux options paraissent ici raisonnables : soit on a une image de fond que l'on fait glisser, soit on fait glisser des éléments de décor dans le sens du défilement. Un problème possible avec l'image de fond est que le jeu peut parfois devenir lent (en particulier si l'image est lourde (`jpeg` de 1 Méga) et que la fenêtre graphique est grande). Faire glisser des éléments de décor demande par contre plus de développement.

Un prototype réalisé par votre enseignant fait défiler des étoiles brillantes sur un fond noir, étoiles de différentes tailles et à différentes vitesses. Ce qui suit constitue un choix possible (Il y a mieux, il y a aussi moins bien...).

```
1 #define MAX_STAR 300
2
3 typedef struct stars{
4   int x[MAX_STAR];
5   int y[MAX_STAR];
6   int size[MAX_STAR];
7   int speed[MAX_STAR];
8   int active[MAX_STAR];
```

```
9     int r[MAX_STAR];
10    int g[MAX_STAR];
11    int b[MAX_STAR];
12  }Stars;
```

Au début du jeu, les étoiles sont inactives. Toutes les frames, en utilisant la fonction `rand` de `stdlib.h`, on lance un dé (avec `rand() % TRUC_BIEN_CHOISI`) et si ça fait zéro, on active une nouvelle étoile sur le bon bord de l'écran. La solution proposée plus haut n'utilise pas d'allocation dynamique mais le jeu ne peut pas supporter plus de 300 étoiles simultanément. Nous vous laissons imaginer que 300 peut être correct si `TRUC_BIEN_CHOISI` à bien été choisi.

Gérer les collisions

Pour gérer les collisions, les éléments du jeu doivent avoir une taille et le jeu doit connaître ces tailles. Pour les spécialistes des jeux vidéos, on parle de hitbox. A chaque frame, le jeu doit identifier les éléments en collisions et résoudre ces collisions. Soyez raisonnables dans les choix d'action. Normalement, les missiles ennemis disparaissent et la vie du joueur diminue quand un missile ennemi atteint le vaisseau du joueur. Les éléments quittant l'écran doivent aussi disparaître (avec potentiellement des appels à `free` si vous ne voulez pas que `valgrind` récupère une partie des points de votre projet (quel salaud ce `valgrind`)). Le joueur ne peut pas quitter l'écran, il faut prévoir d'ignorer ses déplacements qui le ferai quitter l'écran.

Les missiles joueurs et ennemis ne peuvent pas entrer en collision mais il faut les gérer par couple à l'aide de double boucle `for` imbriquées.

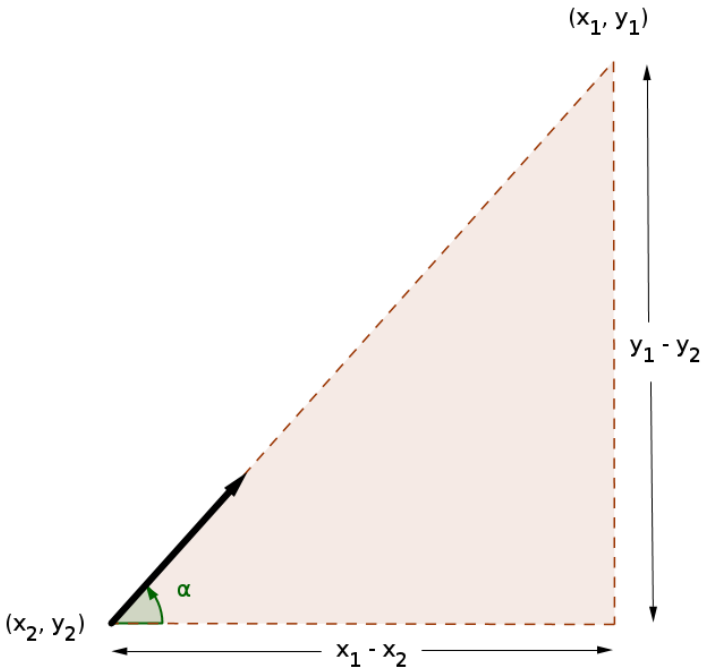
collisions missile joueur VS ennemis
collisions missile ennemis VS vaisseau joueur
collisions ennemis VS joueur ennemis

```
1  for(i=0 ; i<nb_ammo_player ; i++){
2    for(j=0 ; j<nb_enemies ; j++){
3      if (distance(... , ...) < ...){
4        /* We resolve here the collision
5           between a player missile and an ennemy */
6        x = ... ;
7        y = ... ;
8        diaplay_explosion_at_point(..., ..., x, y);
9        ...
10     }
11   }
12 }
```

Viser une cible

Viser c'est dur! C'est comme ça... Il faut faire quelques mathématiques et un peu de trigonométrie. Dans une extrême gentillesse, voici un peu comment on vise dans un programme C ou l'interface graphique est géré pixel par pixel.

