Rapport de soutenance 2

by *EPiCHAT Team*

BALLABENI Stefano — FAHMI Marc
BRIX Jean Rémi — RAUD Cédric

23 février 2006

Table des matières

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 pour cette deuxième soutenance Cedric au moteur physique, à la réalisation d'un superbe site web tout en Ajax pour faire comme Google (mais en mieux), Jean-Rémi et une skybox nouvelle génération, ainsi qu'un début de mapping plus que prometteur et une intégration d'un foetus de son dans le jeu, Marc nous émerveillera avec un loader de fichiers 3Ds totalement révolutionnaire, accompagné d'une optimisation de la gestion de nos codes, et enfin Stefano, avec son début de gestion d'interface, qui, disont-le, promet d'être boulversante.

Le bilan de ce mois et demi qui nous sépare de la première soutenance est que le jeu, commence mine de rien à pointer le bout de son nez, le moteur physique a été entièrement remanié, on est tous passés à la programmation orienté objet, on a changé de loader, et encore d'autres nouveautés qu'il serait dommage de dévoiler tout de suite.

Nous nous sommes encore une fois retrouvés régulièrement chez Marc pour nos *coding-parties*, et sa maison est maintenant notre Q.G.

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

CHAPITRE 1

Moteur Graphique

1.1 *Création d'une map 2D* par JR

1.1.1 Naissance d'une map

J'ai commencé progressivement par 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 et enfin un cube coloré et texturé 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. Ayant fait le choix avec mes collègues de faire une map, j'ai enfin pu me lancer dans sa création.

Je me suis ensuite dirigé vers la creation de carte constitué de plusieurs carré placés les uns à coté des autres afin de me créer un quadrillage, et m'appercevant qu'il me faudrait quelque chose de plus élaboré, je me suis tourné vers une carte affichée à laide d'un Fichier Texte.
Je me suis vite rendu compte que ce genre de quadrillage n'était pas suffisant pour etre utilisé comme une carte dans le futur jeu.

Je me suis donc tourné vers une carte plus élaborée, et plus pratique pour le futur et pour l'utilisation qu'on compte en faire. Cette carte est donc contenue dans un format map que l'on crée nous meme à partir d'un fichier texte. On appellera ce format la map tout simplement. Il se présente sous cette forme :

```
200
20

22222223333222222222
22222223444322222222
22222223444322222222
22222223444322222222
20000000000000100002
20000000000000100002
```

```
      .....
24441111100144444442
2444111100014444002
204401100001000044002
204400100001100044002
204401100001000000002
244440100001100000002
200000333333000000002
200000333333000000002
200000333333000000002
200000333333000000002
200000355553000000002
222222355553222222222
222222355553222222222
```

Le premier nombre correspond à la longueur de la matrice et le deuxieme nombre à la largeur de la matrice en fait à la largeur et la longueur de la map. Nous allons donc ici créer un tableau de 20*200 cases donc un rectangle. On rempli donc ce tableau avec différent chiffres, ici des chiffres allant de 0 à 4. Cela nous servira à approprier une texture différente selon le chiffre. Dans ce cas le 0 represente la texture ligne.bmp, le 1 represente la texture eau.bmp, le 2 represente la texture feu, le 3 represente la texture pierre.bmp, le 4 represente la texture herbe.bmp. Donc nous pouvons imaginer l'eau comme obstacle et le feu comme contour de la map. Maintenant qu'on a notre map, il faut donc la lire et enfin l'afficher. Pour cela on va procéder par différentes étapes : On va d'abord lire les deux premières lignes afin de définir la taille du tableau à l'aide d'une procédure. On dessine ensuite nos carrés.
On reviendra par la suite sur l'application de textures.
A laide du "case of", on va attribuer à chaque numéro du fichier une texture différente, sans oublier de créer notre carré texturé selon son numéro.

```
glGenTextures(4, @Textures);

  Herbe := TFichierBitmap.Create;
  Herbe.ChargerBitmap('dgreen.bmp');
  Herbe.CreerUneTextureGL(256, DimX, DimY);
  glBindTexture(GL_TEXTURE_2D, Textures[0]);
  glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  GL_REPEAT );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
  glTexImage2d(GL_TEXTURE_2D,    {Type de texture = 2D}
                    0,               {Niveau de détail}
                    GL_RGBA,         {Nombre de composants pour la couleur}
                    DimX,              {Largeur de la texture}
                    DimY,              {Hauteur de la texture}
                    0,               {Bordure [0 ou 1]}
                    GL_RGBA,         //Format de couleur. Nombre de composants pour
                      // décrire une couleur}
```

```
                    GL_UNSIGNED_BYTE,{Type de données dans le tableau de pixels}
                    Herbe.PtrPixels);       {Pointeur vers un tableau de pixels}
```

Aprés avoir initialisé la map, on va enfin tout dessiner en appelant tour à tour nos procédures.
Pour cette partie, on va avoir besoin d'une nouvelle librairie clBmp qui va nous servir à convertir une image du type BMP en un tableau de pixels, le format dont OpenGL a besoin. L'application d'une seule texture est quelque chose de facile, mais en appliquer plusieures est assez difficile du fait qu'il faut mémoriser la texture afin de ne pas avoir à la recharger tout le temps se qui ferait ralentir considérablement le jeu. Et donc grâce à cette librairie, on va pouvoir manipuler nos textures facilement.
Pour cela on va utiliser glGenTextures() ; qui va nous servir d'espace de stockage pour nos textures. On va pouvoir construire notre texture à l'aide de glTexParameteri() qui va nous servir à améliorer la qualité de la texture, et on va enfin entrer les données de l'image BMP à l'aide de glTexImage2D(). On aura ensuite plus qu'a activer la texture à l'endroit voulu avec glBind Texture(). On a enfin, notre map texturée.

1.1.2 Problèmes Rencontrés

Les problemes rencontrés sont l'application de plusieures textures qui est toujours difficile et l'utilisation subtile d'une matrice.
Pour la troisème soutenance, j'ai pour objectif de tout d'abord finaliser la map et bien sur de la modifier pour que cette map soit en 3D.

1.2 *Gestions graphiques* par Marc

1.2.1 Quoi de neuf docteur ?

Lors de la précedente soutenance qui était d'ailleurs la première, nous avions montré comment créer une fenêtre grâce à GLFW, y charger une texture, mais aussi y charger un modèle 3D au format 3DS. Or tout le code à été revu, maintenant des classes fenêtre et graphique ont été créés. Pour créer une fenêtre, rien de plus simple, soit la variable FFenetre du type jeu, on l'initialise grâce à l'instruction 'FFenetre := TFenetre.Create(parametres)' et nous avons ainsi créé une fenêtre OpenGL. Maintenant pour dessiner, il faut just faire FGraphic.draw et voila, on a une fenêtre sur laquelle on peut dessiner quasiment ceux que l'on veut. Il reste encore des choses à améliorer et des optimisations à faire mais l'essentiel est là, ça marche et c'est en objet.

1.2.2 Mais Encore ?

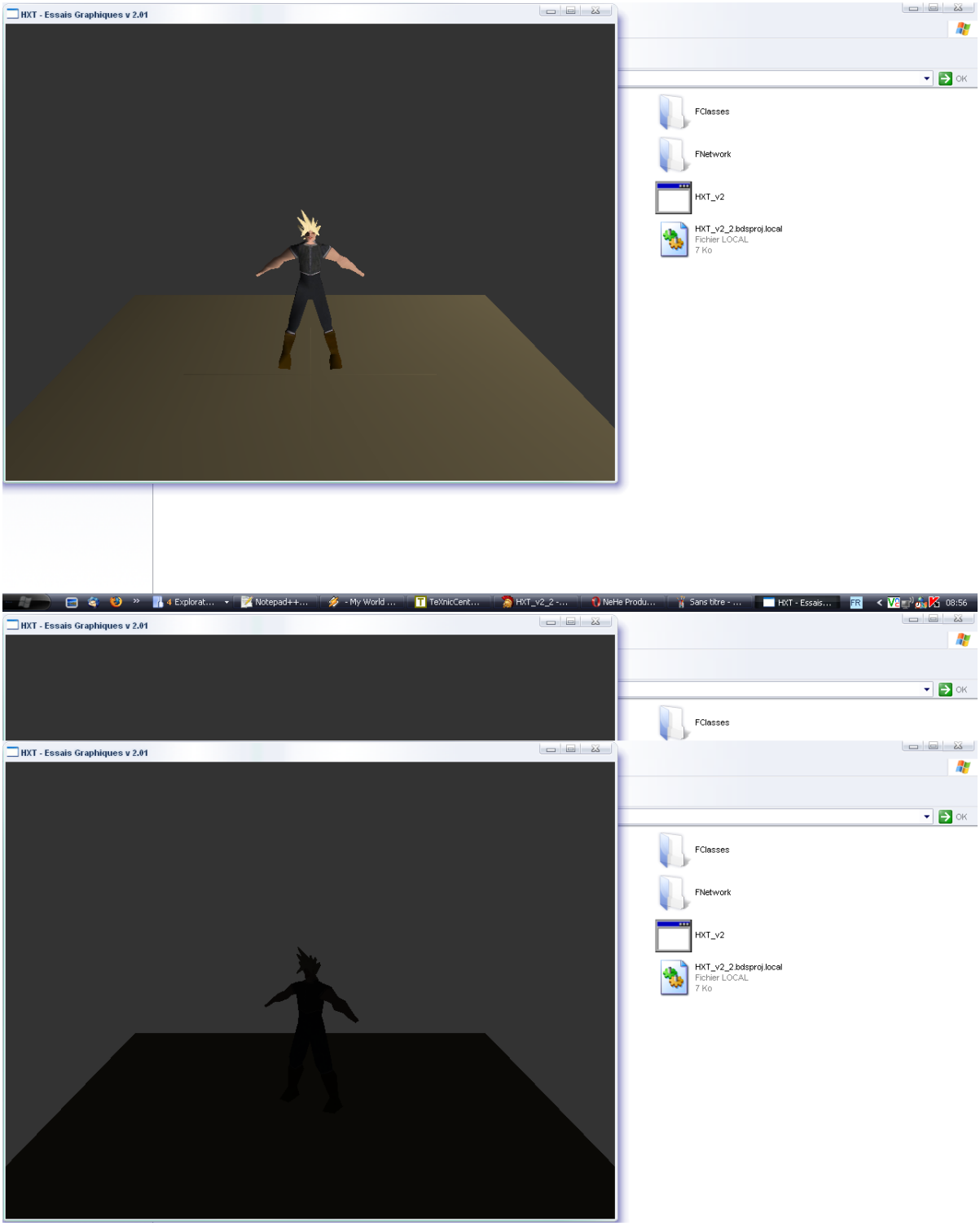
On utilise de l'objet, oui et alors. C'est simple, mais encore ? Car si on y dessine rien et on y applique aucun effet, cela reste inutile. Donc au final c'est bien beau de pouvoir dessiner en faisant seulement 'FGraphique.Draw' mais si c'est pour avoir un écran noir, ça reste nul. On va donc ainsi voir qu'elle sont les effets graphiques possibles que l'on peut appliquer sur la scène grâce à OpenGL.

