Projet "Colmycraft"

Sommaire :

1. Introduction :

Dans le cadre du cours de Synthèse d'Images, nous avons travaillé sur le projet « Colmycraft ». L'objectif de ce projet est de réaliser un jeu vidéo avec un hovercraft contrôlé par l'utilisateur, qui doit passer par tous les checkpoints du terrain le plus rapidement possible.

Dans ce rapport, nous présenterons les éléments réalisés dans ce projet, qu'ils aient été demandés dans le sujet ou optionnels. Ensuite, nous détaillerons la démarche suivie et les problèmes rencontrés lors dans la réalisation du projet.

2. Liste des travaux :

Eléments demandés et codés qui fonctionnent

- Manipulation de l'aéroglisseur
- Création du terrain et des checkpoints, à partir d'un fichier texte
- Collision avec les checkpoints
- Gestion des bords de la carte
- Gestion de la caméra

Eléments demandés et codés qui ne fonctionnent pas

/

Eléments demandés mais non codés

/

Options codées qui fonctionnent

- Design
- Gestion du texte

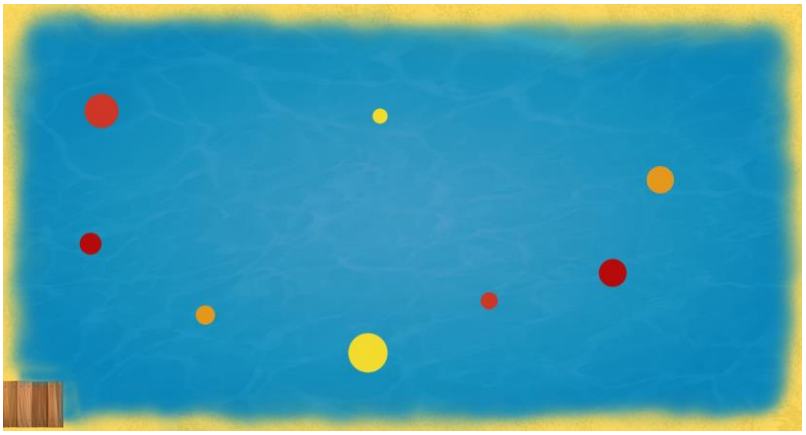
Options codées qui ne fonctionnent pas

- Son

3. Démarche et problèmes rencontrés :

Etape 1 : Création graphique

Nous avons commencé par réaliser le design du jeu, ce qui est pour nous un élément important. Nous avons donc commencé avec la maquette ci-contre :



Cette maquette nous permet d'avoir une idée de ce que nous voulons obtenir à la fin. Mais, pour le départ, nous avons mis d'autres checkpoints et uniquement une couleur de fond, afin d'effectuer des tests.

Etape 2 : Création des structures, des constructeurs et dessin des éléments

Ensuite, nous avons commencé à créer le jeu. Pour cela, la première partie a été la création d'un fichier contenant les informations du terrain et la récupération de celles-ci. Nous avons donc récupéré le nombre de checkpoints et les données de chacun, afin de les afficher dans la fenêtre, en créant des cercles de taille et de couleur différentes, définis par la structure suivante :

CHECKPOINT
entier rayon;
entier centreX;
entier centreY;
entier couleurR;
entier couleurV;
entier couleurB;
entier visible;

Le terrain lui-même est décrit par une structure, comme vous pouvez le voir ci-dessous :

TERRAIN
entier nbCheckPoints;
CheckPoint tableCheckPoints[100];

Nous considérons qu'il est plus simple de faire un tableau avec un nombre donné de checkpoint maximum, que de passer par de l'allocation dynamique. Nous ne pensons pas, d'ailleurs, faire plus de 100 checkpoints, d'où le nombre de « cases » maximum du tableau.