Soit un vaisseau joueur en position (x_1, y_1) et un ennemi en position (x_2, y_2). L'ennemi veut faire un tir en ligne droite vers le joueur. L'objectif en maintenant de récupérer l'angle de tir α en radian pour que l'ennemi vise le joueur.



On commence par prendre le triangle rectangle passant par nos deux points et donc les cotés perpendiculaires sont deux à deux parallèles aux axes verticaux et horizontaux. Dans ce triangle rectangle, on connaît la mesure des deux cotés adjacent à l'angle droit. Le coté horizontal a pour mesure la différence des deux abscisses de nos points alors que le coté vertical a pour mesure la différence des ordonnées.

On peut alors exprimer la tangente de l'angle α qui permet d'orienter le canon de l'ennemi vers le joueur. Par définition, la tangente d'un angle dans un triangle rectangle, c'est la mesure du coté opposé (ici la partie verticale) divisée par la mesure du coté adjacent (ici la partie horizontale). Au final

$$tan(\alpha) = \frac{y_1 - y_2}{x_1 - x_2}$$

et donc

$$\alpha = arctan(\frac{y_1 - y_2}{x_1 - x_2})$$

En C, dans la bibliothèque `<math,h>`, on peut croiser une fonction qui calcule l'arctangente d'un angle (man 3 atan). Cette fonction retourne une valeur comprise entre $-\frac{\pi}{2}$ et $\frac{\pi}{2}$, de ce fait, suivant que l'ennemi est gauche ou à droite de l'écran, il faut corriger la valeur avec $+\pi$ pour ne pas viser à l'opposé exact d'où se trouve le joueur mais bien sur lui.

Interface graphique

L'interface graphique est une très grosse partie du projet. Cette dernière devra être développée avec la bibliothèque graphique C de l'université : la libMLV. Une large documentation à propos de cette bibliothèque est accessible à l'URL :
`http://www-igm.univ-mlv.fr/~boussica/mlv/`
Les machines de l'université en sont équipées.

Ce choix de la libMLV est une obligation. Il y a mieux, il y a aussi moins bien... On attend ici de vous que vous vous familiarisez avec cette bibliothèque écrite par autrui, et que via sa documentation seulement, vous puissiez construire une véritable application graphique.

Modularité

Votre projet doit organiser ses sources sémantiquement. Votre code devra être organisé en modules rassemblant les fonctionnalités traitant d'un même domaine. Un module sans entêtes pour le `main`, un module pour la représentation du jeu et un module graphique semble être un minimum. La partie graphique devrait être très grosse et on peut même l'imaginer subdivisée au besoin (module pour les ennemis, module pour les explosions si vous en faites...).

Votre projet devra être compilable via un `Makefile` qui exploite la compilation séparée. Chaque module sera compilé indépendamment et seulement à la fin, une dernière règle de compilation devra assembler votre exécutable à partir des fichiers objets en faisant appel au linker. Le flags -lMLV est normalement réservé aux modules graphiques ainsi qu'a l'assemblage de l'exécutable (sinon, c'est que votre interface graphique dégouline de partout et que vos sources sont mal modulées).

Pour aller plus loin

Pour les plus récalcitrants d'entre vous, toute amélioration sera considérée comme un plus pour l'évaluation. Voici quelques suggestions possibles de raffinements pour ce projet. Ces propositions sont gradués avec un indice de difficulté entre parenthèse.

- (enfant) Calculer un nombre de point durant la partie (durée ou le joueur reste en vie, nombre d'ennemis tués, ...) et rajouter un écran d'accueil et un tableau des high scores.
- (enfant) Faire un interface avancé qui affiche la barre de vie, le nombre de point de la partie courante, la durée de vie du joueur depuis le début.
- (grand enfant) Le joueur déplace son vaisseau avec de l'inertie (quand on va longtemps à gauche, on va plus vite à gauche).
- (très grand enfant) Gérer des chocs élastiques lors des collisions vaisseaux ennemis. Quand le joueur touche un ennemi, il perd de la vie un peu mais il est aussi projeté dans le sens inverse de son choc (demande d'avoir gérer l'inertie, un choc élastique devient alors un vecteur vitesse inversé par rapport au point de collision).
- (grand enfant) Certains des ennemis savent viser.
- (grand enfant) Certains ennemis envoie des missiles à têtes chercheuses.
- (grand enfant) Inclure des explosions frame par frame dans votre interface graphique.
- (grand enfant) Implémenter des ennemis qui enchaîne une séquence de déplacement (par exemple, les ennemis suivent une courbe sinusoïdale, ...).
- (adulte confirmé : Ubisoft n'a qu'à bien se tenir) Faire un gestionnaire de niveau. Au lieu de générer aléatoirement un seul niveau au kilomètre, votre jeu est capable de charger un fichier qui contient des descriptions d'ennemis ainsi que des séquences d'apparition d'ennemis frame par frame.
- (jeune adulte mais adulte quand même) Faire un boss. Un boss est un ennemi qui a plus de vie que les autres ennemis de base. Il reste de manière pérenne à l'écran juste qu'à sa destruction par le joueur. Lorsqu'il est à l'écran, il enchaîne différentes séquences d'attaques mais toujours dans le même ordre (tire en face de lui 2 secondes (le joueur doit se planquer...), puis envoi deux missiles en ligne droite vers le joueur (le joueur doit les éviter), puis envoie un missile à tête chercheuse mais qui se déplace lentement

