Plan de Soutenance

by *EPiCHAT Team*

BALLABENI Stefano — FAHMI Marc
BRIX Jean Rémi — RAUD Cédric

5 janvier 2006

Table des matières

1 Introduction **2**

2 Moteur graphique **3**
2.1 *Affichage de primitives* par JR 3
2.1.1 Ce qui a été fait : 3
2.1.2 3D Studio Max 7
2.1.3 Ce que nous devons faire pour la prochaine soutenance : . . 8
2.2 *Chargement d'un modèle 3D* par Marc 9
2.2.1 Qu'est-ce que la POO ? 10
2.2.2 Pourquoi la POO ? 12

3 Moteur physique **13**
3.1 Introduction 13
3.2 *Déplacements de l'Hoverboard* par Stefano 14
3.2.1 Introduction 14
3.2.2 Où on en est 14
3.2.3 L'avenir, ce qu'on prévoit 17
3.3 *Détection des collisions* par Cédric 18
3.3.1 Introduction 18
3.3.2 Déplacements classiques 18
3.3.3 Collisions avec les bords de la map 19
3.3.4 Collisions avec des objets sur la map 20
3.3.5 Ce qu'il reste à faire 23
3.3.6 Conclusion 24

4 Réseau **25**
4.1 *LeChat* par Cédric 25
4.1.1 Introduction 25
4.1.2 EPiCHAT Team's Chat 25
4.1.3 Ce qui reste à faire / Conclusion 26

5 Conclusion **28**

CHAPITRE 1

Introduction

Hoverboard X-TREM (HXT pour les intimes) est un jeu de course axé arcade sur des planches de skateboard volantes, vulgairement appelées hoverboards. Il est en cours de développement par quatre épitéens en quête d'expérience : Jean-Rémi, Cédric, Marc et Stefano. Chacun dans cette aventure s'est vu confier différentes tâches, on retrouvera alors Stefano avec Cédric au moteur physique, au réseau et à la réalisation du site web et avec Marc au moteur de jeu, Cédric qui, avec Jean-Rémi s'occupera de l'intelligence artificielle. Ce dernier quand à lui partagera avec Marc les joies du moteur graphique, de la gestion du son, et des graphismes en général.

L'heure du bilan sonne en ce début du mois de janvier, et pas mal de premières étapes ont été franchies. On constatera que le côté physique du jeu commence à prendre forme, alors que le graphique pour sa part a percé sa coquille et s'impose petit à petit dans HXT. On regrettera malheureusement l'absence de la gestion des sons et de l'IA, qui sont prévus pour des soutenances ultérieures.

Alors qu'au départ chacun codait de son côté, c'est autour de Marc que depuis quelques temps nos aventuriers de l'impossible se retrouvent afin de mettre en commun le produit de leur peines. Dorénavant l'entraide est de mise, et l'union fait notre force !

Pour l'instant le projet n'est disponible qu'en une multitude d'executables chacun représentant du travail d'un membre, mais dans un avenir certain, la réunion de tous nos travaux donnera naissance à HOVERBOARD X-TREM !

CHAPITRE 2

Moteur graphique

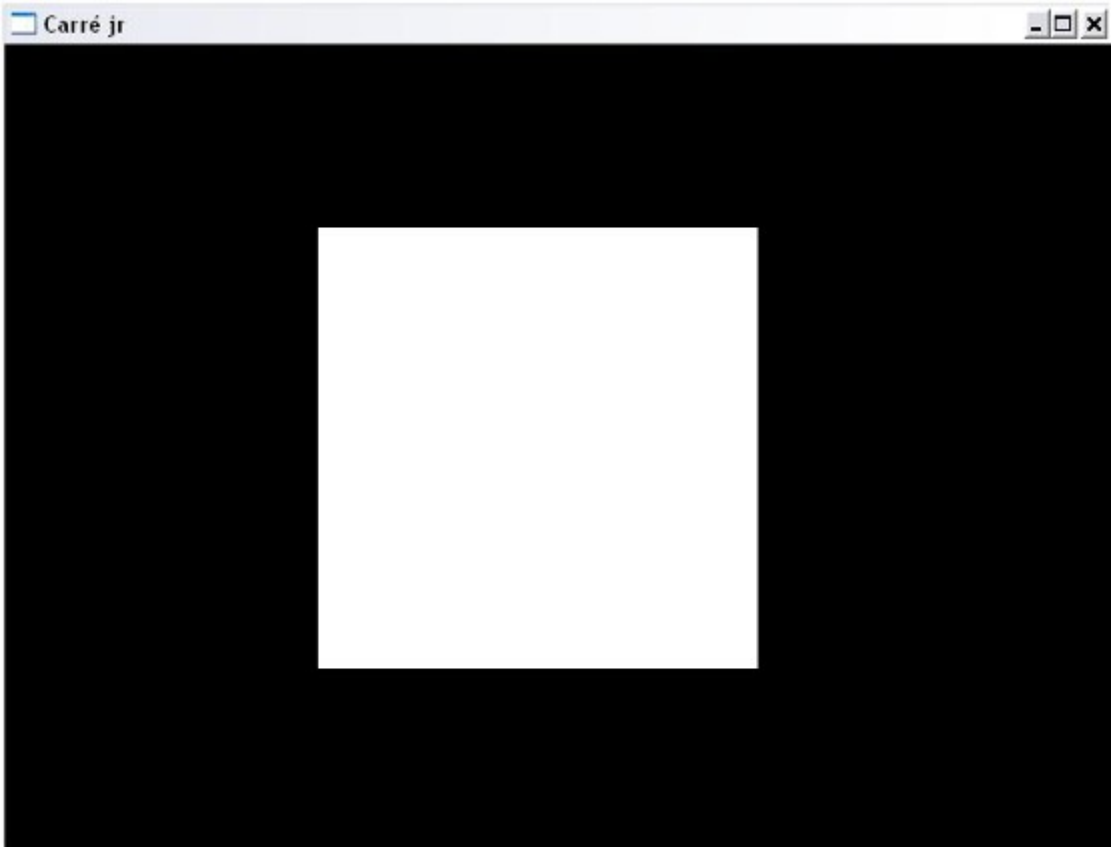
2.1 *Affichage de primitives* par JR

2.1.1 Ce qui a été fait :

Nous avons choisi OpenGL au profit de DirectX car Open GL convenait parfaitement à une utilisation 2D et 3D efficace et la gestion des fenêtres se fait très facilement en utilisant la librairie GLFW, le tout restant multi plateforme. Du côté musique et effets sonores, OpenAL ou fmod remplacent très bien DirectSound tout en restant également multi plateformes.

J'ai donc essayé de maitriser les bases de l'ouverture d'une fenêtre OpenGL avec l'aide de différents tutoriaux et codes sources mis à disposition par Internet, c'est ainsi que nous nous sommes rapidement rendu compte qu'il faudrait utiliser une librairie annexe regroupant des fonctions simples. C'est pour cela que nous nous sommes intéressés à la librairie GLFW.

Il faut néanmoins remarquer que la simplicité d'initialisation d'une fenêtre OpenGL à l'aide de GLFW nous permit de très vite nous concentrer sur la structure même d'une boucle d'affichage, c'est ainsi que j'ai réalisé un petit programme très simple qui affiche un carré à l'écran.



J'ai créé une procédure afin d'afficher les textures et d'initialiser toutes les données d'OpenGL comme les paramètres de la fenêtre (couleur, taille...), le point de vue de la caméra, la position de la caméra, l'angle du champ de vision, la distance minimum et maximum du champ de vision. Cette procédure permet aussi d'établir les rotations possibles ainsi que les translations.

