Langages et Génie Logiciel
Projet

2009—2010

Le but du projet est de produire un jeu vidéo de type "shoot'em up". Le joueur contrôle un vaisseau spatial en vue de dessus. Il doit anéantir les vagues de vaisseaux adverses à l'aide d'armes futuristes, que l'étudiant non-violent pourra éventuellement remplacer par des lapins de l'espace.

1 Travail demandé

Vous devez implémenter le jeu en OCaml. Vous n'avez pas à dessiner les graphismes, qui vous sont fournis. Vous devez vous concentrer sur la structure de votre projet en utilisant les notions vues en cours, notamment :

- en utilisant des modules répartis en plusieurs fichiers ;
- en choisissant avec soin les types de vos structures de données ;
- en utilisant à bon escient la notion d'abstraction ;
- en *testant* le plus régulièrement possible.

La note de votre projet en prendra largement compte.

Vous commencerez par implémenter la base du jeu présentée dans la section 3. Ensuite, vous pourrez implémenter les autres caractéristiques présentées dans la section 4.

Vous devrez rendre votre projet par mail[1] sous la forme d'une archive compressée `nom1-nom2.tar.gz` comprenant elle-même un dossier `nom1-nom2` contenant les fichiers nécessaires pour compiler, ainsi qu'un `Makefile`. La commande `make` doit compiler votre projet sans erreur et sans warning.

2 Base fournie

Les fichiers suivants vous sont fournis[2] :

- les images du jeu ;
- le module `Draw` contenant des fonctions pour dessiner le décor, les personnages et les missiles ;
- le module `Input` permettant de gérer les entrées clavier et souris.

Vous pouvez modifier ces fichiers à votre convenance.

Pour compiler votre projet vous aurez besoin d'inclure les librairies `bigarray`, `sdl`, `sdlloader` et `sdlttf`.

Pour fermer la fenêtre du jeu, vous devez utiliser la fonction `Sdl.quit`, de type `unit -> unit`.

Le module `Input` lance l'exception `Pervasives.Exit` si l'utilisateur ferme la fenêtre de l'application ou s'il tape Ctrl+C dans le terminal.

La documentation des modules `Draw` et `Input` se trouve dans leurs interfaces `draw.mli` et `input.mli`.

3 Jeu de base

Le but de cette partie est d'avoir une base solide à partir de laquelle le jeu pourra évoluer. Le joueur devra pouvoir déplacer son vaisseau et tirer des missiles. Il devra aussi pouvoir quitter le jeu à tout moment à l'aide de la touche Echap (Esc).

Cette base du jeu fera l'objet d'un rendu intermédiaire le 12/03/10.

[1] `bardou@lri.fr, Christine.Paulin@lri.fr`

[2] `http://www.lri.fr/~paulin/LGL/base.tar.gz`

Déroulement du temps L'état du jeu contient les informations telles que la position du vaisseau et le score du joueur. Toutes les Δt secondes, où Δt est une durée bien choisie telle que $0.01s$, l'état du jeu est mis à jour. Il est ensuite affiché à l'écran. Le passage de l'état courant à l'état suivant sera appelé une *étape* du jeu.

Pour mesurer le temps on pourra utiliser la fonction `Sdltimer.get_ticks`, de type `unit -> int`. Celle-ci mesure le nombre de millisecondes écoulées depuis une date arbitraire choisie par le système. Pour bloquer le jeu quelques instants on pourra utiliser la fonction `Sdltimer.delay`, de type `int -> unit`, qui prend en entrée le nombre de millisecondes à attendre.

Dessin Le module `Draw` suppose un axe des abscisses X qui va de gauche à droite, le 0 correspondant à la limite du décor à gauche. L'axe des ordonnées Y va de haut en bas, le 0 correspondant à la limite du décor en haut. Notez que dans la base fournie, le module `Draw` construit une fenêtre de taille 800×600. Cela signifie que le dernier pixel visible a pour coordonnées 799×599.

