DA GNIII PROJECT

RAPPORT DE LA SECONDE SOUTENANCE

INFOSUP B1

BENIERE Alexandre
D'HALLUIN Florent
LEC Laurent

Table des matières

1 Introduction

Nous voici arrivés à la seconde soutenance ! A la première nous avions en main beaucoup d'outils développés mais qui ne servaient alors pas à grand chose. Depuis cette soutenance, nous avons donc travaillé avec pour poser les premières fondations du jeu final.

C'est donc ce travail que nous vous proposons de suivre au travers de ce beau rapport de soutenance qui complète ce qui n'aura pu être dit pendant l'oral en lui même, 15 minutes c'est court (mais 42 * 15 minutes c'est long je vous l'accorde !). Et puis rien que pour les beaux dessins, ça vaut le coup.

Bulletin spécial : en ce mardi 27 février, à 23h39, nous sommes toujours sans nouvelles du petit Alexandre qui a subitement disparu sur le chemin de sa maison le jeudi 16 dans la soirée. La police n'écarte pas l'hypothèse une fugue.

2 Rapport

2.1 Collisions

Le moteur de gestion des collisions a été en développement depuis la première soutenance en Janvier. Ce que nous attendons du moteur n'est pas une gestion de collisions entre modèles très précise, puisque pour rendre le jeu équitable, la zone de dommage des personnages dans les FPS est identique pour tous les joueurs. Par contre, il nous faut pouvoir traiter un grand nombre d'objets simultanément sur scène, afin de rendre le jeu plus immersif.

Aini, le module physique qui traite les collisions fonctionne de manière générale sur tous les objets de l'environnement, et ce, en trois temps :

Traitement des messages

On interprète d'abord les messages provenant du réseau, ce qui permet de savoir si un joueur se déplace, tire, ou quitte le jeu. La gestion du temps et du nombre de rafraichissements étant gérée par le module réseau, donc de manière centralisée par le serveur de jeu, le rafraichissement de l'environnement de jeu est déclenché par la réception du message correspondant.

Calcul des collisions

Le module parcours alors une liste d'objets contenus dans une instance d'environnement global :

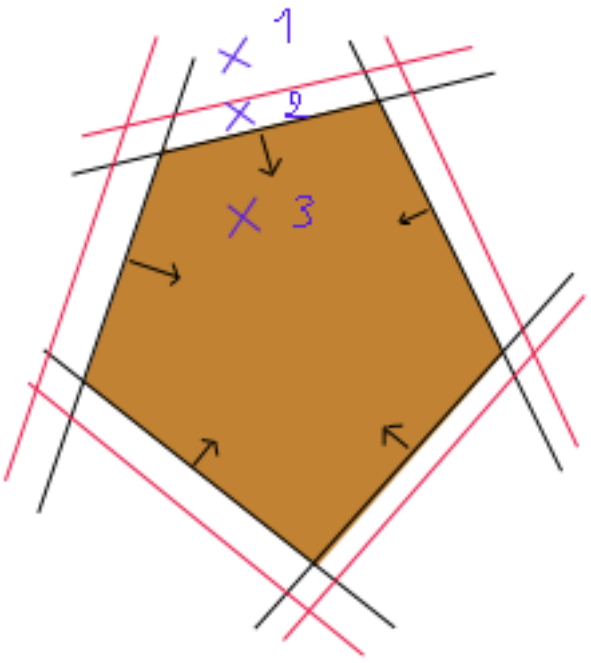
```
TEnvi = class
private

public
  IDLoc:       Cardinal;
  Gniii:       pTGniii;
  Octree:      pTOctree;
  Menu:        pTMenu;
  HUD:         pTHud;
  Polys:       pTDChaine;
  PhyNorm:     TDChaine;
  PhyAff:      TDChaine;
  PhyForces:   TDChaine;
  Camera:      TCamera;
  CIDTable:    TCIDTable;
  CIDPTable:   TCIDPTable;
```

Fig. 1 – (seule une partie de la classe est détaillée ici)

- Octree : Pointeur vers l'octree qui contient tous les objets instanciés, ainsi que les polygones simples composant la map, triés selon leur emplacement dans l'aire de jeu.
- PhyNorm : Liste chainée de tous les objets soumis aux collisions (par opposition aux objets aux trajectoires précalculées, comme les particules).
- CIDTable : matrice à deux dimension contenant des pointeurs de procédure, qui déterminent selon la classe de deux objets qui entrent en collision, leur comportement (rebond, destruction, perte de vie,...).
- CIDPTable : vecteur contenant des pointeurs de procédures, utiles cette fois lorsqu'un objet rencontre un polygone de la map.