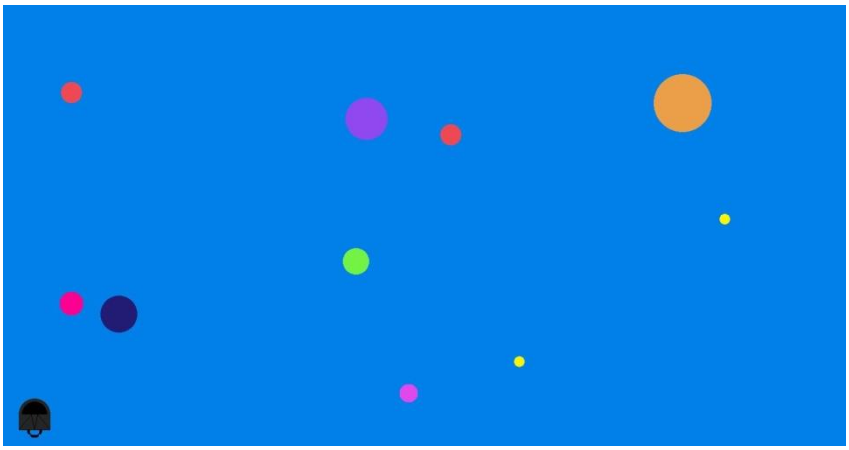
L'étape suivante a été la création de l'hovercraft unitaire, fait de plusieurs formes géométriques. Nous avons réalisé, pour cela, une structure hovercraft décrivant celui-ci et un hovercraft nommé « ColmyCraft » :

HOVERCRAFT
entier positionX;
entier positionY;

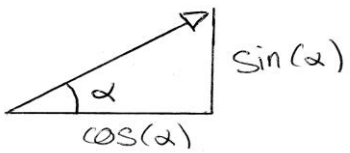
entier tailleX;
entier tailleY;
entier anglePosition;
entier angleMouvement;
réel vitesse;
réel acceleration;
réel deceleration;

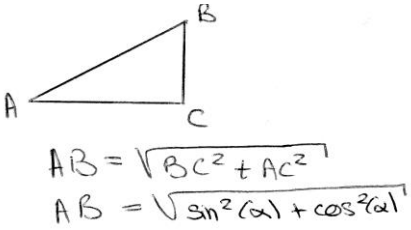
Comme leur nom l'indique, les premières variables correspondent à la position de l'hovercraft et à sa taille. Ensuite, nous avons l'angle de rotation de l'hovercraft. « angleMouvement » correspond à l'angle par lequel tourne l'hovercraft : celui-ci tournera toujours de plus ou moins cet angle. Nous avons, pour le mouvement, une variable vitesse pour le déplacement effectué par l'hovercraft, en pixels par image. L'accélération et la décélération sont des valeurs fixées d'augmentation et de diminution de la vitesse. Nous avons, d'ailleurs, une variable globale ACCELERATION_MAX.

Cet hovercraft est dessiné en réalisant des mises à l'échelle de tailleX et tailleY, une rotation de rotationPosition et un déplacement de positionX et positionY. Nous avons, aussi, ajouté des fonctions d'initialisation du terrain et des checkpoints et un constructeur à paramètres pour l'hovercraft, afin de lui donner une position de départ et une taille, choisies par l'utilisateur (pour pouvoir mettre plusieurs hovercrafts à des endroits différents de la carte si besoin).



Etape 3 : Mouvement de l'hovercraft



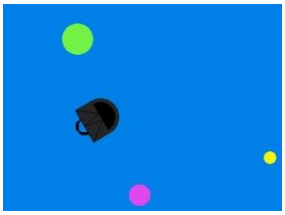


Nous avons ensuite voulu déplacer notre hovercraft. Pour tester, nous sommes parties sur un déplacement basique selon les quatre flèches du clavier. Nous avons, ensuite, réfléchi à la manière de gérer l'angle de rotation de l'hovercraft, pour pouvoir utiliser uniquement les flèches droites et gauches. La difficulté a été purement mathématique et physique : comment calculer la prochaine position de l'hovercraft ? La réponse est venue naturellement en dessinant le problème.