Que la lumière soit

Et oui, grâce à OpenGL on peut créer des lumières (8 au maximum). Pour les utiliser il suffit de faire glEnable(GL LIGHTING). Mais attention, il existe différents types de lumière possibles :

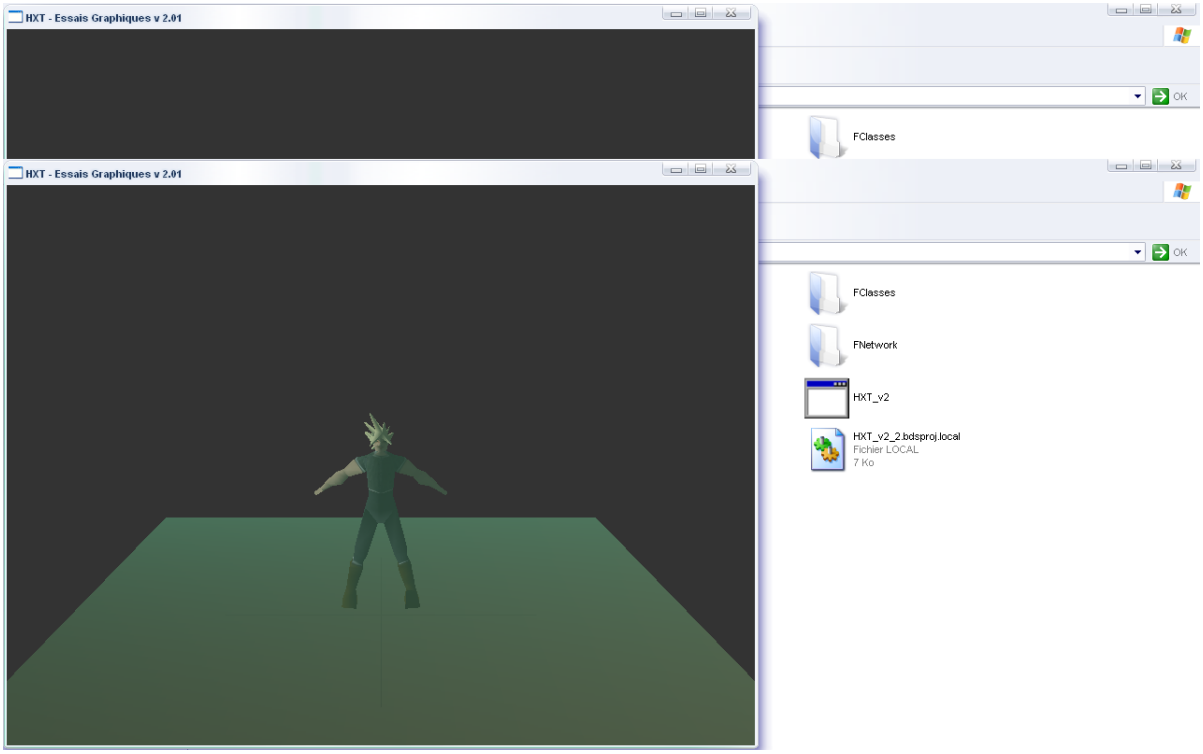
- La lumière ambiente : c'est un peu comme la lumière du soleil, elle se diffuse partout, et on ne sait pas d'où elle vient exactement.
- La lumière diffuse : c'est un peu comme la lumière de la lampe de poche avec un faisceau lumineaux qui s'élargit et s'atténue avec la distance.
- La lumière spéclaire : celle-ci serait plutôt consideré comme la capacité à refléchir la lumière. Par exemple, le verre à une lumière spéculaire plus élevé que le bois, sachant que le bois refléchit très peu la lumière.

Donc lors de la création d'une lumière il ne faut pas oublier de la configuer. Il est bien évidemment possible de faire un mélange des différents types de lumières cités plus haut. Mais, hélas, cela ne suffit pas pour utiliser les lumières. Cela aurait été trop simple sinon. Pourquoi faire simple quand on peut faire compliqué. En effet pour utiliser les lumières d'OpenGL correctement il faut penser à préciser commment les objets présents sur la scène vont réagir à la lumière et (car sinon c'est toujours trop facile) il ne faut pas oublier les normales des objets affichés qui, si elles ne sont pas configurées correctement, feront que les objet de la scène reflechissent la lumière d'une manière incompréhensible (à ne pas oublier).


HXT - Essais Graphiques v 2.01
FClasses
FNetwork
HXT_v2
HXT_v2_2.bdsproj.local
Fichier LOCAL
7 Ko
HXT - Essais Graphiques v 2.01
FClasses
HXT - Essais Graphiques v 2.01
FClasses
FNetwork
HXT_v2
HXT_v2_2.bdsproj.local
Fichier LOCAL
7 Ko

Je ne vois rien, c'est flou !

OpenGl c'est bien, car ça permet de faire du brouillard très facilement. Mais à quoi cela sert-il, pourquoi est-ce utile et comment l'utiliser ? Tout d'abord le brouillard permet de rendre un effet plus réaliste de la scène car une image trop net parait completement irréelle, notre vison devient plus flou avec la distance et le brouillard permet de faire cela. Mais le brouillard permet aussi de simplifier certain calcul au niveau du processeur ou de la carte graphique. Et pour l'utiliser, il n'y rien de plus simple, d'abord penser à faire glEnable(GL FOG) pour l'activer, quelques petits paramètres à modifier pour configurer le brouillard tel que le type de brouillard, la distance de départ, la distance d'arrêt (derrière, OpenGL ne dessine plus), la densité, la couleur et c'est à peu près tout. Rien de bien compliqué donc et très pratique.



Un effet très transparent si je puis dire. La transparence fait parti des effets graphiques que l'on peut faire avec OpenGL. Simple à activer, comme presque tout en OpenGL en fait, il suffit de faire glEnable(GL BLEND). Pourtant la transparence est beaucoup plus difficile à manipuler car lorsque OpenGL dessine les vertex, il dessine dans un certaine ordre, il faut donc faire attention à cette ordre pour obtenir au final l'effet de transparence désiré, sinon l'effet obtenu sera complètemnet différent de celui attendu. La transparence n'est pas simple à gérer car les objet possédant une transparence devront être traité différemments du reste, donc une mise en garde contre ceux-ci est de rigueur car de plus la transparence proposé par OpenGL n'est pas une réelle transparence mais un mélange de couleurs. Cependant, si ceux-ci sont correctement manipulés et à l'aides des effets de lumières, ont peu obtenir des effets très saisissants.

C'est déja fini

J'ai cité ici les options les plus utilisées d'OpenGL, mais il en existe d'autre et de très interessantes, tel que glCallList permettant de dessiner un objet qui a été enreregistrer sous une List GL ou l'option d'antialiasing qui permet d'éliminer l'effet d'escalier sur la surface de dessin et d'autres encore moins utilisés.