J'ai commencé progressivement pour l'affichage en 3D. A l'aide de tutoriaux basiques OpenGL, j'ai appris à afficher tout d'abord un trait, puis un carré, puis un sol (non texturé) et enfin un cube coloré et texturés par la suite. Tout ceci à l'aide de fonctions d'Open GL : glcolor3f(r,b,v) pour la couleur et glvertex3F(X,Y,Z) pour les coordonnées des épingles.

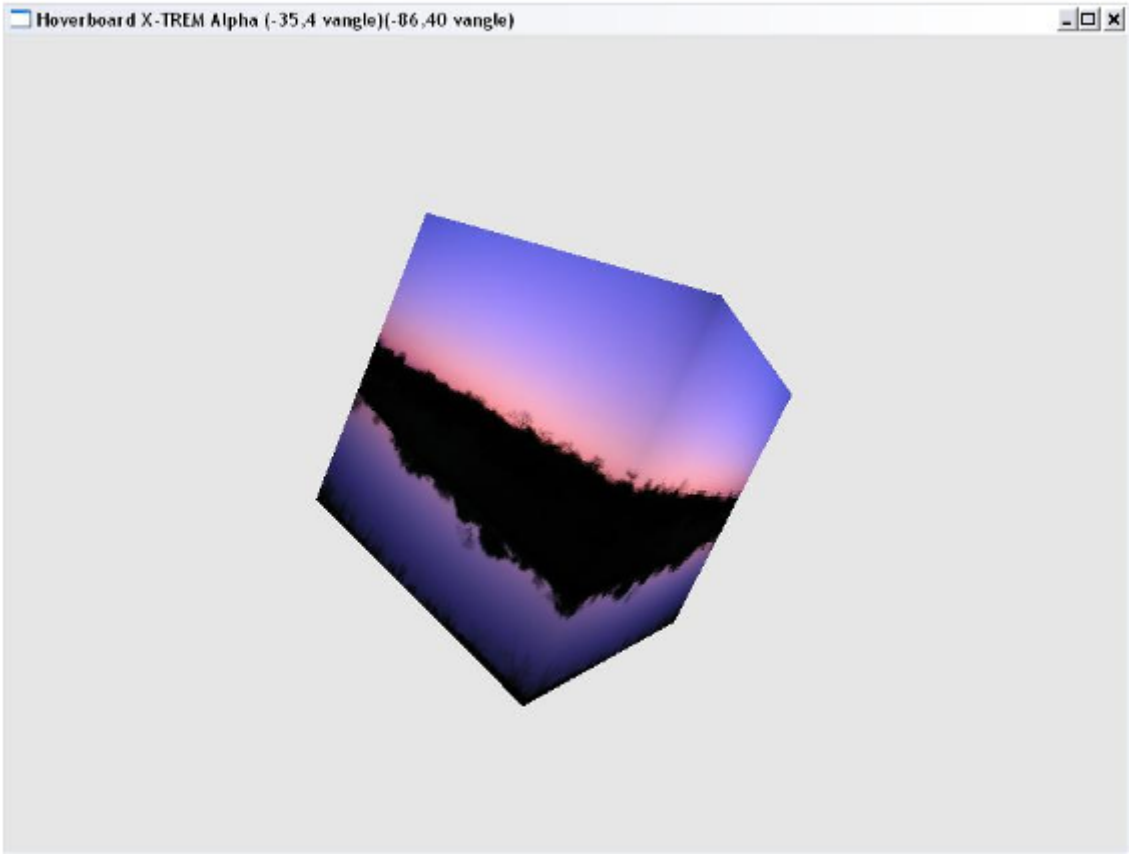


FIG. 2.1 – Cube texturé

Après avoir réussi à afficher quelque chose ressemblant à un sol, quatre murs et un plafond, nous avons décidés de tenter d'appliquer des textures dessus pour que cela ressemble à une skybox. Pour débuter, j'ai utilisé des textures pour qu'on ait l'impression d'être au milieu d'un paysage en 3d. Nous avons rencontré des problèmes pour afficher plusieurs textures, mais nous avons tout de même réussi afficher toutes les textures.

Voila ce que rend la skybox à l'aide de la procédure gluLookAt(0, 1, 0, 0, 0, 0, 1, 0, 0) qui permet de positionner la camera dans l'environnement 3D.

```
procedure Dessin;
  begin
    glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity;

    gluLookAt(0, 1, 0, 0, 0, 0, 1, 0, 0);

    glBindTexture(GL_TEXTURE_2D, Texture);

    glClearColor(0.9, 0.9, 0.9, 0);
    glRotate(vAngle, 0, 0, 1);
    glRotate(hAngle, 1, 0, 0);
    glScale(4, 4, 4);
```

gluLookAt() simplifie la spécification de transformation de la visualisation. La caméra est positionnée en (camx,camy,camz), elle est dirigée vers le point (centrex,centrey,centrez) et l'axe (hautx,hauty,hautz) correspond à la verticale de la caméra. gluLookAt() calcule les rotations et translations nécessaires pour positionner la caméra correctement.

2.1.2 3D Studio Max

3D Studio Max est un logiciel de production d'images de synthèse de qualité professionnelle qui comprend visualisation, modélisation et animation d'objets tridimensionnels. Ce logiciel a été conçu pour les architectes, producteurs vidéo et producteur de jeux vidéo. Il est donc possible avec ce logiciel de modéliser des objets qui ont existé par le passé, d'autres que vous croisez autours de vous ou bien des objets futuristes. Il laisse également la place à tout autre type de création provenant de votre imagination grâce à sa vaste gamme d'outils. Son interface,un peu austère au premier abord, se révèle redoutablement efficace pour ceux qui ont eu suffisamment de volonté pour la maîtriser. De surcroît, l'interface est totalement réglable c'est-à-dire que vous pouvez afficher tous les outils les plus fréquemment utilisés sur le plan de travail. En conclusion de cette brève présentation, ce logiciel est l'outil idéal pour qui aime la 3D et veut l'approfondir pour faire de superbes réalisations ; les seules limites étant votre imagination et la puissance de votre ordinateur ...