Nous avons donc réalisé ce pseudo-code et l'avons ensuite « traduit » :

FONCTION DEPLACER
entier positionX
entier positionY
entier tailleX

```
entier tailleY
entier anglePosition
entier angleMouvement
réel vitesse
SI clique_touche
        SI touche = flèche_droite ALORS
                anglePosition <- anglePosition – angleMouvement
        FIN SI
        SI touche = flèche_gauche ALORS
                anglePosition <- anglePosition + angleMouvement
        FIN SI
        positionX <- positionX + vitesse*sinus(anglePosition)
        positionY <- positionY + vitesse*cos(anglePosition)
        deplacement(positionX, positionY)
        ROTATION (anglePosition)
FIN SI
```



Nous avons donc un hovercraft qui se déplace d'un angle particulier, avec une vitesse fixe.

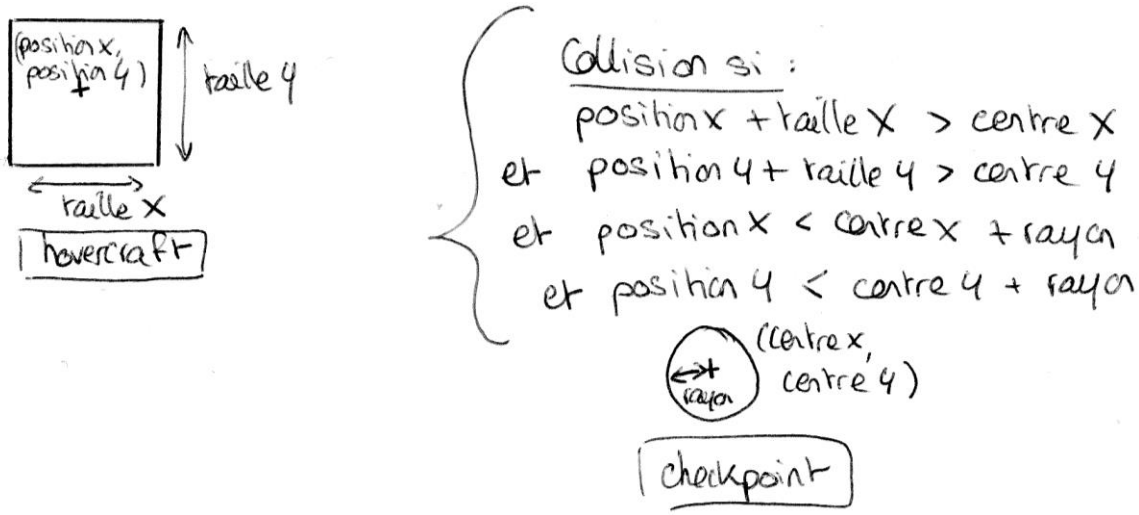
Notre objectif, ensuite, est d'ajouter une accélération et une décélération. Nous avons choisi de mettre une décélération constante, toujours présente, pour faire un effet de frottements. Nous avons donc ajouté ce pseudo-code à la fonction **DEPLACER HOVERCRAFT** :

```
réel accélération
réel décélération
réel acceleration_max
SI touche = flèche_haut ALORS
        SI vitesse < acceleration_max ALORS
                vitesse <- vitesse + acceleration
        FIN SI
FIN SI
SI vitesse > 0 ALORS
        vitesse <- vitesse – deceleration
FIN SI
```

Après plusieurs essais, nous avons trouvé des valeurs d'accélération et de décélération « jouables ». En effet, si la décélération est trop forte, l'utilisateur doit appuyer de nombreuses fois sur la flèche du haut pour avoir une vitesse correcte. Nous avons eu, à ce moment-là un second problème : l'hovercraft descendait et la vitesse ne restait pas à 0. Nous avons compris que le problème venait d'une comparaison par 0, alors que nous avons des réels, il fallait donc comparer avec sept zéros après la virgule (6 ne suffisant pas).

Etape 4 : Gestion des collisions

Pour les collisions, nous avons aussi réfléchi sur papier afin de se faire une idée de la démarche à suivre.