1.2.3 Ne parlons plus d'OpenGL ! Maintenant il faut faire quoi ?

Les effet graphique, c'est beau quand c'est bien implementé. On obtient des trucs super jolis à l'écran, mais ça fait surtout beaucoup plus de calculs à faire surtout si la map à affiché est gigantesque, donc il faudrait implémenter des algorithmes qui vont justement éliminer ces calculs inutiles gourmands en ressource mémoire. Il y a par exemple le BackFace Culling, le Frustrum Culling, les arbre BSP, les Octree ou encore les Quatree, et d'autres encore. Tout ces algorithmes permettent d'optimiser la vitesse d'affichage pour ne pas avoir un nombre de FPS trop faible. Il faudrait donc essayer d'en implementait certains pour les prochaines soutenances.

CHAPITRE 2

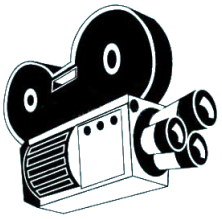
Moteur Physique

2.1 *Moteur physique* par Cedric

2.1.1 Introduction

Après avoir développé un moteur physique capable de détecter une collision avec un quadrilatère ou un triangle quelconque on pouvait se demander quelles seraient les évolutions les plus logique de se moteur. Fallait-il enchaîner directement sur une gestion basique de la 3D ou au contraire amméliorer le système existant ? L'arrivé de la programmation orientée objet (POO) au sein de l'équipe a réglé la question : la plupart de ce que nous avions fait pour la soutenance précédente était à refaire ou du moins à adapter grandement. C'est pourquoi la plupart des changements apportés à cette soutenance ne sont pas visible directement : le moteur a été partiellement réécrit et le résultat à l'écran est presque identique, à quelques détails près.

2.1.2 L'objet TCamera



Contrairement à ce que l'on pourrait croire, la caméra d'une scène OpenGL a plus de lien avec la partie physique qu'avec le moteur graphique. En effet, si l'on y regarde de plus près, qu'est-ce qui la caractérise? La position de l'oeil, la position du point que l'on regarde et la direction qui indique où est le haut.

Après, les valeurs des paramètres précédents ne dépendent que de calculs mathématique qui s'appuient sur les données du moteur physique. Auparavant, la fonction `gluLookAt` qui se charge de positionner la caméra prenait comme paramètres des variables globales où étaient stockées les coordonnées de l'objet à regarder (en l'occurrence la planche). Depuis la POO, il a fallut entièrement revoir la manière dont étaient stockées les informations et les coordonnées en question sont désormais propres à un objet TBoard qui représente la planche que l'on suit. Histoire de pouvoir utiliser plusieures caméras dans le jeu, j'ai créé un objet TCamera qui contient toutes les informations utiles pour configurer le positionement de la vue.

L'objet se décompose de la manière suivante :

```
TCamera = Class
  Public
   FCZ, FCRZ, Fps, FCRSpeed, FCSpeed, FProx, FLength: Single;
   procedure Init;
   procedure Reset;
   procedure setFps(nfps: Single);
   procedure RotateLeft;
   procedure RotateRight;
   procedure MoveUp;
   procedure MoveDown;
   procedure Show(Board: TBoard);
  End;
```

Comme on peut le voir, les fonctions possibles avec cet objet sont la remise à zéro de la position de la caméra, la rotation autour de l'objet, le déplacement en hauteur et l'assignement d'un objet à suivre.

2.1.3 Les Objets sur la map

La dernière version du moteur gérait les quadrilatères et les triangle quelconques. Il a donc été nécessaire de stocker ces éléments dans des objets. La structure objet est particulièrement adaptée pour les obstacles de la map. En effet, cela permet de leur donner un par un des paramètres personnalisés et de rendre la map encore plus variée. Par exemple, nous avons un objet TQuad dans lequel sont stockées ses coordonnées ainsi que les caractéristiques de chacune de ses faces (couleur ou texture). Chaque objet peut donc avoir son propre style : un batiment, un arbre, etc... L'objet TMap contient juste la taille de la map ainsi qu'une liste où sont stockés tous les objets de la map.

2.1.4 L'objet TBoard

Le plus important dans le moteur physique c'est l'objet que l'on déplace et qui est la cause de tous les tests de collisions. Etant donné que la partie réseau nécéssitera la présence de plus d'un Hoverboard sur le terrain, l'objet TBoard s'est imposé de lui même.
Ses composantes principales sont ses coordonnées en 3D mais également les indices de rotation sur les trois axes.
A partir de là, il a falloir faire un choix sur la forme physique que pourrait

prendre l'hoverboard. Vu qu'un hoverboard est plutôt rectangulaire, un tableau où sont stockées les coordonnées des extrémités de la planche fera l'affaire. A cela on ajoute une variable qui indique la hauteur de la planche et une classe permettant de savoir de quelle manière dessiner l'objet (relatif à la partie graphique).
Néanmoins si l'on observe la partie graphique, on découvre que les mouvements de la planche seront provoqués par une translation ou une rotation de la matrice. Cela signifie que le tableau où sont stockées les coordonnées de ses extrémités de la planche sera fixe et ce sera au niveau graphique que les mouvements seront appliqués. Or, il est indispensable pour les calculs physiques de savoir précisement à quel endroit par rapport au repère de la map sont situés chacune des extrémités de la planche. Nous avons donc besoin d'un deuxième tableau qui sera mis à jour à chaque mouvement.

Intéressons nous maintenant à la manière donc les mouvements de la planche seront calculés. A la dernière soutenance, un simple appui sur une touche provoquait un deplacement direct de la planche en modifiant ses coordonnées. Or, dans l'optique d'avoir un moteur physique qui se rapproche le plus proche possible de la réalité, il a fallu revoir entièrement la manière dont cela fonctionnait. Dans la réalité, tous les objets obéissent à un ensemble de forces dont la résultante détermine s'ils restent au repos ou non. Il restait donc à implémenter une gestion des forces pour la planche. Première question, où ces forces seront elles appliquées ? Dans la mesure où nous faisons un jeu de course arcade et non pas une simulation, il n'était pas nécessaire de pouvoir appliquer une forces sur un point quelconque de la planche. Ici, les extrémités suffisent. Et comme chaque extrémité sera amenée à recevoir plusieures forces en même temps il nous faut rajouter un tableau dynamique de vecteur à notre objet.

Les fonctions pour ajouter une force à une extrémité de la planche sont toutes simples et ne font qu'ajouter un vecteur de plus dans le tableau. Là, où le challenge est un peu plus relevé, c'est lorsqu'il s'agit de calculer la résultante des forces pour pouvoir faire avancer l'objet.

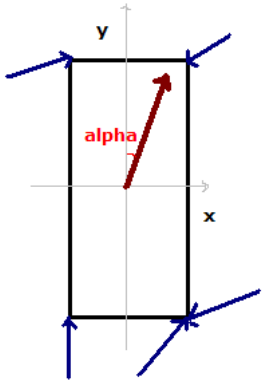


FIG. 2.1 – Exemple de forces appliquées à un objet

La résultante (en rouge sur le schéma) s'obtient en sommant tous les vecteurs entre eux grace à des calculs mathématiques. Mais comme les mouvements de la planche sont limités à une rotation et une translation il faut extraire de cette résultante les informations nécessaires pour pouvoir les traduire en ces deux mouvements. Ici, c'est les fonctions cos et sin qui viennent -comme toujours- nous porter secours.

Et ça marche !

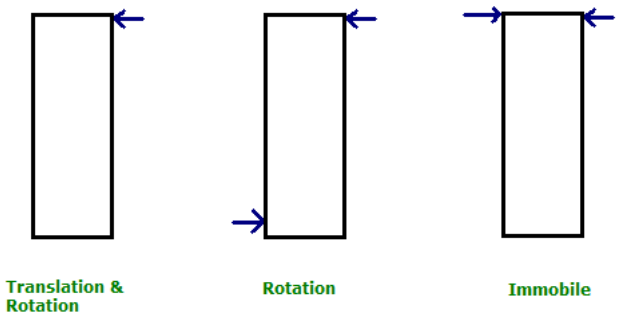


FIG. 2.2 – Réprésentation des mouvements suivant les forces appliquées

J'en ai profité pour intégrer le système d'accélération de Stéfano de la première soutenance directement dans l'objet ce qui rend les déplacement plus fluides et agréables.

