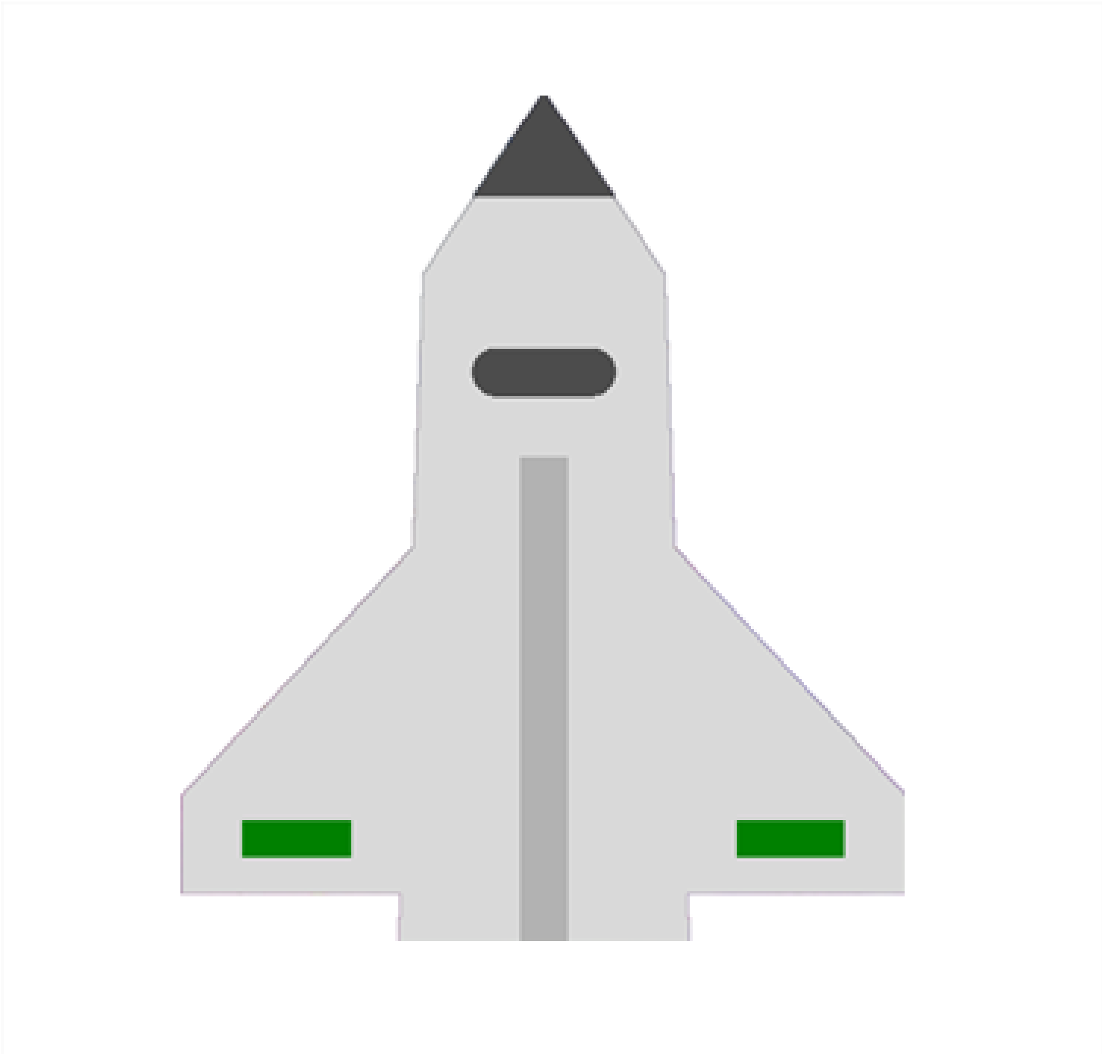
ALTAIR

Projet de synthèse d'images – OpenGL/SDL – IMAC 1

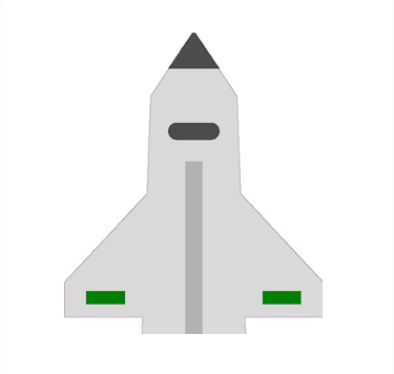
Alexandre REHO – Héloïse ROUSSEL



Mai 2016

I – LA LISTE DE NOS TRAVAUX

- La physique de l'hovercraft (du vaisseau spatial) : CODÉE ET FONCTIONNE
- Le pilotage au clavier : CODÉ ET FONCTIONNE
- La modelisation du terrain et des checkpoints : CODÉE ET FONCTIONNE
- Les collisions avec les checkpoints : CODÉES ET FONCTIONNENT
- La lecture des élements du terrain sur un .txt : CODÉE ET FONCTIONNE
- La gestion des bords : CODÉE ET FONCTIONNE
- Le zoom sur la map : CODÉE À MOITIÉ ET FONCTIONNE À MOITIÉ
- Le chargement du terrain en lecture, avec des codes de terrain : NON CODÉ
- Le menu pour choisir la difficulté et le parcours : CODÉ ET FONCTIONNE
- Le son avec SDL_Mixer : CODÉ ET FONCTIONNE
- L'identité visuelle du jeu : DESSINÉE ET FONCTIONNE (on espère)
- Le contrôle par manette : NON CODÉ
- L'IA : NON CODÉE



II – NOTRE PROJET

LE VAISSEAU

Pour commencer, nous nous sommes penchés sur la physique du vaisseau. Nous nous sommes réunis avec plusieurs autres groupes pour discuter de la meilleure manière de représenter un véhicule qui se déplace de manière réaliste, comme s'il était sur l'eau sans friction, ou sur la glace. Nous avons donc opté pour la manière suivante : recoder les structures et les fonctions Vector3D et Point3D du TP sur le raytracer, en les transformant en Vector2D et

Point2D, qu'on appliquera sur les mouvements du vaisseau (nous avons pris l'habitude d'écrire vaisseau au lieu d'hovercraft car nous voulions décaler l'univers du jeu mais la physique reste exactement la même). Tout simplement, à chaque frame du programme, nous mettons à jour la position, la vitesse, et l'accélération du vaisseau en fonction de son angle et de sa vitesse. De plus, on lui ajoute une déccélération représentant une force de friction. Une fonction va s'occuper d'appeler toutes les fonctions de mises à jour juste avant le dessin du vaisseau. Quant au vaisseau lui-même, nous nous sommes basés sur de vrais véhicules volants de type supersoniques. De plus, nous lui avons ajouté des flammes de propulsion qui ne s'affichent que quand on accélère, ainsi que ses diodes arrières qui deviennent rouges (vertes quand on n'accélère pas).