Après plusieurs essais, nous sommes arrivées à ce pseudo-code pour trouver si une collision a eu lieu :

FONCTION COLLISION
entier positionX
entier positionY
entier tailleX
entier tailleY
entier centreX
entier centreY
entier rayon
booléen collision
SI (positionY + tailleY >centreY) && (positionY < centreY + rayon) && (positionX + tailleX > centreX) && (positionX < centreX + rayon) **ALORS**
    collision = 1
SINON
    collision = 0
FIN SI
RETOURNE collision

Le principe présenté sur le pseudo-code est répété pour chaque checkpoint du terrain. Nous avons donc une boucle, parcourant chaque checkpoint du terrain, vérifiant pour chacun s'il a touché l'hovercraft : si oui, nous mettons la variable « visible » du checkpoint (initialement à 1) à 0. Dans la boucle d'affichage, nous ne dessinons donc que les checkpoints ayant la valeur « 1 » à ce paramètre. Visuellement, si nous avons une collision, le checkpoint concernant disparaît du terrain.

Etape 5 : Gestion des bords

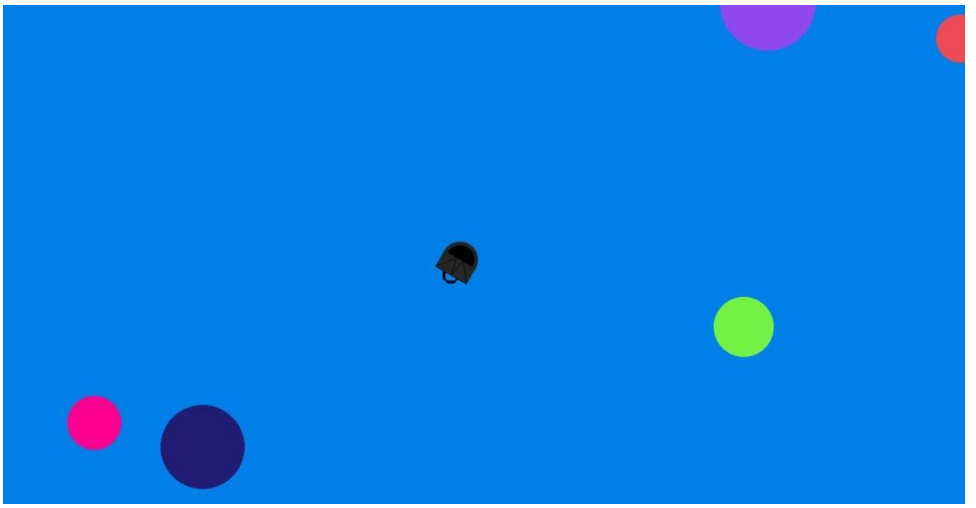
Pour gérer les bords de la carte, nous avons décidé, après avoir modifié la position de l'hovercraft, de vérifier si les positionX et positionY sont supérieures aux valeurs maximales de la carte ou inférieures à 0. Dans ce cas, nous replaçons l'hovercraft au bord. Par exemple, si l'hovercraft va en dessus de 0 dans l'axe X, on met sa positionX à 0 + tailleX, pour qu'il soit positionné correctement.

Etape 6 : Gestion de la caméra

Nous avons commencé par modifier la fenêtre pour quelle aille de 0 à Z00M pixels, en changeant les valeurs de « gluOrtho2D » de la fonction « reshape ». ZOOM étant une variable globale correspondant au nombre de pixels que nous voulons sur l'axe x. Nous avons donc un effet de « zoom » sur la carte. Nous avons aussi changé la taille de l'hovercraft (30px au lieu de 60) pour qu'il ne soit pas trop grand.

Maintenant, l'objectif est de « déplacer » la caméra en même temps que l'hovercraft. Pour cela, nous avons mis un « translatef », sans « pushMatrix » et « popMatrix », ce qui permet de déplacer la carte en fond. Pour commencer, nous avons mis les valeurs de translation en X et en Y par rapport à la position et à taille de l'hovercraft.