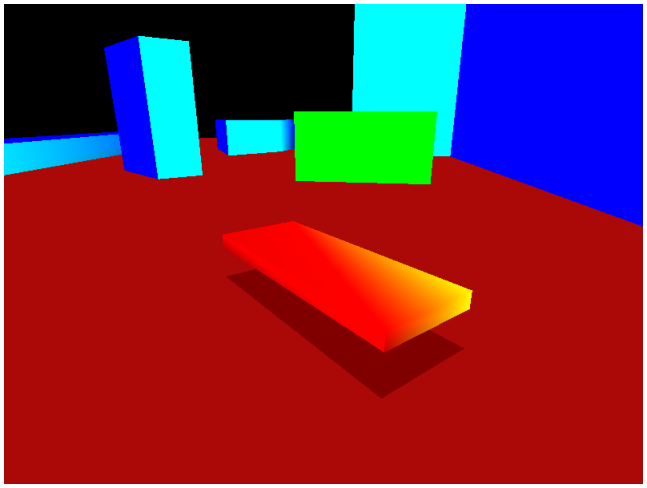


FIG. 2.3 – Capture d'écran de la dernière version du moteur

2.1.5 Ce qu'il reste à faire

Malheureusement, le temps ne m'a pas permis de terminer le moteur de collisions gérant les forces. Les calculs sont autrement plus compliqués qu'avec la version précédente puisqu'au lieu de s'arrêter net lorsqu'un point entre en collision avec un objet, il faut désormais que ce soit une force qui repousse le planche à l'extérieur. Néanmoins, cela ne devrait pas être trop difficile maintenant que la longue conversion vers de la POO a été effectué.
Dans l'avenir, il faudra également faire en sorte que la planche puisse grimper sur des objets et qu'il soit possible de sauter. Cela n'est qu'une extension du modèle de force de la 2D vers la 3D et ne constitue en rien un obstacle insurmontable.
La partie la plus pénible du travail a été effectué et à part la conversion du moteur de collisions vers les forces et une optimisation du moteur physique, rien ne devrait plus bloquer ma progression désormais.

CHAPITRE 3

Moteur de Jeu

3.1 *L'interface* par Stefano

3.1.1 Introduction

Dans un jeu il faut une interface graphique, ne serait-ce que pour le menu d'accueil, mais c'est toujours important de donner un accès à un menu quand par exemple on appuye sur pause pendant le jeu (et pas forcément le même que celui d'accueil).
L'idée était alors de créer un ensemble de classes capables de gérer les affichages et les différents évenements du menu, le menu étant évidemment en 2D.

Au stade actuel du développement de l'interface, seuls les boutons ont été implémentés, et ceux-ci devront être paramétrables, et pouvoir réaliser certaines actions lors d'évênements.
Voici une liste non exhaustive des actions et des évenements possibles que géreront les boutons :

- paramètre **taille** : largeur et hauteur du bouton.
- paramètre **position** : position horizontale et verticale.
- paramètre **couleur** : sélection de la couleur avec une chîne de caractères.
- paramètre **transparence** : réglage de l'opacité du bouton.

- évenement **OnClick** : gestion des actions à réaliser quand on clique sur le bouton.
- évenement **OnHover** : gestion des actions à réaliser quand on passe sur le bouton avec le curseur.

Pour l'implémentation de cette classe de manière simple il m'a semblé logique dès le début d'utiliser un type FIFO. J'ai donc du refaire le type file chaîné (tant qu'à faire). Si j'ai choisi un type FIFO c'est parce qu'il est plus pratique de pouvoir afficher le dernier élement ajouté dans le cas d'une superposition d'élements.

3.1.2 La classe TMenuBouton

Comme son nom l'indique, la classe TMenuBouton sert à créer des objets "boutons".
Cette classe contient sept champs et quatre méthodes.

Les champs de la classe

Voici les différents champs que l'on peut trouver dans cette classe :

```
width, height : integer;
x, y : integer;
points : array [1..4] of array [1..2] of integer;
color : array [1..4] of Double;
lbl : string;
```

Width et Height

Width et Height sont deux variables de type entier, elles servent à définir la largeur et la hauteur du bouton (qui est rectangulaire). On les utilise lors de la construction de la classe.

x et y

x et y servent à donner la position dans le repère 2D, par contre il faut bien songer que l'origine (0, 0) est en bas à gauche (et non pas en haut à gauche) dans notre cas. C'est en fait la coordonnée du point bas gauche du "rectangle" qui représente le bouton.

points

La matrice points est une matrice de 4*2, dans laquelle on stocke les coordonnées de chacun des points du rectangle qui représente le bouton. On pourrait bien évidemment utiliser un tableau simple à une dimension et à deux entrées, mais cela nous obligerait à calculer les coordonnées à chaque tour, et pour le peu de place que ça prend en plus (comparé au programme en entier), j'ai choisi de privilégier ce petit gain de vitesse.

color

Le tableau de quatre réels color, est celui qui nous permettra de stocker la couleur du bouton, avec les paramètres RVB (Rouge Vert Bleu). On remarquera que si la couleur s'utilise avec trois paramètres (rouge, vert et bleu), le tableau color contient quatre champs, car on gèrera aussi la transparence du bouton.

lbl

lbl, comme label, sauf que label est un mot reservé par Delphi. Lbl contient donc tout simplement un identifiant du bouton, sous la forme d'une chaîne de caractères.

Les méthodes de la classe

Voici les différentes méthodes que l'on trouve dans la classe TMenuBoutons :

```
procedure SetButtonColor (couleur : string);
procedure SetButtonTransparence(t : double);
procedure Display(x, y : integer);
function AnsiIndexStr(AText : string; const AValues : array of string) : integer;
```

SetButtonColor et AnsiIndexStr

Actuellement les boutons ne gèrent pas encore les textures, il faut donc au moins leur donner une couleur, sinon on aura du mal à les différencier. Grâce à cette procedure, on peut définir la couleur à l'aide d'une chaîne de caractères.
OpenGL n'utilise pas une chaîne de caractères pour la gestion des couleurs, alors il faut donner des valeurs aux champs du tableau color, afin que les valeurs de celui-ci correspondent à la couleur et pour qu'on puisse les réutiliser afin de gérer la bouleur du bouton.
La solution évidente serait un case, mais Delphi ne les gère que pour les entiers, d'où la fonction AnsiIndexStr.

La fonction AnsiIndexStr prend en paramètre une chaîne de caractère, et un tableau de chaînes de caractères, puis effectue une recherche dans le tableau afin de déterminer à quelle position se trouve la chaîne passée en premier paramètre, puis retourne cette valeur, ou -1 si elle ne la trouve pas.
Dès lors on peut utiliser un case avec cette fonction, ce qui nous donne ceci :

```
procedure TMenuBouton.SetButtonColor (couleur : string);
 begin
   case AnsiIndexStr(couleur, ['red', 'blue', 'green']) of
     0 : begin
           color[1]:= 1;
           color[2]:= 0;
           color[3]:= 0;
         end;

     1 : begin
           color[1]:= 0;
           color[2]:= 0;
           color[3]:= 1;
         end;

     2 : begin
           color[1]:= 0;
```

```
          color[2]:= 1;
          color[3]:= 0;
        end;

    -1 : begin
           color[1]:= 1;
           color[2]:= 1;
           color[3]:= 1;
         end;
  end;
 end;
```

Pour l'exemple c'est limité à 3 couleurs, mais il est tout à fait possible d'en ajouter très simplement.
De plus on comprend l'utilisation du tableau color, en premiere case on a la proportion de rouge, en deuxième celle du vert et en troisième celle du bleu, chacune de ces proportions étant des réels, compris entre 0 et 1.

SetButtonTransparence

La procédure SetButtonTransparence est un peu la continuité de SetButtonColor, dans le sens où son action s'effectue aussi sur le tableau color, mais cette fois sur le quatrième champs, qui représentera donc la "transparence" du bouton. Encore une fois il s'agit d'un réel compris entre 0 et 1.
La procedure pour l'ajout est toute simple :

```
if (t<=1) and (t>=0) then color[4]:= t;
```

Display

La procdure Display affiche le bouton, elle prend en position l'endroit auquel il doit être affiché, c'est à dire les champs x et y cités ci-dessus.

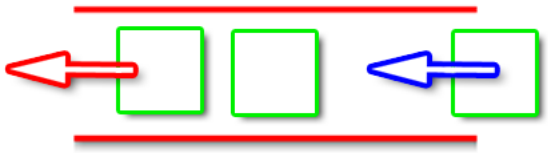
Voici à quoi ressemble cette procedure :

```
    procedure TMenuBouton.Display(x, y : integer);
begin
  glColor4f(color[1], color[2], color[3], color[4]);
  glBegin(GL_QUADS);
    glVertex2i(points[1][1]+x, points[1][2]+y);
    glVertex2i(points[2][1]+x, points[2][2]+y);
    glVertex2i(points[3][1]+x, points[3][2]+y);
    glVertex2i(points[4][1]+x, points[4][2]+y);
  glEnd;
end;
```

Comme on le voit, la procédure display met en place la couleur et la transparence du bouton (avec glColor4f), puis affiche le rectangle correspondant au bouton en ajoutant x et y aux coordonnées des quatre points du rectangle.
A noter que la matrice points et le tableau color sont initialisés lors de la construction du type, le tableau color initialisé de manière à ce que par défaut la couleur soit à blanc et le bouton soit opaque, et la matrice points des manière à ce que le point en bas à gauche du rectangle correspondant au bouton soit au point (0, 0).

3.1.3 Le type File

Pour implémenter un nombre n de TMenuBoutons, il me fallait une structure de données qui me permette de stocker un certain nombre de variables (tableau, liste, file...), et mon choix s'est porté sur les files pour les raisons déjà exposées. Notion déjà vue en cours, la file est une structure de données de type FIFO (First In First Out), c'est à dire que les éléments sont ajoutés les uns à la suite des autres, puis quand on les utilise, on les récupère dans l'ordre d'ajout (premier ajouté est le premier récupéré).