Au départ, le vaisseau est à la position (0 ; 0), de vecteur direction (0 ; 1) (direction verticale), et de vitesse et accélération égales à 0. Les champs de la structure du vaisseau permettent de savoir directement si le vaisseau est en train d'accélérer ou non, de tourner ou non, ainsi que de récupérer son orientation, sa position, sa vitesse et son accélération.

Si nous accélérons, nous appliquons au vecteur accélération un dixième du vecteur direction (il pourra être modifié en fonction de la vitesse que l'on souhaite), et zéro sinon. Tout simplement. Pour la mise à jour de la vitesse, nous rajoutons la valeur du vecteur accélération au vecteur vitesse, puis nous soustrayons à la vitesse un cinquantième de sa valeur (c'est la friction). Quant à la position, nous incrémentons à la valeur en x de la position, la valeur en x de la vitesse, et idem pour y. Enfin, si le vaisseau tourne, nous lui incrémentons ou décrémentons 5 degrés de rotation, tout en modifiant le vecteur direction en fonction des cosinus ou sinus de l'angle.

Et voilà, nous avons la physique de notre vaisseau.

LE TERRAIN

Pour charger le terrain du jeu ou tout simplement n'importe quelle image dans notre jeu, nous avons utilisé les fonctions loadImage() et dessinTerrain() vues en TP.

Concernant les checkpoints, nous avons choisi de mettre en place une liste chaînée de checkpoints que l'on remplit à la lecture d'une matrice .txt composée de 0 et de 1. La matrice est construite avec 80 lignes de 80 caractères. Notre programme ayant une définition de 800x800 pixels, chaque caractère de la matrice correspondra à un emplacement espacé de 10 pixels du suivant, et ainsi de suite. Si le programme rencontre un 1 au 7ème caractère de la 48ème ligne, il dessinera un checkpoint au point (70 ; 480).

Lorsque notre vaisseau rencontre un checkpoint, c'est-à-dire que la distance entre la position de notre vaisseau et la position du checkpoint est inférieure au rayon du checkpoint, ce dernier disparaît. Le jeu s'achèvera quand tous les checkpoints auront été passés.

Nous avons choisit une gestion des bords en accord avec notre thème qui est l'espace en les ignorant et en faisant un monde Torique, c'est-à-dire que, lorsque l'on arrive sur un bord de la fenêtre on réaparaît ensuite sur celui opposé.

Pour le zoom, nous avons préféré ne pas en faire car cela dévalorisait trop notre idée d'espace et de grandeur. Nous avons tout de même réussi à l'implémenter grâce à la fonction « lookat » d'openGL.