Ensuite, nous avons ajouté en X : (windowWidth - ZOOM)/2, et en Y : ((windowWidth - ZOOM)/2)/2, pour pouvoir placer l'hovercraft au centre de la carte. Pour pouvoir trouver ces valeurs nous avons eu quelques difficultés. En effet, nous n'arrivions pas à trouver comment placer l'hovercraft au centre, nous avons donc choisi des valeurs « au hasard », pour ensuite trouver le lien entre celles-ci, ZOOM et windowWidth.

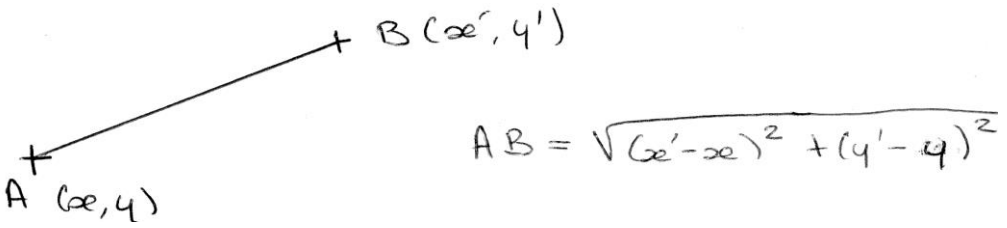


Etape 7 : Indication de l'emplacement des checkpoints

Le jeu étant actuellement jouable, nous avons décidé de l'améliorer en permettant à l'utilisateur de voir où sont les autres checkpoints. Pour cela, nous avons décidé d'ajouter une flèche vers le checkpoint le plus proche, la longueur de la flèche indiquant la distance de celui-ci avec l'hovercraft.

Nous avons commencé par ajouter un paramètre à l'hovercraft, qui est un pointeur sur CheckPoint nommé « prochainCheckpoint », correspondant au checkpoint le plus proche de l'hovercraft. Ensuite, nous avons initialisé celui-ci au premier checkpoint du terrain (pour éviter les erreurs de segmentation).

Nous avons dessiné le problème :

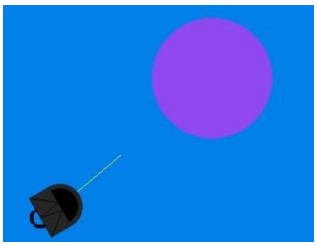


Pendant le dessin des checkpoints du terrain, nous avons mis cet algorithme :

POUR tous les checkpoints du terrain
    SI le checkpoint est visible **ALORS**

```
                SI prochainCheckpoint est visible ALORS
                        SI DISTANCE (hovercraft, checkpoint) < distance(hovercraft,
prochainCheckpoint) ALORS
                                prochainCheckpoint <- checkpoint
                        FIN SI
                SINON
                        prochainCheckpoint <- checkpoint
                FIN SI
        FIN SI
FIN POUR
DESSINERTRAIT (hovercraft, prochainCheckpoint)
```

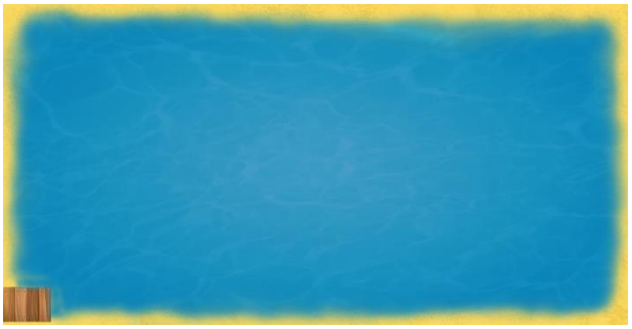
Nous vérifions si « prochainCheckpoint » est visible, car s'il ne l'est plus, il ne faut plus qu'une flèche se dirige vers lui.