On voit sur le schéma avec une flèche bleue à droite une donnée (carré vert) qui rentre dans la file (le tuyau rouge), et avec la flèche rouge, une autre donnée, qui sort. La donnée qui est entre celle qui sort et celle qui rentre sera la prochaine à sortir, puis viendra le tour ce celle qui est en train d'entrer.

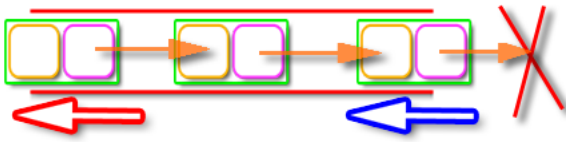
Le problème qui se posait, c'est qu'en cours on n'arretait pas de nous parler des listes chaînées, des files chaînées, des pointeurs et des choss comme ça, mais sans pour autant nous dire ce que c'était, ni comment ça fonctionnait, juste des "ça utilise beaucoup moins de mémoire" ou encore des "c'est plus propre", alors je me suis dit "pourquoi ne pas leur faire confiance ? Essayons !". Bref, tout ça pour dire que pour mon interface j'ai du implémenter un type File, qui de plus est chaîné.

Principe

Le principe de la file à déjà été rapidement vu ci-dessus, maintenant il faut voir comment implémenter cette file de manière dynamique, c'est à dire en évitent d'allouer un certain nombre de places dans la mémoire alors qu'on n'est même pas sur de devoir toute les utiliser, ou pire, dont on risque d'en manquer. On va donc créer une file de d'objets de classe TMenuBouton, et ce à l'aide de pointeurs.

Les pointeurs sont des variables, mais qui contient des adresses mémoire, dans lesquelles se trouvent des variables. Ces variables pointées par le pointeur se trouvent dans la partie dynamique de la mémoire de l'ordinateur, qui peut contenir jusqu'à 4 milliards d'octets. Si j'avais créé le type file en statique, chacune des variables aurait été stockée dans la partie statique de la mémoire, qui elle est beaucoup plus limitée.

Un pointeur a donc la particularité de pointer vers une variable, et le grand avantage est que cette variable peut être d'un type structuré, et c'est justement grâce à cette possibilité de pointer vers un type structuré que l'on va pouvoir faire un type File chaîné, car devinez ce que l'on peut mettre comme variable dans un type structuré ? Je vous le donne dans le mille : un pointeur !
C'est fou non ? Et bien accrochez vous car ce n'est pas fini, vous allez enfin découvrir la puissance d'un pointeur ; selon vous, mon pointeur dans mon type structuré, sur quoi il peut pointer ? Ne laissons pas le suspense monter plus longtemps, la réponse est : vers une variable de ce même type structuré ! Et oui, et à partir de là on comprend aussi le sens du mot chaîné, les pointeurs permettant de relier entre elles différentes variables.



On reconnait notre file de tout à l'heure, avec toujours la sortie indiquée par la flèche bleue, et l'entrée par la flèche rouge, mais cette fois c'est une file d'élements de types structurés, dont une des variables de ce type est un pointeur, qui pointe vers une autre variable de type structuré, ainsi la première (qui est celle qui se trouve au niveau de la flèche rouge) point vers la deuxième, qui elle même pointe vers la troisième (celle au niveau de la flèche bleue, et cette dernière ne pointe sur rien du tout.

Implémentation

Déclaration du type File

```
type
  PFileElem = ^TFileElem;
  TFileElem = record
    Elem : TMenuBouton;
    Suivant : PFileElem;
end;
```

On a donc un type TFileElem, qui est une file d'élements, ces élements étant un type structuré qui contient un objet de classe TMenuBouton, et un poin-

teur qui pointe vers une variable de type TFileElem, et qui donc pointera vers l'élement ajouté juste après dans la file.

Il faut maintenant implémenter les différentes routines à utiliser avec ce type. Il y a 4 routines associées à ce type :

```
function FLNouvelle : PFileElem;
function FLVide(F : PFileElem) : boolean;
function FLDefiler(F: PFileElem) : PFileElem;
function FLTete(F: PFileElem) : TMenuBouton;
procedure FLDestroy(F: PFileElem);
```

FLNouvelle

La fonction FLNouvelle ne prend pas de paramètre et retourne un pointeur qui ne pointe vers "rien", mais qui pourra dès l'ajout d'un élement (de type PFileElem) pointer vers celui-ci afin de créer la file.
Le code de cette fonction se résume à :

```
result:= nil
```

Pour indiquer qu'un pointeur pointe vers "rien", en Delphi on lui attribue la valeur nil.

FLVide

La fonction FLVide prend en paramètre une file et nous indique si elle est vide à l'aide d'un booléen (Vrai pour vide et Faux pour non-vide.
Cette fonction est très simple :

```
result:= (F = nil);
```

Encore une fonction très simple, on retourne simplement le test "la pointeur ne pointe vers rien", si il ne pointe vers rien, dans ce cas il n'y a pas d'élément dans la file, on peut retourner vrai, sinon on retourne faux.

FLDefiler

La fonction défiler retire la "tête" de la file, la tête devient alors l'élement qui se trouvait en deuxième position dans la pile, et on ne peut plus acéder à l'élement enlevé.

```
function FLDefiler(F: PFileElem) : PFileElem;
begin
  if not FLVide(F) then
  begin
    result:= F^.suivant;
    dispose(F);
    F:= nil;
```

```
    end
    else result:= F;
  end;
```

La fonction FLDefiler, retourne F^.suivant. On a vu que dans le type TFileElem, il y a la classe TMenuBouton, mais aussi le pointeur suivant, qui pointe vers l'adresse de la variable qui se trouve à la suite dans la file. L'utilisation F^permet d'accéder à l'élement sur lequel pointe F, et suivant, est un pointeur qui pointe vers le deuxième élement de la file, et c'est justement ce qu'on désire obtenir quand on dépile, on retourne donc F^.suivant. Mais le pointeur qui a été passé en paramètre continue d'exister, il faut donc supprimer ce qu'il pointe en faisant dispose(F)(seulement après avoir récupéré l'adresse de l'élément suivant), mais aussi lui attribuer la valeur nil, et cela car une fois la variable pointée supprimée, le pointeur continue de pointer à cette adresse, et si par mégarde on venait à réutiliser le pointeur qui pointe vers cette adresse à laquelle il n'y a plus rien, on obtiendrait littéralement n'importe quoi.
Dans le cas ou la file est déjà une file vide, il n'y a rien à dépiler, on le retourne tel quel (on pourrait tout aussi bien retourner nil).

FLTete

La fonction FLTete prend en paramètre une file, et retourne un objet de classe TMenuBouton, c'est à dire qu'on récupère la valeur de la variable qui se trouve en tête de la file.

```
if not FLVide(F)  then result:= F^.Elem;
```

Si la file n'est pas vide, dans ce cas on retourne l'élement pointé (c'est à dire un objet de classe TMenuBouton).

FLDestroy

La procédure FLDestroy s'occupe de supprimer la File, c'est à dire d'effacer de la mémoire chacune des variables se trouvant dans la file.

```
procedure FLDestroy(F: PFileElem);
var
  tmp, tmp2 : PFileElem;
begin
  tmp:= F;
  while tmp<>nil do
  begin
    tmp2:= tmp^.suivant;
    dispose(tmp);
    tmp:= tmp2;
  end;
end;
```

Afin de détruire tous les élements pointés, on effectue une boucle qui récupère chacun des élements de la file, puis détruit les élements pointés (en ayant pris soin avant de récupérer l'adresse de l'élement suivant dans une variable temporaire).

FLEnfiler

Pour ajouter des élements à la file, on utilise la fonction FLEnfiler, qui retourne un pointeur vers la file.
On lui passe en paramètre une file à laquelle ajouter l'élement, et l'émenet à ajouter.

```
function FLEnfiler(F : PFileElem; B : TMenuBouton ) : PFileElem;
 var
   tmp, parcours : PFileElem;
 begin
   new(tmp);
   tmp^.suivant:= nil;
   tmp^.Elem:= B;

   if F = nil then
   begin
     result:= tmp;
   end
   else
   begin
     parcours:= F;
     while parcours^.suivant <> nil do
     begin
       parcours:= parcours^.suivant;
     end;
     parcours^.suivant:= tmp;
     result:= F;
   end;
 end;
```

Pour enfiler, il faut bien ajouter l'élement à la fin de la file, pour cela on effectue une boucle jusqu'à la fin de la file, et une fois qu'on y est, on attribue une variable à pointer au pointeur qui pointe vers "rien" (nil), cette variable ayant été préalablement créée (en lui ayant attribué un pointeur qui pointe vers nil, puisqu'elle est ajoutée à la fin de la file, et comme élement un objet de classe TMenuBouton).