LES CHECKPOINTS

Nos checkpoints sont représentés par des petits points bleu clair, le but étant d'aller dessus afin de les récupérer. Nous avions le choix entre les rentrer dans une matrice depuis notre document texte ou d'en faire des objets depuis des liste chaînées. Nous avons donc préféré les liste chaînées puisque leur utilisation nous paraissait plus logique. Ainsi, nous avons créé deux structures,

l'une pour les checkpoints contenant leur position et le fait qu'ils soient actifs ou non et une autre pour avoir une liste de checkpoints.

L'utilisation de cette chaîne s'est ensuite faite logiquement par des fonctions comme passage_checkpoint() ou dessin_checkpoint().

LES ASTÉROÏDES

Parce qu'un jeu n'est pas drôle s'il est trop facile, nous avons décidé de mettre en place des astéroïdes qui, s'ils touchent le vaisseau, mettent fin à la partie. Ils ont été implémentés avec un squelette assez semblable à celui des checkpoints puisque leur utilisation se fait également par listes chaînées. Ces astéroïdes ont une vitesse ainsi qu'une direction aléatoire. Leur gestion des bords est identique à celle du vaisseau et lorsque l'opération se fait, leur vecteur de vitesse change. Il sera alors nécessaire d'éviter les astéroïdes afin de récupérer les checkpoints et ainsi terminer le jeu. Le nombre d'astéroïde sera variable selon le niveau de difficulté qu'on aura choisi. Si un joueur clique directement sur JOUER, il sera directement dirigé vers la map « Magellan 31 », avec le niveau facile « Astronaute du dimanche ».

LES OPTIONS

Dans nos options, nous avons choisi de mettre en place deux choses : le choix de la map ainsi que le choix de la difficulté. La position des checkpoints, quant à elle, sera déterminée aléatoirement entre plusieurs matrices .txt.

LE SON

Parlons maintenant du son. Pour souligner l'atmosphère spatiale de notre programme, nous avons choisi de passer en fond la bande son du jeu Portal 2. Pour cela, nous avons utilisé le canal « musique » (.mp3) de la bibliothèque SDL_Mixer. Aussi, au cas où cette bibliothèque ne serait pas installée sur les PC de l'université, nous vous invitons à profiter pleinement de notre programme en installant les paquets suivants : *libsdl-mixer1.2* et *libsdl-mixer1.2-dev*.

Pour encore enrichir notre programme, nous avons utilisé l'autre partie de la bibliothèque SDL_Mixer, c'est-à-dire la partie des canaux sonores. Contrairement au canal « musique » unique, SDL_Mixer propose de nombreux canaux « sonores » (.wav) que nous pouvons utiliser comme bon nous semble. C'est donc pourquoi nous avons allouer 2 canaux sonores : un pour une simulation d'ouverture de sas au lancement du jeu et un pour le moteur du vaisseau (oui, il n'y a pas de bruit dans l'espace mais nous trouvions ça cool). Un petit mixage de volume pour chacun des canaux et libre à nous de les lancer directement depuis la boucle événementielle de notre programme.

L'ORGANISATION DU CODE

Notre projet est organisé au travers de l'arborescence suivante :

- src : les fichiers .c
- include : les fichiers .h
- obj : les .o
- bin : l'executable
- images
- musique (et sons)
- la racine : qui contient le Makefile ainsi que les matrices de checkpoints.

En ce qui concerne le main.c (appelé altair.c dans notre projet), nous avons organisé la gestion des différentes options grâce à des variables d'état. En effet, toute notre partie de dessin ainsi que toute la partie événementielle en SDL_MOUSEBUTTONDOWN sont coupées en plusieurs sous-parties correspondant chacune respectivement à une variable d'état, grâce à des conditions if(state). Au lancement du programme, par exemple, la variable state_home vaut 1, tandis que state_options et state_difficulty valent 0. C'est seulement au clic sur JOUER ou à la sortie de la rubrique OPTIONS que toutes les variables valent 0 que le jeu se lance. Pour le choix des options, nous avons décidé de couper horizontalement la fenêtre en trois parties égales. Il nous aura donc suffit de récupérer la position de la souris en y pour déterminer sur quelle option nous avons cliqué.

Voilà donc comment se présente notre programme d’hovercraft décalé dans un univers spatial.

LES DIFFICULTÉS RENCONTRÉES

Les difficultés que nous avons rencontrées concernent les déplacements du vaisseau ainsi que la programmation des checkpoints et des astéroïdes.

Pour le vaisseau, au début, nous n'avions pas du tout une implémentation faite à partir de vecteurs et de points position.

Aussi, il a été difficile d'implémenter les liste chainées et la lecture de fichier qui ne fonctionnait pas correctement.