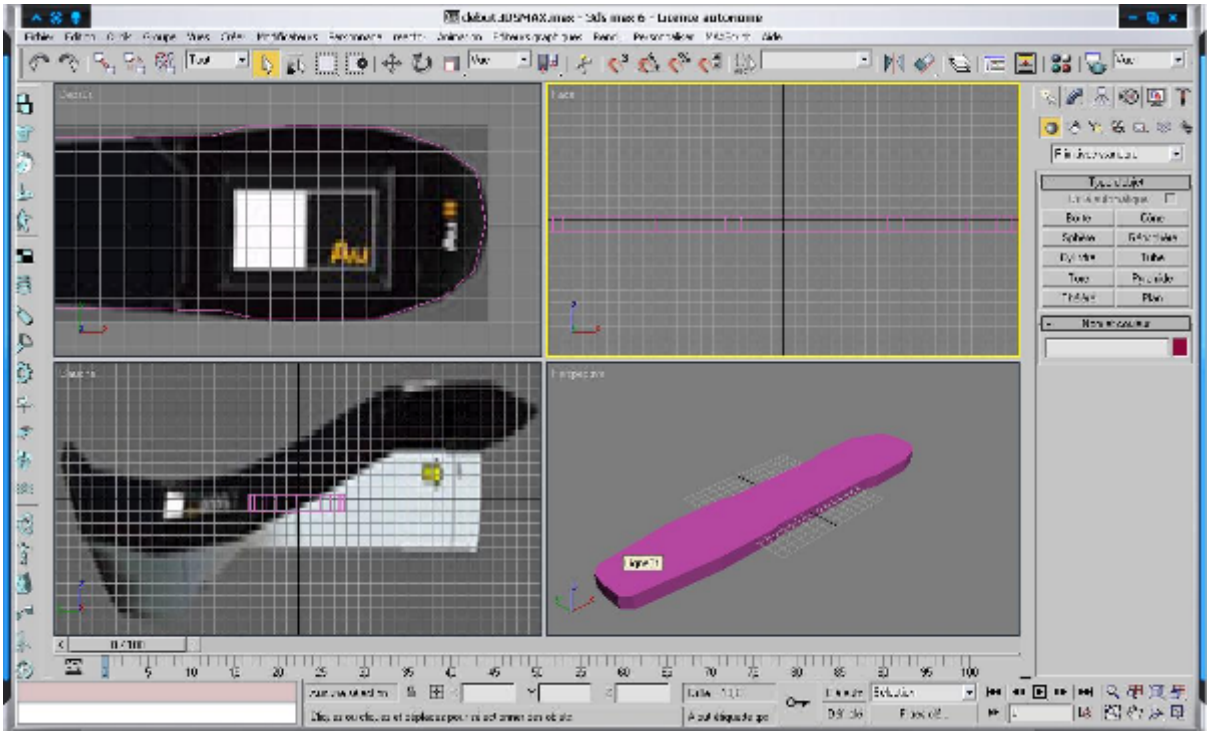


FIG. 2.2 – Création et modélisation d'un hoverboard créé dans 3D Studio Max.

2.1.3 Ce que nous devons faire pour la prochaine soutenance :

Pour la prochaine soutenance, nous prévoyons de créer nos propres textures afin de pouvoir peut-être créer une map en rapport avec le thème de notre jeu mais aussi de pouvoir modéliser dans le programme des textures ou des objets créés avec 3DSMax dans le but d'afficher des personnages avec des hoverboards, du décor.

2.2 *Chargement d'un modèle 3D* par Marc

Après avoir initialisé OpenGL grâce à la libraire GLfw et dessiné des primitives, nous avons tenté de charger un modele 3D et de l'afficher, car nous aurons besoin plus tard dans le jeu d'afficher des objet bien plus complexes que des cubes et des triangles dans la fenêtre (c'est magnifique, les cubes qui se baladent par ci, par là, on voit ça tous les jours dans les jeux video actuels !), pour cela nous avons eu besoin d'un loader 3D. Le format de fichier des modèles 3D choisis est le '.3DS' car d'un point de vue traitement de données, il s'avère très rapide (plus que d'autres), le fichier étant écrit en binaire, ainsi c'est un loader 3DS qui sera utilisé pour le chargement d'objet 3D. C'est ainsi que commença une recherche sur internet pour en trouver un. Ceci fait, il nous a fallu comprendre son fonctionnement pour ensuite l'utiliser avec efficacité (ce qui n'est pas encore totalement le cas). Il nous a donc aussi fallu comprendre la structure d'un fichier 3DS, le décomposer et le lire correctement pour réussir à en extraire correctement les informations nécessaires et qu'elles puissent être utilisables à tout moment (Structure du fichier ci dessous).

[Extrait du site http ://freddec.free.fr/] Le format 3DS est contitué de noeuds (chunck en anglais). Chaque noeud contient des informations sur ses propres données, son identifiant unique (ID) et sur l'emplacement du noeud suivant. Ainsi, si vous ne comprenez pas ce que contient un noeud, vous pouvez quand même passer au suivant. La position (spécifiée en octet) du noeud suivant est relative à la position du noeud courant. Un noeud est défini de la manière suivante : Début Fin Taille Description 0 1 2 Identifiant du noeud (ID) 2 5 4 Taille en octet du noeud

Les nombres font référence à la longueur d'un octet. Les noeuds du fichier ont une hiérarchie imposée par leur identifiant. C'est pourquoi un fichier 3DS commencera toujours par le noeud qui possède l'identifiant 4D4D (en hexadécimal). Pour vous donner un aperçu, le diagramme suivant recense la hiérarchie le plus souvent observée dans un fichier. Les noeuds sont pourvus de noms car cela est plus lisible que des ID hexadécimal.

```
PRINCIPAL (0x4D4D)
|
+--VERSION (0x0002)
|
+--EDITEUR (0x3D3D)
|
+--MAT (0xAFFF)
| |
| +--MAT_NOM (0xA000)
| +--MAT_AMBIENT (0xA010)
| | |
| | +--COL_RGB (0x0011)
| | +--COL_INDEX (0x0012)
| |
| +--MAT_DIFFUSE (0xA020)
| | |
| | +--COL_RGB (0x0011)
| | +--COL_INDEX (0x0012)
```

```
| |
| +--MAT_SPECULAIRE (0xA030)
| | |
| | +--COL_RGB (0x0011)
| | +--COL_INDEX (0x0012)
| |
| +--MAT_ECLAT (0xA040)
| | |
| | +--TYPE_INT (0x0030)
| |
| +--MAT_TRANSPARENCE (0xA050)
| | |
| | +--TYPE_INT (0x0030)
| |
| +--MAT_TEXTURE (0xA200)
| |
| +--MAT_FICHIER (0xA300)
|
+--OBJ (0x4000)
|
+--OBJ_MAILLAGE (0x4100)
|
+--OBJ_SOMMET (0x4110)
+--OBJ_FACE (0x4120)
+--OBJ_MAT (0x4130)
+--OBJ_TEXCOORD (0x4140
```

[fin de l'extrait]

Ainsi apres avoir regardé plusieurs loaders 3DS et de quelle manières ils sont codés, nous avons remarqué que la programmation orienté objet était fréquemment utilisé. Mais qu'est-ce donc et pourquoi est-ce si utile ?

2.2.1 Qu'est-ce que la POO ?

La POO est définie par des classes et des objets. Une classe est ce qui permet de définir la structure et le fonctionnement d'un objet, elle est composée de :

- Champs : ce sont différentes variables appartenant à la classe variable qui peuvent aussi être d'autres classes
- Méthodes : ce sont des procédures ou des fonctions spécifiques à la classe
- Proprietés : permettant d'accéder ou de modifier certains champs de la classe

Une classe ressemble étrangement à un enregistrement sauf pour les methodes et les proprietes, et de ce fait elles sont manipulées différemments.