3.1.4 La classe TFiledeboutons

C'est bien gentil d'avoir implémenté un type file, mais encore faut il s'en servir. Comme ça a été vu plus haut, notre type file contient des boutons, et ces boutons ont des propriétés, mais ce serait bien de pouvoir tout gérer grâce à un objet, et justement, la classe TFiledeBoutons est là. Elle contient à l'heure actuelle du développement, un champ et deux methodes.

```
Elts : PFileElem;
procedure afficher(F:PFileElem);
procedure ajouter(width, height, x, y : integer; transparence : double; couleur :
```

Description

La classe TFiledeboutons contient un champ Elts, qui est la file des TMenu-Boutons.

afficher

‘

La procédure afficher parcourt la file afin d'afficher tous les boutons (à l'aide de la procedure Display de la classe TMenuBouton). Ce qui donne :

```
procedure TFiledeboutons.afficher(F:PFileElem);
begin
  if not FLVide(F) then
  begin
    FLTete(F).Display(FLTete(F).x, FLTete(F).y);
    afficher(F^.Suivant);
  end;
end;
```

Le fonctionnement est très simple, on affiche l'élement de tête, aux coordonnées avec lesquelles on aura "préparé" le bouton, puis on rapelle la fonction pour qu'elle effectue la même chose sur l'élement suivant de la file, jusqu'à ce qu'on ait parcouru tout la file.

ajouter

On a maintenant notre file de boutons, nos boutons, il ne manque plus qu'à les ajouter, pour cela la procédure ajouter fait son apparition (contrairement à l'epichat qu'on ne voit plus :( ).

```
procedure TFiledeboutons.ajouter(width, height, x, y : integer; transparence : double; c
var
  tmp : TMenuBouton;
begin
  tmp:= TMenuBouton.Create(width, height);
  tmp.x:= x;
  tmp.y:= y;
  tmp.SetButtonTransparence(transparence);
  tmp.SetButtonColor(couleur);
  Elts:= FLEnfiler(Elts, tmp);
end;
```

Cette fonction prend pas mal de paramètres, j'avais le choix entre faire en plusieurs fois, ou une seule, et il m'a semblé plus pratique de le faire en une seule, donc je l'ai fait (car j'aime bien m'écouter).
Donc encore une fois rien de bien compliqué, on créé le nouvel objet bouton, on lui attribue les différentes valeurs qu'on a prises en paramètre, et puis pour finir on l'enfile.

3.1.5 Utilisation

Maintenan l'utilisation est très simple, il faut tout d'abord, avant d'entrer dans la boucle de jeu créer un objet de classe TFiledebouton, puis définir des boutons avec la procédure ajouter, ce qui donne quelque chose comme ça :

```
boutons:= TFiledeboutons.Init;
  boutons.ajouter(500, 500, 30, 30, 1, 'green');
```

Ensuite il ne faut pas oublier d'appeler la procedure d'affichage de la file de boutons dans la boucle procédure dédiée aux affichage.

```
boutons.afficher(boutons.Elts);
```

Et voila, on obtient alors un "bouton" vert, de taille 500*500 pixels et à 30 pixels du bas de la fenêtre et à 30 pixels du bord gauche de la fenêtre.
*

3.1.6 Conclusion

L'interface est loin d'être terminée, pour l'instant elle ne fonctionne que sur un repère bidimentionnel (alors que le jeu se passe dans un repère tridimentionnel.
De plus il reste encore pas mal de fonctions à implémenter, notemment les évenements OnHover et OnClick, mais aussi des propriétés de "decoration" du bouton, comme par exemple l'ajout d'une bordure (et pourquoi pas plusieurs types de bordure : thin, double, solid, dashed...).

3.2 *La gestion du son* par JR

3.2.1 FMOD, librairie sonore

La partie sonore ma beaucoup intéressé, elle est plus complexe que prévu, c'est en fait sur cette partie que j'ai passé le plus de temps. Sans grande surprise j'ai choisi d'utiliser la librairie FMOD, une API gratuite et qui gère de manière assez complète tout ce qui touche l'environnement sonore d'un jeu. La musique et les bruitages du jeu sont donc implémentés grâce à FMOD. Cette librairie nous offre une palette d'outils simples et performants à la gestion des effets sonores. Elle supporte entre autre les formats mp3, wav et midi.

Après une longue recherche, j'ai trouvé comment déclarer les types Tsample nécessaire au lancement d'un son.

```
FSOUND_Init(44100, 32, 0) ;
fond_sonore:=FSOUND_Stream_Open('roky.mp3',FSOUND_NORMAL,0,0);
FSOUND_Stream_Play(1,fond_sonore);
```

J'ai maintenant pour objectif de charger plusieurs bruitages, avec plusieures musiques selon les niveaux du jeu, dès que notre personnage touchera un objet il y aura un bruitage qui se declarera.

3.2.2 Musique et bruitages

Le jeu devait contenir un environnement sonore afin que l'on puisse accompagner la glisse de notre hoverboard d'effets audio et mettre en fond une musique.

De l'initialisation à la lecture

L'initialisation se fait tout simplement à l'aide de la procédure FSOUND_Init. Celle ci prend compte le nombre de plages existant dans le projet.

La musique et les bruitages sont deux effets sonores bien distincts. La structure FSOUND_STREAM est utilisée pour la musique et la structure FSOUN0D_SAMPLE pour les bruitages. Chacune de ces structures possède ses propres routines. Il suffit d'une seule procédure pour charger une musique, que l'on peut lancer en boucle.
Toute musique et tous bruitages chargés sont libérés de la mémoire à la fermeture de jeu.

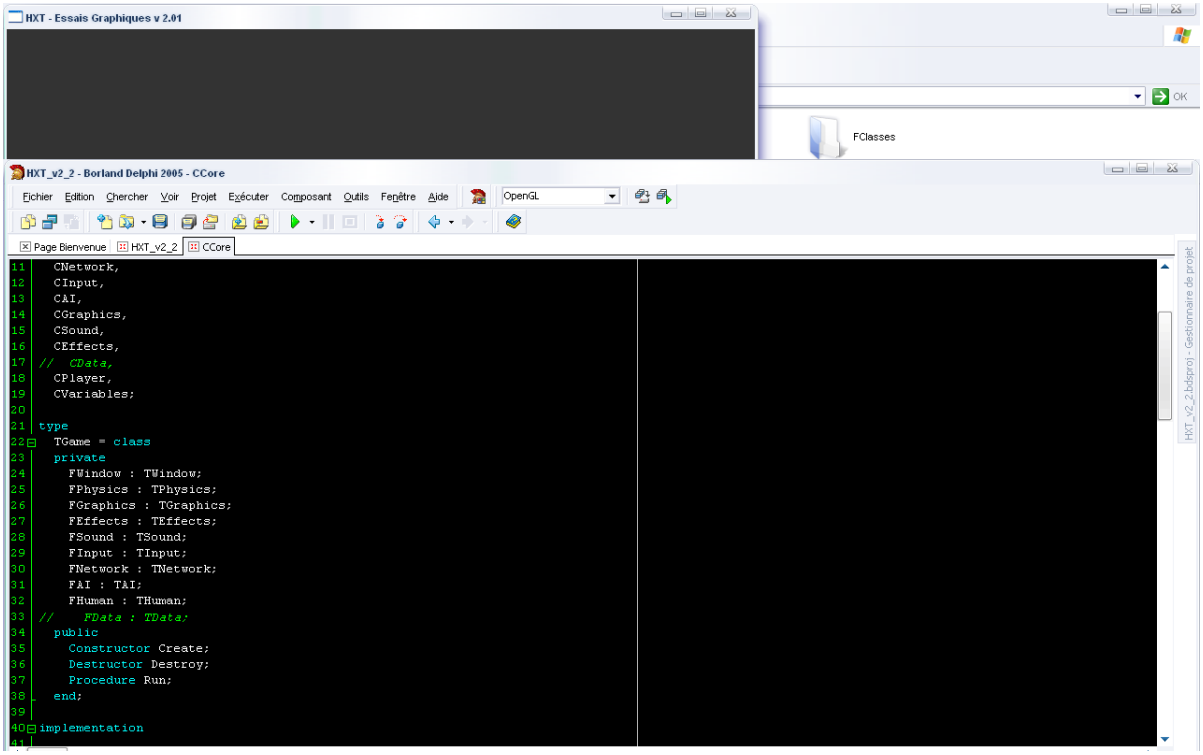
Une musique est lue avec FSOUND_Stream_Play. Un bruitage est lu avec FSOUND_PlaySound. Lorsque une musique se termine on en relance une autre.

Chaque musique a pour but de créer une ambiance associée au jeu. C'est pourquoi nous avons cherché à savoir quelles sensations nous voulions transmettre au joueur durant une partie de manière à ce qu'elle corresponde le mieux à l'esprit HOVERBOARD Xtrem.