Ensuite, nous avons dessiné un trait allant de l'hovercraft au centre des checkpoints. Mais, l'idéal étant une flèche, nous avons décidé de dessiner un trait allant de l'hovercraft à un point B, dans la direction du checkpoint, avec une longueur liée à la distance entre les deux points. Après avoir dessiné le problème, nous avons vu que le second point avait pour coordonnées : (hovercraftX + (checkpointX – hovercraftX) / λ, hovercraftY + (checkpointY – hovercraftY) / λ).

Nous avons choisi, après plusieurs essais, que le « meilleur » λ serait 2.5.

Etape 8 : Ajout d'une image de fond



Nous nous sommes dit qu'il serait intéressant d'ajouter une image de fond pour marquer les bords du terrain et avoir autre chose qu'un aplat de bleu pour signifier l'eau. Nous avons donc repris le TP d'OpenGL réalisé avec SDL_Image afin d'ajouter une image jpeg :

La difficulté ici a été d'adapter l'exercice fait sur un carré blanc à ce que nous avons actuellement. Nous avons donc modifié notre Makefile, ajouté tout ce qui était nécessaire dans notre projet, et mis la texture au niveau de notre terrain. Pour mieux intégrer notre texture, nous avons aussi modifié la couleur de fond, qu'elle corresponde à celles des « bords » de l'eau.

Etape 11 : Gestion du texte

Nous avons voulu améliorer le jeu en affichant constamment le score du joueur, ce score correspondant au nombre d'hovercraft touchés.

Pour cela, nous avons commencé par chercher comment ajouter du texte. Sur ce site : http://moodle.epfl.ch/pluginfile.php/1531756/mod_resource/content/1/Serie_17_19_complement_OPENGL_GLUT_texte.pdf, nous avons trouvé une fonction permettant de réaliser cela. Pour que la fonction puisse fonctionner, nous avons ajouté GLUT à notre projet, à travers une nouvelle inclusion, et donc une

modification de notre Makefile. Cette fonction permet, en passant en argument un texte, sa police, sa taille et son emplacement, d'afficher celui-ci sur le terrain. Nous avons donc pu afficher, dans la boucle, le score. Nous avons donc réussi à afficher un texte.

Pour afficher le score, nous ajoutons une variable correspondant au nombre de checkpoints non visibles : entier checkPointsNonVisibles. Dans la boucle d'affichage, si le checkpoint courant n'est pas visible, nous avons voulu ajouter 1 à cette dernière valeur. Mais, cette boucle étant appelée 24 fois par seconde, nous ne pouvons pas avoir un score correct. Nous avons donc ajouté un paramètre « teste » aux checkpoints indiquant si nous avons déjà ajouté 1 à checkPointsNonVisibles, et si ce n'est pas le cas on n'augmente pas cette variable. Nous avons donc le nombre de checkpoints déjà « touchés » par l'hovercraft qui sont bien récupérés et nous pouvons les afficher.

Pour placer le score en haut à gauche, nous avons cherché qu'elles étaient les meilleures valeurs, par rapport à la position de l'hovercraft, pour que le texte soit où nous voulions qu'il soit. Nous avons donc trouvé qu'il fallait enlever 350 en X et ajouter 175 en Y

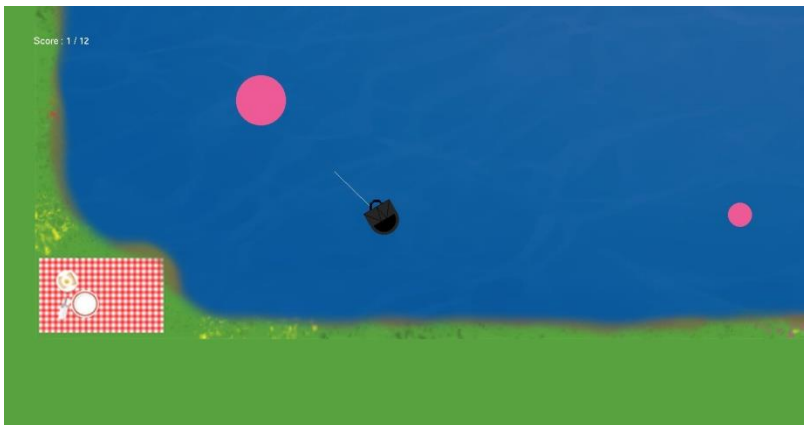
Nous avons ensuite ajouté une fonction « aGagné ». Appelée à chaque image, celle-ci parcourt tous les checkpoints et retourne 1 si aucun n'est visible et 0 si au moins un des checkpoints est visible. Si la fonction retourne 1, on affiche donc « Vous avez gagné ! » au-dessus de l'hovercraft.