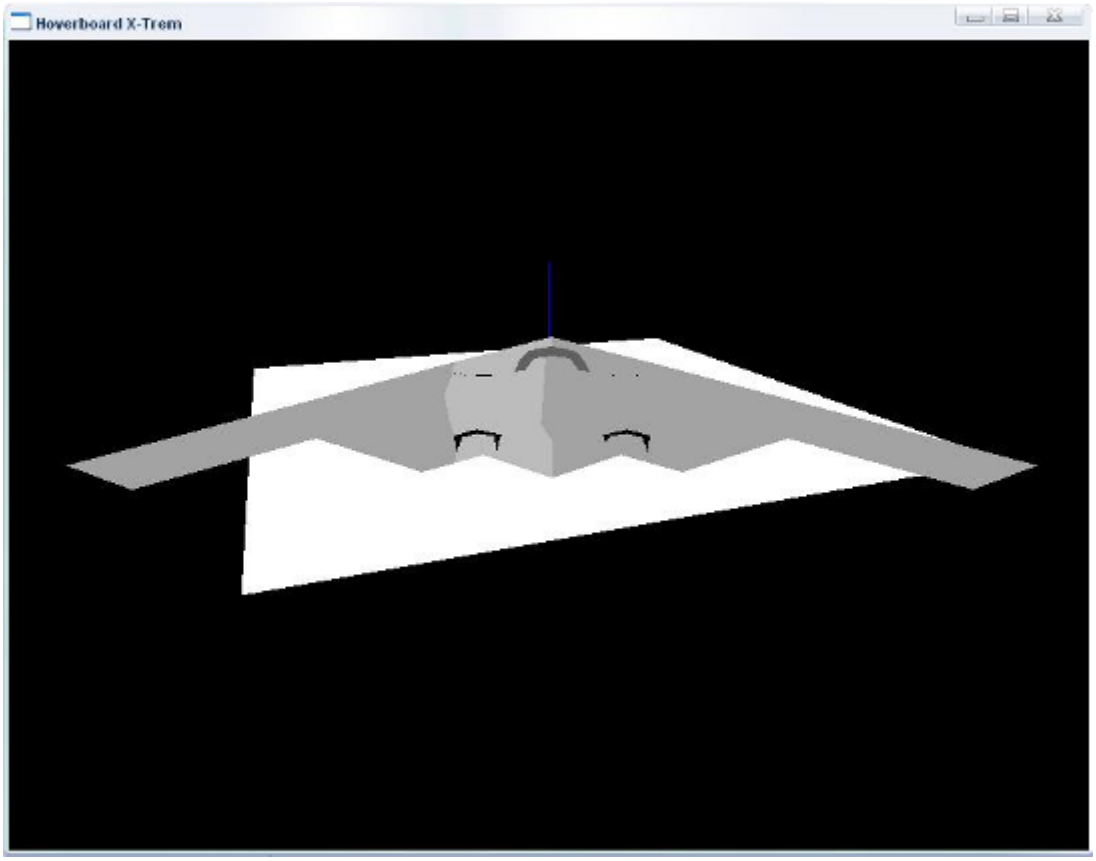
Pour illustrer ceci, voici un exemple de code issue du loader

```
T3DModel = class(TObject)
private
  FMaterials:array of TMaterial;
  FFileHandle:Integer;
  FRootChunk:TChunk;
  function GetMaterialCount: Integer;
```

```
    function GetObjectCount: Integer;
    procedure CleanUp;
    procedure ComputeNormals;
  public
    Objects:array of T3DObject;
    constructor Create;
    destructor Destroy;override;
    function AddMaterial:TMaterial;
    function AddObject:T3DObject;
    procedure Clear;
    procedure VisibleAll;
    function LoadFromFile(const FileName:string):Boolean;
    function FindObject(const aName:string):T3DObject;
    function Select(const Index:Integer):T3DObject;
    procedure Draw;
    property ObjectCount:Integer read GetObjectCount;
    property MaterialCount:Integer read GetMaterialCount;
end;
```

Ici on definit la classe T3DModel et cette classe permet de manipuler un modèle 3D. Pour l'utiliser, on déclare un objet de type T3DModel de cette maniere : `Model: T3DModel;` Il faut ensuite l'instancier, c'est-à-dire lui allouer de l'espace memoire. On peut ensuite utiliser les méthodes définies pour cette classe d'objet. Ainsi pour charger un fichiers 3DS, on ecrira par exemple avec `Model.LoadFromFile('NomDuFichier.3DS');` Le fichier est ainsi lu et chargé en mémoire, et tous les champs sont remplis d'après les informations contenus dans le fichier 3DS. Le modèle 3D peut ainsi être affiché grace à la fonction Draw. Dans une procedure d'affichage par exemple elle sera utlisé de cette manière :

```
procedure affichage( paramètres si besoin est);
    Var
        variable si besoin est;
    begin
...
...
Model3D.Draw ;
...
...
    end;
```



2.2.2 Pourquoi la POO ?

La POO est utlisée pour la manipulation d'objets dont le type n'est en général pas prédéfini. Par exemple dans Delphi, les Form, Label, Edit, Bouton ou autres sont des Objet definit a partir d'une classe. Dans notre cas, la POO est utilisé dans le loader pour faciliter la manipulation des differentes Instances contenues dans le fichier chargé. On peut ainsi manipuler les objets 3D, les materiaux, les transformations (type transformation) ou tout autre proprieté (si définies et si besoin est).

CHAPITRE 3

Moteur physique

3.1 Introduction

Un jeu de course sans moteur physique étant aussi passionant qu'un démineur sans la souris, un gros travail devait être effectué de ce coté là pour que le comportement de la planche ne soit pas trop ridicule. Un Hoverboard a beau tenir de la science-fiction, il n'en reste pas moins que ses mouvements doivent à l'inverse être réalistes ! Quand on pense au moteur physique, deux composantes nous viennent immédiatement à l'esprit : le déplacement de la planche sur un circuit contenant des obstacles et la manière dont il se déplace. Le premier relève de la détection des obstacles et de la géométrie de la map et le second est en rapport avec les forces extérieures appliquées sur la planche comme l'accélération. Etant donné qu'il est plus judicieux de découvrir comment l'Hoverboard se déplace avant de voir ce qui l'arrête, les déplacements de la planche seront ici présentés en premier.

3.2 *Déplacements de l'Hoverboard* par Stefano

3.2.1 Introduction

Si on vous donnait un jeu de course et que votre véhicule (quel qu'il soit) restait immobile pendant que vos concurents vont, cheveux au vent vers la ligne d'arrivée, vous vous diriez qu'on vous a pris pour une mauviette, et chez EPiCHAT Team on est pas comme ça !

Comme vous n'êtes pas des mauviettes, nous sommes actuellement en train de vous développer un système de gestion des déplacements révolutionnaire ! Par ailleurs les pourparler avec plusieurs studios de développement de jeux vidéos qui désireraient (déjà) nous le racheter quand il sera achevé avancent bien. Mais trève de blablas, HXT sera gratuit et OpenSource !

Tout le monde veut savoir où nous en sommes, et aujourd'hui la réponse est là.

3.2.2 Où on en est

A ce stade du developpement, le moteur de déplacement n'est pas vraiment utilisable dans le cadre d'un jeu car les mouvements sont trop irréalistes pour être exploités, cependant on peut diviser en trois parties ce qui a été fait jusqu'à présent.

Les mouvements : L'altitude de la planche est pour l'instant relativement constante mais en réalité un subtil effet de flottaison est créé grâce à la fonction sinus, qui est comme son nom l'indique sinusoïdale, et varie régulièrement entre -1 et 1. L'altitude de la planche dépend alors d'un code de ce type :

$$z = \frac{sin(\frac{i}{vZ})}{aZ}$$

z est la variable qui représente l'altitude. i est une variable qui est incrémentée à chaque itération modulo $2 * pi$ (une période de la fonction sinus). vZ est une constante qui sert à 'ralentir' l'effet de flotaison (i étant réduit, les valeurs varient moins vite). Dès lors le modulo de variation de i est $2 * pi * vZ$. Enfin, aZ est également une constante qui nous permet de réduire l'amplitude des variations de z, afin que la planche ait un mouvement plus réaliste.

Voila pour l'altitude.

Si on ne considère pas l'altitude qui ne sert qu'à créer un effet visuel, le déplacement de l'hoverboard se fait sur un plan (xOy). Pour créer l'hoverboard,

une suite de carrés, de rectangles et de triangle nous permettent d'avoir une forme plus ou moins proche d'un hoverboard, parfait pour les essais.