3.3 *Le Noyau du Jeu* par Marc

3.3.1 Présentation

Le Noyau, c'est le coeur même du jeu, la partie qui regroupe toute les autres en une seule. C'est grace à ce fichier qui rassemble toute les autres unités au sein d'une même d'une partie que notre programme pourra fonctionner correctement. Le noyau va executer, en faisant appel aux instructions des différentes unités chacune des commandes du jeu et ainsi permettre le bon déroulement du jeu. Le Noyau exécutera donc ainsi de suite les commandes des unités de réseau, d'IA et d'Input, puis la Physique et enfin le Graphqiue et tout cela dans la boucle principale jusqu'a l'arrêt du jeu. Il est donc très important que les différentes unités soient écrites le plus proprement possible (plus c'est propre, mieux c'est, mais ce n'est pas une tache facile qu'il nous à été donné d'accomplir, mais nous sommes là pour apprendre, non ? !), il faut que cela soit simple à comprendre et rapidement accessible car lors des modifications ultérieures il ne faut que l'on n'ait à tout relire et chercher quelles fonction utiliser pour faire telle ou telle chose, ne pas oublier de le commenter non plus car si c'est pour essayer de relire un code non commenté et mal écrit 2 mois après, alors autant le réecrire de puis le début, le travail sera mieux fait. Tous cela pour dire finalement que pour cette seconde soutenance, nous avons commencé à codé tous le projet en POO (càd Programmation Orientée Objet). Des Classes ont ainsi été écrites dans les différentes unités pour faciliter la manipulation des données. Il est beaucoup plus simple, pour faire avancer un joueur de écrire : 'joueur.personnage.marcher' que de passer par des tas de fonctions situés un peu partout et dont on ne se rappelle plus de leur réelle utilité pour finalement modifer la place du joueur. La POO ouvre donc de grandes possibilités mais permet aussi d'imposer certaine restrictions, ce qui ne peut être que bénéfique si le code est bien écrit et utilisé à bon escient dans des conditions optimales (mais là „je vais un peu loin alors que nous ne sommes encore qu'amateurs). Grâce à la POO, on a donc créé les types TJeu, TFenetre, TInput, TGraphic, TPhysique, TPlayer, TCamera et plein d'autre encore mais ils sont encore incomplets.



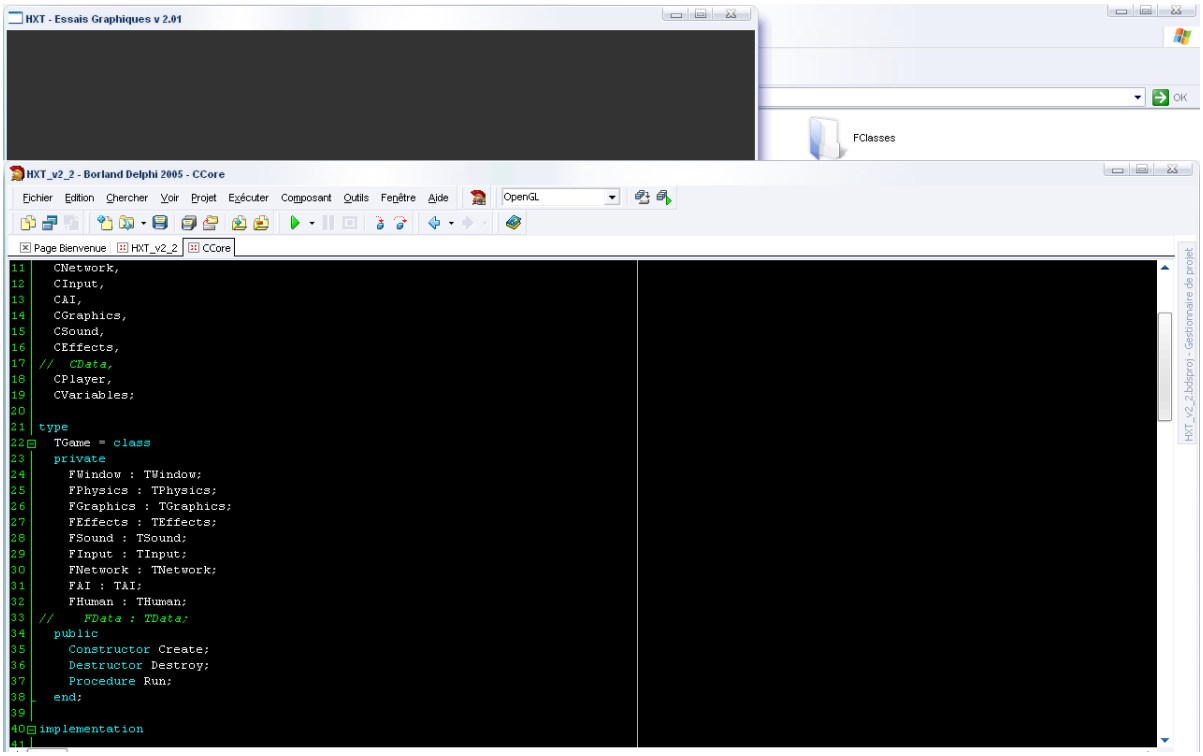
3.3.2 Problèmes Rencontrés

Un certain nombre de problèmes ont été rencontrés lors du passsage à la l'Objet, notamment lors des sessions de déboguage qui se sont averéés beaucoup plus complexes. Les bugs et erreurs sont bien plus difficile à repérer et mieux dissimulés surtout, d'ailleurs certaines erreurs lorsqu'on les remarque s'avère en réalite très futiles ("comment ai-je fait pour ne pas l'avoir vu celle-là, ca n'est pas possible", l'erreur stupide qui te fait perdre une heure pour rien, enfin bref). Alors que d'autres sont beaucoup mieux dissimulés et l'on ne se serait même pas douté que ceux-ci pouvait être des erreurs empechant la compilations de l'exécutable. Cependant avec l'habitude les erreurs sont moins fréquentes et mieux repérables mais ca n'est pas toujours le cas malheuresement.

3.3.3 Résultat

Cependant le résultat obtenu après un travail de dur labeur est plutôt impressionnant. Les données sont facilement manipulables et plus simple d'accès. Leur utilisation est de plus beaucoup plus intuitive et la compréhension s'en trouve grandement facilitée. De plus le code en lui même est plus propre et plus optimisé. On a tout de même le type Jeu qui possède 3 méthode dont un constructor Create qui va initialiser en mémoire tous ce dont le jeu à besoin, une fonction Run qui execute réellement le jeu en lui même et un Destructeur Destroy qui va liberer la mémoire et fermer l'application. Ce type fait bien sûr appel à d'autres objets qui ont aussi été créés pour facilité la manipulation de

données et la compréhension du code. Nous pouvons donc dire que le noyau est plutôt bien avancé mais encore loin d'être terminé.



3.3.4 A Venir

Il reste tout de même un long travail à faire car toutes les classes ne sont pas encore totalement terminées ou écrites, certaines d'ailleurs n'ont même pas été créées, et ne seront complètes seulement lorsque toute les parties du projet seront réunies, ce qui est encore loin d'être le cas. Cependant pour la prochaine soutenance il faut terminer autant de classe possible, créer celles dont ont a besoin et qui ne le sont pas encore et si possible améliorer le code, le rendre plus clair et plus optimal.

CHAPITRE 4

Le site web

4.1 Site par Cédric

4.1.1 Introduction

Dans la mesure où nous devons conduire un projet informatique du début à la fin, notre démarche est similaire à celles que nous serons amenés à poursuivre dans la vie professionnelle. La principale différence est qu'au lieu de suivre celle-ci à des simples fins éducatives, le principal objectif des developpements futures sera d'avoir un produit finit qui tienne la route commercialement parlant. Or, même si l'on sait qu'on a le meilleur produit du monde, il est de renommée publique que la bonne foi du vendeur ne suffit pas pour faire vendre! C'est pour cela qu'il ne faut surtout pas négliger l'aspect marketing du projet afin d'éviter que le produit finit soit mis sur le marché dans l'ignorance générale des acheteurs potientiels.

Voilà pourquoi j'ai tenu à mettre l'accent sur ce qui pourrait être considéré comme négligeable et secondaire face au reste du projet : le site web. En effet, avec la forte importance d'internet dans la société actuelle, avoir un site web pour présenter son produit est devenu un passage obligatoire dans n'importe quel plan commercial. Dans un processus de "fabrication" traditionnel, l'aspect développement et marketing sont pris en charge par deux équipes différentes pour la simple et bonne raison que les compétences requises pour programmer un jeu et trouver un slogan accrocheur ne sont pas les mêmes.
Cependant, même si le travail d'un developpeur n'est pas de faire du marketing et que la vocation initiale de notre site web est juste de présenter le projet tout en offrant la posibilité de le télécharger, j'ai tenu à avoir une approche plus générale du site et à lui donner toute l'importance qu'il mérite vis à vis du jeu. Il a donc été conçu dans une optique d'attirer l'oeil du visiteur, de le pousser à en savoir plus sur HOVERBOARD Xtrem, le jeu.

4.1.2 L'aspect technique

Avant de se lancer tête baissée dans le code, il convient de choisir auparavant judicieusement les technologies à employer pour qu'il puisse correspondre à nos attentes.
Les objectifs du site sont les suivants :

- Fournir au visiteur le maximum d'informations sur le projet :
  Présentation du jeu et de l'équipe
  Montrer l'avancement du développement au jour le jour
  Proposer en téléchargement le cahier des charges, les divers rapports, les éxécutables du jeu mais aussi son code source
  Afficher un ensemble de liens vers les sites proposant des ressources (images, sons, librairies, etc...) que nous aurions pu utiliser
- Rendre le jeu le plus attrayant possible
- Donner envie au visiteur de revenir en lui offrant une navigation confortable et visuellement riche

Au regard de cette liste, nous pouvons d'ors-et-déjà commencer à choisir les outils les plus adaptés. Comme nous souhaitons pouvoir mettre régulièrement à jour le site afin de pouvoir fournir à nos fans assoiffés les dernières informations concernant l'avancement du projet, la solution la plus répandue (et surtout gratuite) est celle du langage PHP s'appuyant sur une base de données MySQL.