Gestion des objets physiques Tous les objets du jeu (vaisseau du joueur, vaisseaux adverses et missiles) doivent avoir des déplacement convaincants. Rappelons l'équation physique exprimant l'accélération $\vec{a}$ en fonction des forces appliquées et de la masse :

$$m \times \vec{a} = \sum \vec{F}$$

L'accélération est la dérivée de la vitesse $\vec{V}$:

$$m \times \frac{\mathrm{d}\vec{V}}{\mathrm{d}t} = \sum \vec{F}$$

On discrétise cette équation :

$$m \times \frac{\Delta\vec{V}}{\Delta t} = \sum \vec{F}$$

ce qui se réécrit en :

$$\vec{V}' = \vec{V} + K \times \sum \vec{F} \tag{1}$$

où K est une constante. De plus, la vitesse est elle-même la dérivée de la position $\vec{P}$, et en discrétisant on obtient :

$$\vec{P}' = \vec{P} + K' \times \vec{V} \tag{2}$$

Pour avoir des mouvements convaincants il suffira donc de stocker, pour chaque objet physique, sa position $\vec{P}$ et sa vitesse $\vec{V}$ et de les mettre à jour à chaque étape à l'aide des équations 1 et 2.

Par exemple, lorsque le joueur appuie sur la flèche de gauche, une méthode naïve serait de modifier directement $\vec{P}$ en $\vec{P} - (1, 0)$, ce qui revient à affecter $(-1, 0)$ à la vitesse. Mais pour avoir des mouvements plus réalistes vous pouvez aussi voir cette commande comme l'application d'une force $\vec{F} = (-1, 0)$ et appliquer l'équation 1 en affectant $\vec{V} + (-1, 0)$ à $\vec{V}$. A vous de choisir $\vec{F}$ en fonction de ce que vous voulez obtenir comme mouvement.

Génie logiciel Il est suggéré d'implémenter un module `Vector`, avec un type abstrait `Vector.t` représentant les vecteurs et des opérations courantes comme l'addition de vecteurs ou le produit scalaire. De même, il pourra être pertinent de factoriser le code permettant de gérer les objets physiques.

4 Suppléments

Il s'agit maintenant d'étendre le jeu pour le rendre plus intéressant. Vous n'avez pas besoin de tout ajouter, le plus important est que ce que vous choisissez soit implémenté de façon intelligente. Il vous est néanmoins demandé d'implémenter les vaisseaux adverses.

Vaisseaux adverses Des adversaires doivent apparaître à partir des bords de l'écran, se déplacer sur l'écran, puis être supprimés lorsqu'ils sortent de l'écran. Les adversaires pourront apparaître par vagues, sous diverses formations. Ils pourront éventuellement tirer eux-mêmes des missiles visant le joueur. Les mouvements des vaisseaux adverses peuvent être variés : ligne droite, zig-zag, recherche du joueur, etc.

Collisions et points de vie (bouclier) Le joueur a un nombre de points de vie affiché sur l'écran. Ce nombre diminue si le personnage entre en collision avec un adversaire. S'il arrive à 0, le joueur a perdu. Les vaisseaux adverses pourront aussi avoir des points de vie. Il faudra aussi tester les collisions entre les missiles et les vaisseaux.

Armes multiples Vous pouvez donner la possibilité au joueur de choisir entre plusieurs armes. Celles-ci pourront envoyer des missiles dans des directions différentes, faire plus de dégats, être plus rapides, se fragmenter en plusieurs missiles...

Bonus à ramasser Vous pouvez ajouter des bonus que le joueur peut ramasser pour améliorer ses armes, restaurer ses points de vie, ...

Effets spéciaux La base fournie contient le dessin de quatre étapes d'animation d'une explosion, que vous pouvez utiliser lorsqu'un vaisseau est détruit.

Niveaux Le jeu pourra comporter plusieurs niveaux.

Munitions Vous pouvez ajouter une jauge de munitions limitant le nombre de missiles que le joueur peut tirer. Vous pourrez ajouter la possibilité de ramasser des munitions sous forme de bonus.

Scores Vous pourrez maintenir et afficher un score pour le joueur. Celui-ci peut augmenter à chaque vaisseau adverse détruit, et diminuer si le joueur utilise trop de missiles ou subit des dégats. Vous pourrez maintenir une liste des meilleurs scores, enregistrée sur le disque dur.

Défilement vertical Vous pourrez faire défiler les étoiles du fond pour donner une impression de vitesse.

Autres idées Vous pouvez implémenter d'autres idées mais rappelez-vous que le plus important est d'implémenter chaque nouvelle possibilité d'une façon intelligente et modulaire. La qualité du code est plus importante que la quantité d'idées implémentées.