Pour faire simple, on 'crée' temporairement un nouveau repère auquel on applique des transformations (rotations, homothéties et translations) puis on replace ce repère dans le repère original, ce qui nous permet de créer des déplacements facilement.

```
glPushMatrix;
  glTranslate(posX, posY, posZ);
  glRotate(angle, 0, 0, 1);
  glScale(1, 1, 1/6);
  glCallList(id_hoverboard);
glPopMatrix;
```

glPushMatrix représente cette création du nouveau repère, et il s'achève par un glPopMatrix, qui 'remet' le repère dans le repère initial. Entre les deux, on place les transformations qu'on veut appliquer, puis on apelle une glCallList, qui est une liste d'operations OpenGL, qui nous permettent de faire le dessin de la planche.

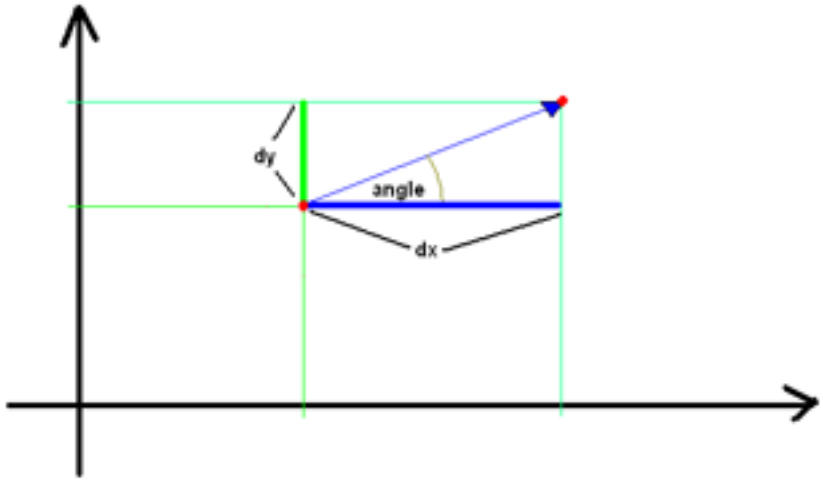
On applique d'abord une translation, puis une rotation, la translation est créée pour mettre dans les nouvelles positions de x, y et z (avec les variables posX, posY et posZ –j'expliquerai plus tard comment on calcule posX et posY, par contre posZ je vous l'ai déjà dit. Si, si, souvenez vous des sinus !). C'est seulement APRES la translation qu'on applique une rotation, afin que la planche ait l'air d'aller toujours vers l'avant, et pas de se déplacer sur le côté par exemple (on ne fait pas un hoverboard *Rally dans la boue* !). Le glScale permet de créer des homothéties. Ici on mutlitplie par 1 la longueur et la largeur (ça ne change pas donc) et par 1/6 la hauteur (bah ouais, c'est une planche, pas un tronc !). Enfin, sur ce nouveau repère on apelle la liste opengl qui contient les coordonnées des différents points à afficher pour dessiner notre hoverboard.

C'est bien beau de positionner la planche, mais encore faut il intéragir avec ! Et oui, avec une planche qui ne bouge pas, vous risquez toujours de passer pour une mauviette ! Alors voici la première utilisation intéressante (à part l'ouverture de la fenêtre bien sur :p) de GLFW ! Et oui, GLFW gère les inputs claviers, et de manière très aisée qui plus est ! (Il gère aussi les inputs souris, et joystick, et d'autres belles choses encore, mais nous n'en avons pas encore besoin)

Pour vérifier qu'une touche est appuyée, on utilise `glfwGetKey(valeur de la touche)`. Comme ici on va utiliser les flèches du clavier, les valeurs des touches seront :

```
-GLFW_KEY_UP : pour la touche HAUT.
-GLFW_KEY_DOWN : pour la touche BAS.
-GLFW_KEY_LEFT : pour la touche GAUCHE.
-GLFW_KEY_RIGHT : pour la touche DROITE.
```

Alors si on revient à notre hoverboard et qu'on réflechit à ce qu'on l'on veut obtenir, on arrive à quelque chose du genre : Si l'on appuit sur HAUT, alors la planche va de l'avant. Si l'on appuit sur BAS, alors la planche va de l'arrière. Si l'on appuit sur GAUCHE, alors la planche tourne vers la gauche. Si l'on appuit sur DROITE, alors la planche tourne vers la droite.



Le code s'approche de quelque chose comme ça :

```
if glfwGetKey(GLFW\_KEY\_UP)=1 then
begin
  posX:= posX + cos(angle)*dx;
  posY:= posY + sin(angle)*dy;
end;

if glfwGetKey(GLFW_KEY_LEFT) = 1 then
begin
  angle:= angle + dAngle;
end;
```

Il n'y a ici que la gestion de l'appui sur la touche HAUT et la touche GAUCHE, mais le fonctionnement est analogue pour la touche BAS et la touche DROITE. Jettons un oeil sur la gestion de l'appui sur HAUT. A la variable posX, qui représente la position au moment où on l'écrit de l'hoverboard sur l'axe Ox, on ajoute cos(angle), que l'on multiplie à la variation de x voulue. Cette variation peut être prédéfinie et constante, ou dépendante de la fonction vitesse (ce dernier cas permettant de faire varier la vitesse de déplacement de l'hoverboard).

cos(angle) est là car on se déplace vers l'endroit où pointe l'hoverboard, et si il a un angle de 20°, on ne va pas faire comme si c'était toujours un angle de 0°, il serait alors possible de tourner, mais pas de tourner, et foncer dans un mur fait passer pour une mauviette !

Le principe est le même pour posY.

Quand on appuit sur gauche, on augmente l'angle de la variation angulaire voulue (dAngle), ce qui permet de donner l'orientation de l'hoverboard (quand on utilise le glRotate) et de lui permettre de se déplacer dans cette direction grâce à la définition des posX et posY (utilisés avec glTranslate par la suite).

Les cheveux aux vents, les moucherons entre les dents ! Quelle sensation en effet que de pouvoir donner tout ce qu'on a pour gagner une course d'hoverboard ! Si vous n'avez jamais essayé, je vous conseille vivement d'attendre la version finale d'HXT pour vous ruer dessus. En attendant, pour que ce jeu ait

de l'intéret, il faut faire varier les vitesses, parcequ'à une vitesse constante, le jeu devient beaucoup moins tactique (qui à parlé de MotoGP ? ? ?).

On a vu tout à l'heure que dx pouvait dépendre de la vitesse. En effet, si on augmente la valeur de dx, le déplacement effectué en un tour sera plus grand. N'est-ce pas ce qu'on apelle augmenter la vitesse ?

A ce stade du jeu, la vitesse n'est pas à sa phase finale. Par exemple on n'a encore que des accelérations constantes (entières).

Pour l'instant pour augmenter la vitesse, on crée une variable t qui s'incrémente quand on appuye sur haut et qui représente la durée d'accélération. Cette variable t, on la multiplie par l'accélération, et on multiplie le tout à dx, ainsi plus on accelère, plus t augmente, plus $t * acceleration$ augemente (en valeur absolue, car si on veut reculer on crée une vitesse négative en créant une acceleration négative, qui permet de faire diminuer la vitesse), et plus $t * acceleration$ augmente, plus dx auquel ils sont multipliés augmente et plus la vitesse augmente, tout simplement. Encore une fois la méthode est la même pour dy.

Quand on tourne, il est logique que la planche ne reste pas parralèle au sol, en effet, je ne sais pas si vous avez déjà fait de la moto, mais pour tourner on penche la moto dans le sens où l'on veut aller. Pour l'hoverboard, tout le monde le sait, c'est pareil !

On a donc ajouté une nouvelle variable qui représente l'angle d'inclinaison de la planche.