Cela aurait pu suffire mais il aurait été difficile de remplir le dernier objetif avec une simple page statique. Dans ce cas, la technologie Flash pourrait être une solution puisqu'elle permet d'afficher des animations et des éléments graphiques et sonore avec une aisance imcomparable. Seulement, pour créer une animation flash, il faut passer par les logiciels propriétaires et la plupart du temps hors de prix.
Tenant à rester dans le domaine **légal**, je me suis donc tourné vers la technologie dont la popularité ne cesse d'augmenter en ce moment : l'AJAX!

Qu'est ce que cet étrange animal? Tout simplement un raccourci bien pratique pour parler à la fois de "HTML (ou XHTML) et CSS pour la présentation des informations, DOM et JavaScript pour afficher et interagir dynamiquement avec l'information présentée et enfin XML et l'objet XMLHttpRequest pour échanger et manipuler les données avec le serveur web". Tout un programme! Derrière ces grands mots se cache la possibilité de mettre à jour l'affichage d'une page sans avoir à la recharger depuis le serveur. Les éléments de celle-ci comme les images ou le texte ainsi que les propriétés respectives (position, taille, etc...) peuvent être mis à jour à tout moment via le langage proposé avec tous les navigateurs internet : Javascript. Enfin, avec le joli nom de domaine facile à retenir (**www.hxt.fr**) obtenu grace à Stefano, nous avons tous les outils en main pour proposer à notre futur fan un agréable voyage...

4.1.3 Visite page par page

Lorsque que le visiteur arrive sur le site il se retrouve face à la page d'accueil. C'est donc elle qui va faire office "d'accroche" -tout comme le slogan de Une des journaux- et qui va devoir être la plus intéressante possible pour lui donner envie de rester surfer plutôt qu'aller faire du parapente dans les Pyrénnées. Son importance est donc capitale !
Pour cela, j'ai opté pour un design "en blocs". C'est à dire que tous les éléments se trouvent dans un cadre afin de bien délimiter les zones d'informations à des fins de lisibilité. La charte graphique se veut chaleureuse au possible, c'est pourquoi uniquement des couleurs chaudes dans des tons rouge/orangé (rappelant le feu, la vitesse, l'adrénaline qui caractérise le jeu tel qu'il est dans nos rêves) ont été selectionnées.
Au niveau du contenu, cette page fait office de portail pour le reste du site : elle n'est constituée que de gros boutons -s'inspirant du style des menus de jeux vidéos- pointant vers les catégories les plus importantes : les news, la présentation du projet et la page téléchargement.
Voici une capture d'écran de notre petit bijou :

FIG. 4.1 – www.hxt.fr - La page d'accueil

Le premier effet graphique à base de Javascript concerne la planche : celle-ci oscille lentement, comme si elle était véritablement soumis à la force de ses propulseurs. Une simple fonction cosinus et un timer suffisent pour y arriver.

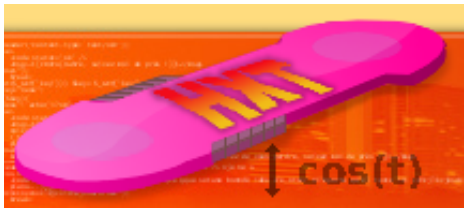


FIG. 4.2 – www.hxt.fr - La planche

Lors du premier clic sur un des boutons du menu ou de la page, le visiteur a alors la surprise de constater que tout s'enchaine sans entrainer un seul rafraichissement total de la page : cliquer sur un bouton amène le panneau principal à se fermer avec une animation, l'onglet de la nouvelle page se met alors en surbrillance dans le menu et le panneau s'ouvre de nouveau avec un contenu correspondant à la catégorie sélectionnée. Voici ce que cela peut donner :

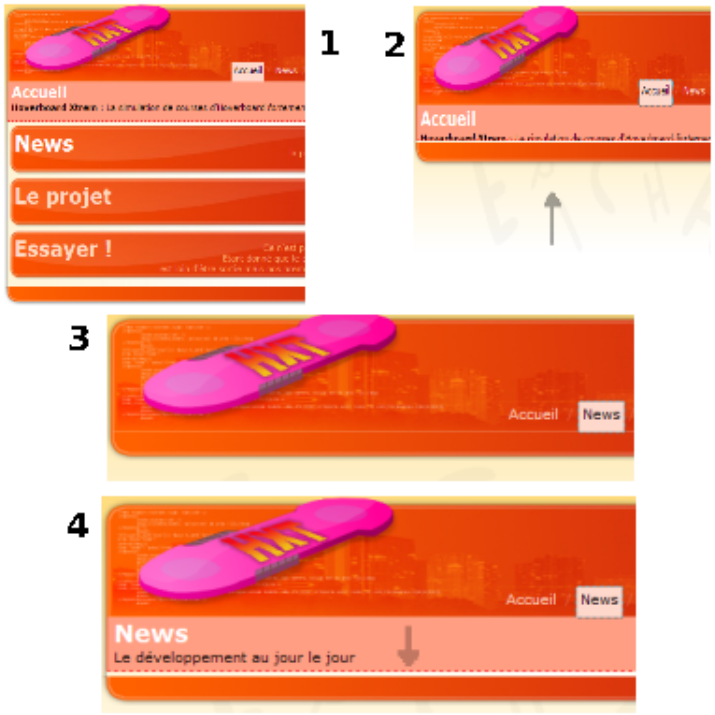


FIG. 4.3 – www.hxt.fr - Animation de transition

Nous arrivons sur la page des news qui est la seule page entièrement dynamique dans la version du site mis en ligne pour la soutenance. C'est ici que seront postées toutes les informations sur l'avancement du projet. Afin de pouvoir interagir avec l'utilisateur, un système de commentaires a été mis en place. La technologie Ajax prend ici tout son sens : alors qu'auparavant il fallait rechercher la page chaque fois que l'on voulait voir les derniers commentaire ou en envoyer un nouveau, notre version permet de réaliser tout ceci de manière la plus fluide possible : un panneau s'affiche par dessus le site, le masquant partiellement avec un effet de transparence et la liste des commentaire est récupérée automatiquement depuis le serveur. Si l'utilisateur souhaite en ajouter un nouveau, il suffit qu'il utilise formulaire prévu à cet effet :

FIG. 4.4 – www.hxt.fr - Boite d'ajout d'un commentaire

La page suivante est la page de présentation du projet, où se trouve toutes les informations utiles pour celui qui ne connaitrait pas du tout en consiste le jeu. Ensuite vient la page de présentation de l'équipe, qui explique pourquoi il est politiquement nécessaire à l'équilibre du monde de vouer un culte à l'EPiCHAT. La page Images regroupe quelques captures d'écran de nos travaux précédent, le tout classé par dossiers de la même manière que la page téléchargement. Enfin vient la catégorie liens où sont regroupées les sites proposant des renseignement utiles, description à l'appui.
Voilà, la promenade est terminée...
Mais il ne faut pas hésiter à revenir : le site sera mis à jour aussi souvent que possible et donc le contenu sera amené à s'étoffer au fil du temps.

4.1.4 Ce qu'il reste à faire

Même si l'on peut considérer le site comme terminé et fonctionnel (en accord avec les prévisions du Cahier des charges), il y a cependant quelque ammélioration possibles qui rendront son usage encore plus pratique.
Actuellement, il est possible (pour les membres de l'équipe) de poster une news aussi simplement que l'on poste un commentaire.
Il serait à l'avenir pratique de pouvoir l'éditer ou la supprimer.
Ensuite, la possibilité de mettre en forme les commentaires et les news avec un langage employé dans les forum comme le BBCode permettrait de rendre les pages moins uniformes.
Enfin, le plus gros du travail sera de remanier les pages projet, team, images, téléchargement et liens de manière à ce que l'on puisse ajouter ou modifier des informations sans avoir à modifier la base de données à la main.

Il reste de quoi faire mais le noyau est là!

CHAPITRE 5

Conclusion

Pendant ces deux mois qui ont séparé les deux soutenances, notre code a bien évolué, d'un côté les parties déjà avancées pendant la première soutenance, qui si de visu peuvent sembler proches des versions précédentes, sont aujourd'hui beaucoup plus évoluées (algorithmiquement parlant) et d'autre part grâce à l'arrivée de nouveaux élements comme le site, le son, ou encore un début d'interface.

Pour la prochaine soutenance, nous fournirons une version du moteur physique et du moteur graphique, non pas terminées, mais déjà vraiment bien avancés et utilisables.
D'autre part nous nous replongerons dans le réseau, et nous mettrons notre nez dans la gestion de l'intelligence artificielle dans le jeu. Et bien sur, vous pourrez suivre tout ça en quasi-direct sur http ://www.hxt.fr !