vers le joueur jusqu'à qu'il n'ait plus de carburant (ici il faut tourner autour du boss en attendant la mort du missile), ...etc...).

Conditions de développement

Le but de ce projet est moins de pondre du code que de développer le plus proprement possible. C'est pourquoi vous développerez ce projet en utilisant un système de gestion de versions Subversion. Le serveur à disposition des étudiants de l'Université est disponible à cette adresse : `https://reposetud.univ-mlv.fr/`. Comme nous attendons de vous que vous le fassiez sérieusement, votre rendu devra contenir un dump du fichier de logs des opérations effectuées sur votre projet, extrait depuis le serveur de gestion de versions ; afin que nous puissions nous assurer que vous avez bien développé par petites touches successives et propres (commits bien commentés), et non pas avec un seul commit du résultat la veille du rendu. L'évaluation tient compte de la capacité du groupe à se diviser équitablement le travail.

Remarques importantes :

- L'intégralité de votre application doit être développée exclusivement en langage C (la documentation technique dynamique fait évidemment exception). Toute utilisation de code dans un autre langage (y compris C++) vaudra 0 pour l'intégralité du projet concerné.
- En dehors des bibliothèques standards du langage C et de la libMLV, il est interdit d'utiliser du code externe : vous devrez tout coder vous-même.
  Toute utilisation de code non développé par vous-même vaudra 0 pour l'intégralité du projet concerné.
- Tout code commun à plusieurs projets vaudra 0 pour l'intégralité des projets concernés.

Conditions de rendu :

Vous travaillerez en binôme et vous lirez avec attention la Charte des Projets. Il faudra rendre au final une archive `tar.gz` de tout votre projet (tout le contenu de votre projet `svn`), les sources de votre application et ses moyens de compilation. Il sera alors crucial de lire des recommandations et conseils d'utilisation de `svn` sur la plate-forme `moodle` (à venir). Vous devrez aussi donner des droits d'accès à votre chargé de TD et de cours à votre projet via l'interface `redmine`.

Un exécutable devra alors être produit à partir de vos sources à l'aide d'un `Makefile`. Naturellement, toutes les options que vous proposerez (ne serait-ce que –help) devront être gérées avec `getopt` et getopt_long.

La cible `clean` doit fonctionner correctement. Les sources doivent être propres, dans la langue de votre choix, et commentées. C'est bien de se mettre un peu à l'anglais si possible.

Votre archive devra aussi contenir :

- Un fichier `log_dev` correspondant au dump des logs de votre projet (nom des commits, qui? et quand?), extrait depuis le serveur de gestion de versions que vous aurez utilisé.

- Un fichier `makefile` contenant les règles de compilation pour votre application ainsi que tout autre petit bout de code nécessitant compilation (comme les tests par exemple).
- Un dossier `doc` contenant la documentation technique de votre projet ainsi qu'un fichier `rapport.pdf` contenant votre rapport qui devra décrire votre travail. Si votre projet ne fonctionne pas complètement, vous devrez en décrire les bugs.
- Un dossier `src` contenant les sources de votre application.
- Un dossier `include` contenant tous les headers de vos différents modules.
- Un dossier `bin` contenant à la fois les fichiers objets générés par la compilation séparée.
- Aucun fichier polluant du type `bla.c~` ou `.%smurf.h%` généré par les éditeurs. Votre dossier doit être propre !
- Si possible, la partie html générée par l'utilitaire `doxygen` à partir de votre application de visualisation. Ceci est optionnel mais tellement plus propre.

Sachant que de nombreux vilains robots vont analyser et corriger votre rendu avant l'oeil humain, le non respect des précédentes règles peuvent rapidement avorter la correction de votre projet. Le respect du format des données est une des choses des plus critiques.