Si on revient sur la rotation de tout à l'heure, on avait glRotate(angle, 0, 0, 1) qui créait une rotation d'angle 'angle' selon l'axe Oz. Et bien là on rajoute tout simplement un `glRotate(angle_inclinaison_virage, 1, 0, 0)` qui créera une inclinaison d'angle `"angle_inclinaison_virage"` selon l'axe Ox. Cependant, pour que l'effet soit réaliste on fait en sorte que cet angle ne varie qu'entre -35 et 35 degrés et que quand on ne tourne pas, la planche revienne à plat (c'est à dire qu'on fait revenir l'angle à 0).

3.2.3 L'avenir, ce qu'on prévoit

La gestion de la vitesse n'est pas réaliste, pour les prochaines soutenances l'accélération ne sera plus constante, mais dépendra de la pente sur laquelle circule l'hoverboard (plus faible en montée, plus grande en descente), selon la densité des élements sur lesquels les hoverboards se déplacent (sur l'eau, sur de l'herbe qui vient accrocher dans les propulseurs et nous ralentit...) et selon plein d'autres choses qui offriront un effet réaliste à notre jeu.

De même la gestion des rotations ne nous satisfait pas. L'hoverboard tourne ici selon son centre de gravité, et selon rien d'autre (l'angle entre en compte aussi, mais on peut presque le considérer comme mineur !). En effet, on ne tient pas compte de la force centrifuge qui fait que l'on ne peut pas faire un virage à 90° comme par magie à 250km/h (il va falloir vous accrocher à la planche !).

De même on pense faire en sorte que l'avant de la planche se soulève légèrement lors des virages lorsque lon ralentit pour mieux les prendre.

Bref, au point de vue de la gestion des mouvements de la planche, il reste encore pas mal de boulot !

3.3 *Détection des collisions* par Cédric

3.3.1 Introduction

Au même titre que l'accélération, que serait un jeu de course sans obstacles ? C'est ainsi dans le but d'éviter le syndrôme de la ligne droite que je m'attelai à la réalisation d'un moteur physique capable de détecter si la planche rentrait ou non dans un obstacle quelconque.

3.3.2 Déplacements classiques

La première tache est l'initialisation d'une fenêtre graphique de base avec un objet déplaçable au clavier. Histoire de me familiariser un peu avec GLFW (notre librairie graphique, pour ceux qui lisent le rapport en partant de la fin) je commence par représenter un rectangle en 3D, fier représentant de la nouvelle génération d'Hoverboards. On assiste ainsi à la naissance de ceci :

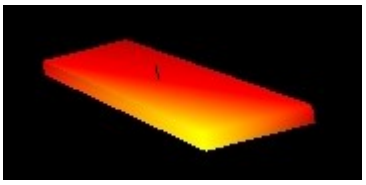


FIG. 3.1 – Hoverboard 1er

Une fois ceci fait, le faire se déplacer ne fut pas trop difficile :

```
if (glfwGetKey( GLFW_KEY_UP )= GLFW_PRESS) then // Déplacement en avant
    begin
     posz:=posz+vitesse;
    end;
```

La condition `glfwGetKey( GLFW_KEY_UP )= GLFW_PRESS` permet de tester si la touche 'Flèche haut' est enfoncée. Si oui, la position en z (n) est augmentée. Note : dans mon programme, l'axe z est l'axe vertical et pointe vers le haut de l'écran. C'est ensuite la fonction '`glTranslatef( posx, 30.0, posz );`' qui se charge de déplacer véritablement l'objet à chaque tour dans la boucle du jeu. Seulement il est vite apparu qu'une simple translation ne pouvait pas suffire pour déplacer la planche. Le schéma suivant explique pourquoi :

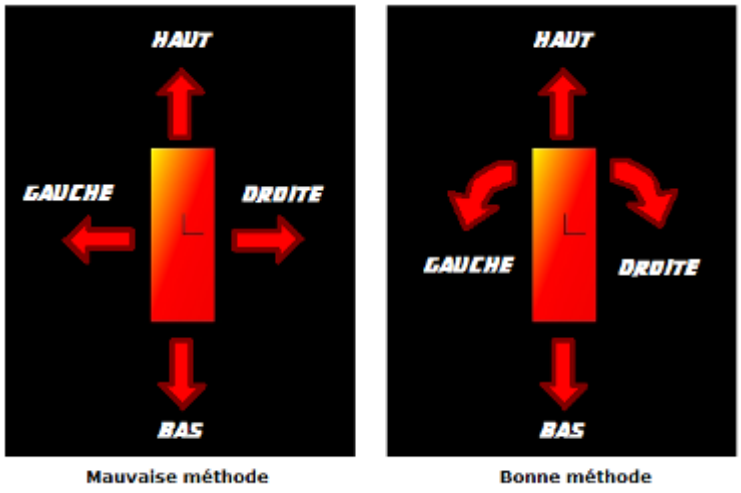


FIG. 3.2 – Schéma des déplacements

Les touches horizontales doivent donc servir à faire tourner la planche et les touches verticales à la faire de déplacer. Pour cela, nos amis cosinus et sinus viennent nous donner un coup de main :

```
posz := posz + speed/fps*cos(DegToRad(angley));
```

Comme vous pouvez le voir, nous multiplions le cosinus à speed/fps. Comme leurs noms l'indiquent, ces variables sont en relation avec la vitesse de la planche et le nombre d'images par secondes (frame per second) de l'application. Le coefficient speed est une contante pour le moment car je ne m'occupe que des collisions. Faire dépendre la vitesse du jeu au nombre de FPS est primordial : cela permet que le jeu tourne à la même vitesse sur différentes machines où le nombre de fps peut varier du simple au dixuple ! A titre d'exemple, cette petite application avec le simple rectangle tournait à 500 fps sur mon portable pour 50 fps sur les PCs de l'école...

La dernière modification nécessaire pour le déplacements est l'inversion de la direction lors de la marche arrière.

3.3.3 Collisions avec les bords de la map

Maintenant que nous avons une planche qui se déplace, l'heure est venu de détecter les bords de notre map virtuelle qui n'est pour l'instant qu'un simple rectangle recouvrant une grande partie de l'écran. Ces fameux bords sont simplement définits pour le moment par quatres variables : minx, maxx, minz et maxz. Il suffirait de comparer les valeurs de posx et posz avec ces valeurs minimales et maximales me diriez vous. Que nenni ! En effet, posx et posz correspondent au centre de ma planche et mon but est de tester si les bords de la planche ne dépassent pas les bords de la map. Analysons la situation... Les points du rectangle de la planche sont stockés dans le tableau suivant :

```
{Points[1][1]
```

```
          *---------- Numéro du coin [1...4]
                     | 1 -> Point en bas  à gauche
                     | 2 -> Point en bas  à droite
                     | 3 -> Point en haut à droite
                     | 4 -> Point en haut à gauche
             *------- Coordonnées [1;2]
                     | 1 -> x
                     | 2 -> z}
  Points: array [1..4] of array [1..2] of Double;
```

Il faudrait donc mettre à jour ces coordonnées à chaque translation ou rotation de la planche. Pour cela, nous créons un second tableau
(`Points,MPoints: array [1..4] of array [1..2] of Double;`) que nous mettrons à jour à l'aide d'une fonction qui se charge de calculer toutes les coordonnées à partir de posx, posz et angley, l'angle de la planche.

Il n'y a plus ensuite qu'à tester si la planche reste bien dans la map à chaque nouvelle position. Dans le cas contraire, elle ne bouge pas sur l'axe où le point est en dehors.