Etape 10 : Gestion du son

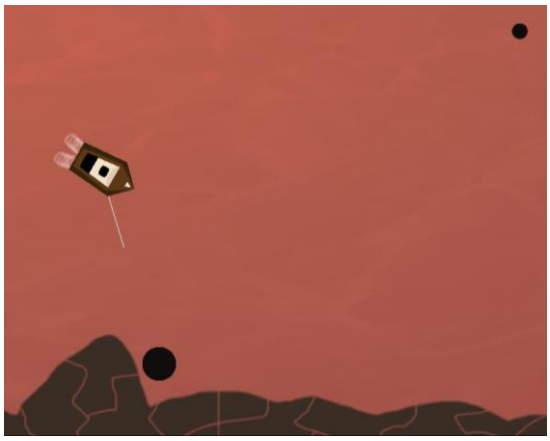
Nous voulions ensuite ajouter du son lorsque l'hovercraft accélère et lorsqu'il « touche » un checkpoint. Pour cela, nous avons commencé par lire des tutoriels pour apprendre à utiliser SDL_sound. Nous nous sommes inspirées du code présent ici : https://gist.github.com/armornick/3447121. Malheureusement, en l'adaptant à notre jeu, nous n'avons pas réussi à le faire fonctionner correctement, le problème étant au niveau du chargement du son, sûrement.

Etape 11 : Finalisations

Pour améliorer le design du jeu, nous avons réalisé trois terrains (niveau1, niveau2 et niveau3.txt), liés à trois images de fond différentes. Le fond « plage » utilisé jusqu'à maintenant, correspond au niveau 1 et voici le rendu avec le niveau 2, qui a de nouveaux checkpoints et une nouvelle image de fond :



Ensuite, nous avons décidé de réaliser un hovercraft plus « joli », ressemblant plus à un



bateau. Nous avons aussi tenté d'ajouter un effet d'eau en mouvement derrière l'hovercraft, lorsque la vitesse de l'hovercraft est supérieure à 0. Le nouvel hovercraft est visible ci-dessous, avec le niveau 3 du jeu :

Finalement, nous avons voulu améliorer le gameplay. Nous avons donc mis les trois niveaux à la suite, en passant au niveau suivant lorsque l'utilisateur réussi le précédent. Cela permet à l'utilisateur d'entrer uniquement le nom de l'exécutable dans le terminal, et non le nom de l'image et du niveau, comme cela l'était avant.

Nous nous sommes aussi dit qu'il serait mieux de mettre une jolie image lorsque l'utilisateur fini les trois niveaux. Nous avons donc remplacé le « Vous avez gagné » par l'affichage d'une image, au-dessus de l'hovercraft :

4. Conclusion

Pour le cours de Synthèse d'Images, nous devions réaliser un hovercraft, qui se déplace dans un univers avec des checkpoints ; son objectif étant de toucher un maximum de checkpoints le plus vite possible. Nous avons réussi à réaliser tous les éléments demandés dans le sujet. Au niveau des options, nous avons ajouté un design pour trois niveaux de jeu différents, un design pour le bateau lui-même, un effet sur l'eau, ainsi que l'affichage du score de joueur et un message lorsque celui-ci réussit le jeu. Nous avons voulu ajouter du texte avec la bibliothèque SDL_SOUND, mais nous n'avons pas réussi à faire fonctionner cela.