3.3.4 Collisions avec des objets sur la map

Nous attaquons maintenant la grosse partie du moteur physique. En effet, si vérifier si la la planche ne sort pas était simple, vérifier qu'elle n'entre pas en collision avec un objet quelconque l'est déjà moins ! Pour commencer, l'idéal est de mettre en place la détection avec des objets rectangulaires. Ceux-ci sont stockés dans le tableau suivant :

```
{Tableau dynamique listant tous les quadrilatères de la map :
  obj_quad[0][1][1]
            *---------- Numéro de l'objet [0...(Nombre d'objets-1)]
               *------- Numéro du coin [1...4]
                      | 1 -> Point en bas  à gauche
                      | 2 -> Point en bas  à droite
                      | 3 -> Point en haut à droite
                      | 4 -> Point en haut à gauche
                  *---- Coordonnées [1;2]
                      | 1 -> x
                      | 2 -> z }
  obj_quad: array of array[1..4] of array[1..2] of Double;
```

Le tableau `obj_quad` est dynamique ce qui signifie qu'il suffit d'agrandir sa taille pour ajouter un autre élément. Notez que le tableau en lui même n'empêche pas d'enregistrer des objets non rectangulaires mais nous parlerons d'eux un peu plus loin... Qu'est ce que qui définit si un point de la planche est à l'intérieur ou à l'extérieur d'un rectangle ? Il doit respecter les conditions suivantes :

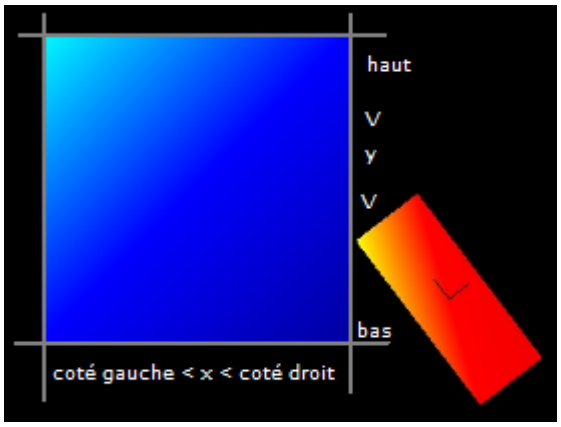


FIG. 3.3 – Collision avec un rectangle

Il ne reste ensuite qu'à effectuer ce test sur tous les points de la planche et sur tous les rectangles.

Mais que se passerait-il si notre quadrilatère n'était pas un rectangle ? Notre méthode n'est alors plus valable car la valeur à ne pas dépasser en x dépend de z et inversement !

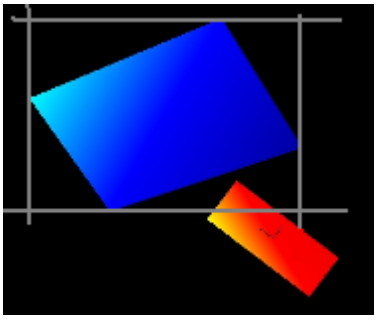


FIG. 3.4 – Collision avec un quadrilatère

Néanmoins, nous pouvons quand même affirmer que si le point n'est pas dans le rectangle qui est formé par les extrémités, il n'y a pas de collisions. Ainsi, inutile de continuer à tester si le point ne remplie pas cette condition.

Pour vérifier si le point est vraiment dans la forme, j'ai obté pour la méthode suivante : je prends les quatres extrémitées du quadrilatère deux à deux en tournant dans le sens trigonométrique, et j'étudie la position du point à tester par rapport à la droite formée par mes deux extrémitées. Dans notre cas, le point doit ainsi répondre aux conditions suivantes :

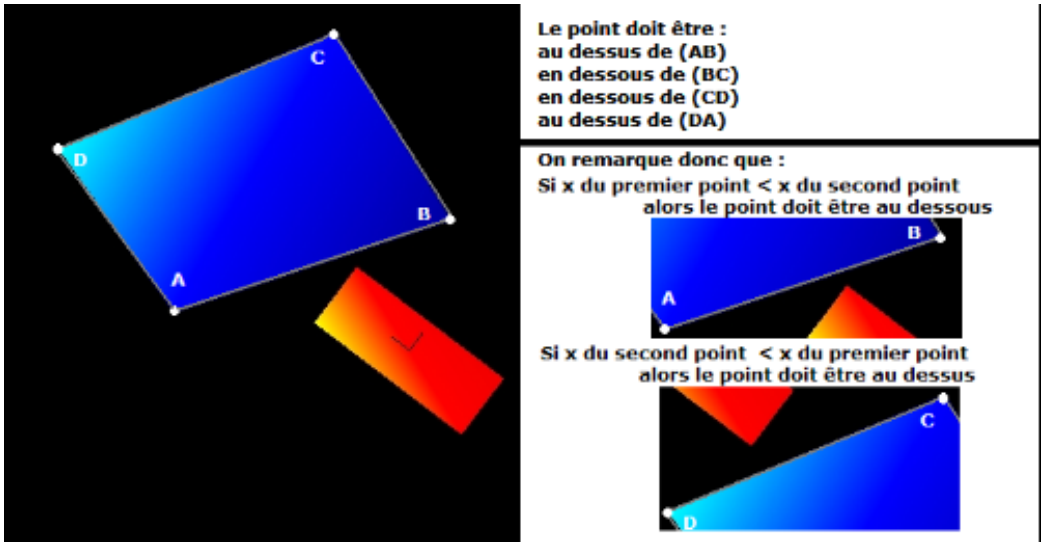


FIG. 3.5 – Collision avec un quadrilatère, explication

Du coté de l'implémentation, la fonction qui s'occupe du test des collisions va ainsi tester tous les points de l'Hoverboard avec toutes les droites formées par les extrémités de tous les quadrilatères. Pour tester si le point est en dessous ou en dessus de la droite, la formule suivante convient à merveille (nous prendrons le A comme premier point, B comme second et X et Y les coordonnées du point à tester) :

$(\frac{PointB_y - PointA_y}{PointB_x - PointA_x} * (X - PointB_x) + PointB_y) >= Y) = (PointA_x < Point_x)$

Une fois cette formule trouvé le reste n'était de l'écriture vu qu'adapter le code pour qu'il puisse aussi gérer des triangle était singulièrement indentique à ce que nous avons fait jusqu'à présent. De la même manière, il est nécessaire d'ajouter un second test à celui-ci : tester que toutes les extrémitées des objets n'entrent pas en collisions avec l'intérieur de la planche. En somme, le test inverse que le précédent. Sans celui-ci, nous arriverions à des situations comme celle-ci :

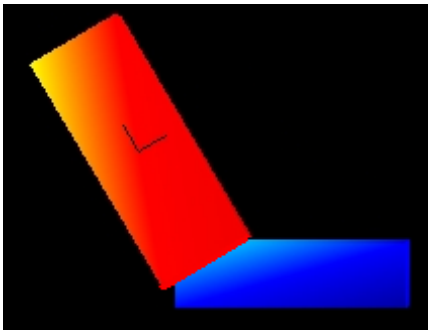


FIG. 3.6 – Collision avec un quadrilatère, le second cas

3.3.5 Ce qu'il reste à faire

- Pour l'instant le moteur se limite à une représentation 2D de la scène. Il va falloir dans le futur amméliorer grandement le moteur et faire la fusion avec la partie déplacement et accélération pour pouvoir rendre la planche un peu plus dynamique (qu'elle puisse prendre un tremplin par exemple).
- Nous comptons au final avoir une map à volumétrie variable sur plusieurs niveaux. Imaginons par exemple que la planche entre à un moment dans un immeuble, deux chemins s'offriront alors à elle : le rez de chaussé ou le premier étage. Ce qui signifie qu'il va falloir optimiser le moteur pour qu'il ne tienne compte que des objets qui sont sur le même niveau que la planche.
- En bonus, il sera peut être intéressant d'ajouter une gestion des objets cylindriques. D'un point de vue graphique, il est très difficile de les afficher mais ils pourront être utilisés pour modéliser des objets graphiques plus ou moins cylindriques

Enfin, nous verrons bien si ces projets seront concrêtement utiles et réalisables...

3.3.6 Conclusion

Nous avons donc un moteur physique capable de détecter la moindre collision d'une planche rectangulaire (ou non) avec un quadrilatère ou un triangle quelconque. Les collisions ne sont gérées qu'en 2D mais une vue 3D est tout de même possible et il faut avouer qu'on commence déjà à appercevoir les prémices de notre jeu. Toujours est-il qu'il remplit sa tache à merveille pour l'instant et qu'il constitue une bonne base pour ajouter les autres éléments du jeu. Voilà une bonne chose de faite !

CHAPITRE 4

Réseau

4.1 *LeChat* par Cédric

4.1.1 Introduction

Comme convenu dans le Cahier des Charges, j'ai commencé à m'intéresser au fonctionnement du réseau en vue d'une implémentation future à notre jeu. Grace aux informations fournies par nos Spés préférés lors des conférences de début d'année, il nous est apparu que ICS était un bon choix pour notre jeu puisqu'il est à la fois relativement bien documenté et ne bloque pas l'application contrairement à son concurrent Indy. Restait ensuite le choix du protocole. UDP ou TCP ? Je me suis concentré pour l'instant sur le second dont les paquets envoyés à travers le réseau ont l'avantage de ne pas se perdre et d'arriver dans l'ordre. L'application la plus simple pour apprivoiser le réseau étant un chat (normal, c'est doux un chat et...non, rien), ce fut donc mon mini-projet réseau pour cette soutenance.

4.1.2 EPiCHAT Team's Chat

Après avoir installé la dernière version d'ICS pour Delphi 7, je pus m'appuyer sur les fichiers exemples fournis avec celui-ci pour comprendre le fonctionnement des composants TWSocket.

Le principal de la tache fut donc de définir les évènement auxquels le client et le serveur intégrés dans le chat devaient réagir. En effet, si l'on démarre le chat comme serveur, il se met en attente d'un client. Si l'on démarre le chat comme client et qu'on essaye de se connecter sur un machine où un serveur tourne, la connexion est effectuée et les clients des deux chats peuvent communiquer ensemble. A la déconnexion de l'un ou l'autre, un message est affiché dans la fenêtre du destinataire.

J'ai passé un peu de temps sur l'interface du chat de sorte à ce que l'utilisateur ne puisse pas entrer un mauvais port ou une mauvaise addresse. Après

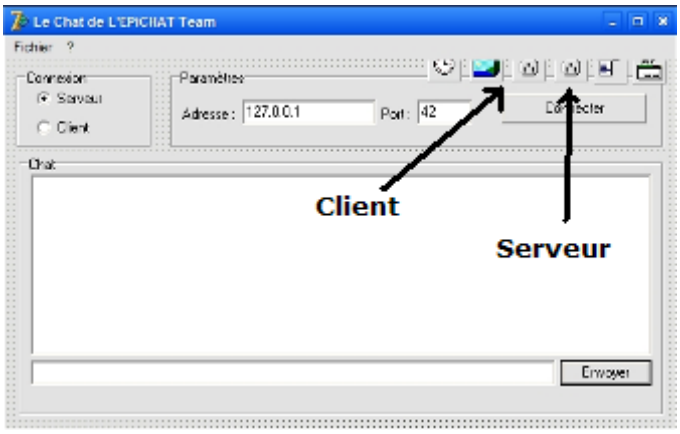


FIG. 4.1 – Les coulisses de la création du Chat !

plusieurs essais en réseau local et à travers le net, nous avons pu constater qu'il marchait parfaitement.

4.1.3 Ce qui reste à faire / Conclusion

Cette application est peut être assez éloignée du projet final mais elle me permit surtout à mieux comprendre le fonctionnement de l'envoi de données à travers un réseau. Dans l'avenir, il va falloir intégrer le serveur et le client ICS dans l'application GLFW pour qu'elle puisse communiquer avec une autre. Le fonctionnement sera assez similaire puisque qu'au lieu d'envoyer de simples messages, ce seront les coordonées des planches qui seront échangées entre clients par le biais du serveur. Il faudra ainsi faire en sorte que plusieurs clients puissent se connecter sur un même serveur. Si les transfert s'avéraient trop lent avec TCP, nous pourrions peut être passer en UDP mais cela, seul l'avenir nous le dira.

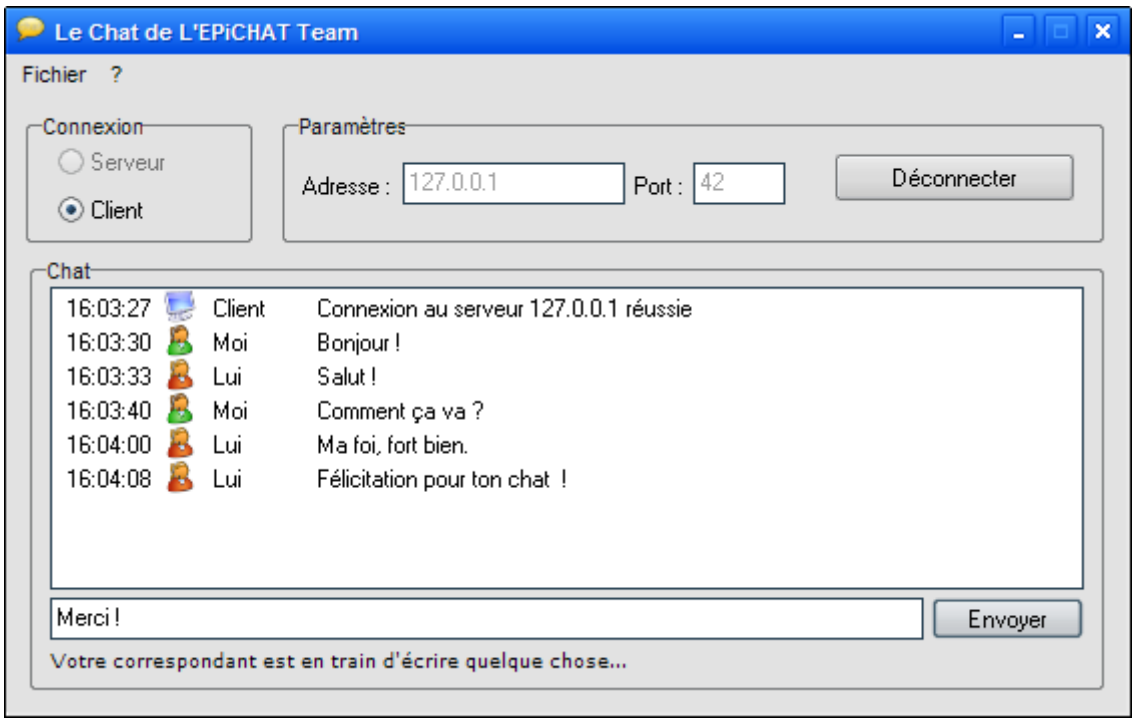


FIG. 4.2 – Et voici LeChat.exe en pleine action !

CHAPITRE 5

Conclusion

Tout d'abord nous sommes satisfait du résultat obtenu jusque là étant donné l'étendu de nos compétences. Nous avons eu beaucoup de mal à débuter, à nous organiser, à nous concentrer sur une longue durée mais une chose est sûre : l'envie de continuer est présente dans l'esprit de chacun.