C'est donc la liste PhyNorm qui est parcourue, et pour chacun des objets qu'elle contient on procède ainsi :

- Récupération de la liste des objets situés dans les registre proches de l'octree (seul le centre de l'objet étant répertorié dans l'octree, le nombre de registres parcourus dépend du rayon dudit objet, et du rayon de l'objet le plus grand présent).
- Détection de collision prochaine entre la sphère représentant l'encombrement de l'objet traité et chacunes des sphères représentant les objets proches. Il s'agit de résoudre l'équation déterminant le temps auquel deux se trouveront à une distance inférieure à la somme de leurs rayons. Si ce temps est inférieure à 1 (rafraichissement de position), la collision est imminente et est traitée immédiatement, en appelant la fonction de résolution située dans la CIDTable.
- Récupération de la liste des polygones proches (qui cette fois, sont insérés dans tous les registres qu'ils coupent)
- Détection de collision entre l'objet et chacun des polygones. Il faut ici non seulement déterminer à quel moment l'objet entrera en contact avec le plan de polygone, mais également si il y aura contact entre l'objet et le polygone, c'est à dire si le point de contact avec le plan se trouve à une distance inférieure à un rayon de l'intérieur du polygone :



A l'aide de produits scalaires et des normales aux droites des cotés situées dans le plan du polygone, on détermine pour chacun des cotés si

le point est suffisamment proche ou non de l'intérieur (entre les droites rouges). On a ainsi trois registres correspondants à :

1. Pas de collision.
2. Collision de coté.
3. Collision frontale.

La collision est ensuite traitée en fonction de l'ID de la classe de l'objet.

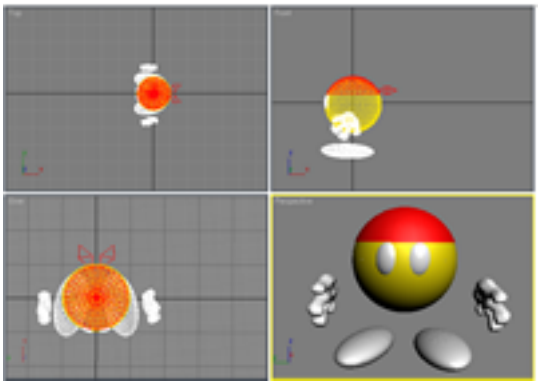
– Si un des objets doit etre détruit, il est ajouté à une liste de destruction, afin que sa suppression de perturbe pas la détection d'autres collisions. En dernier lieu, la liste est parcourue et les objets supprimés.

Mise à jour des positions

Les caractéristiques mécaniques des objets sont finalement mises à jour, c'est à dire position, vitesse, accélération, en fonction de la classe de ceux-cis.

2.2 Graphismes

Bien que le manque d'effectif se fasse ici cruellement sentir, nous avons commencé à réaliser quelques modèles sous 3D Studio Max. Voici par exemple la modélisation 3D du personnage principal, qui pourra être exporté en mesh Direct X.



Toujours en relation avec le graphisme, nous pouvons manipuler les lumières Direct X et les matériaux - le programme présenté en soutenance en montre un exemple. L'architecture du projet est conçue de telle manière que leur utilisation soit immédiate : Modèles est textures sont chargés au début de l'application dans des vecteurs, puis à chaque création d'un objet ou polygone, son modèle et/ou texture lui est directement lié, ce qui garanti un gain de temps maximal, et dans l'instanciation, et dans le rendu de chaque scène.

En plus des objets 3D, les sprites DirectX sont utilisés, notamment pour l'affichage des menus ou du HUD

2.3 Threads

Comme il a été présenté à la soutenance précédente, l'architecture du project se divise en plusieurs modules indépendants. Grace à l'intermédiaire des threads, nous pouvons simuler un calcul en parallèle de chacun des modules :

Comment fonctionne un thread

Un thread est un objet possédant une méthode principale Execute. Le système d'exploitation alloue tour à tour la puissance du processeur à chacun des threads lancés pendant un temps défini, ce qui leur permet d'effectuer un certain nombre d'opérations atomiques (telles que lire un emplacement mémoire, écrire sur un emplacement, effectuer un calcul ou un test). Cependant, lorsque cette période est écoulée, l'exécution du thread est stoppée, et d'autre threads peuvent utiliser la même plage mémoire. D'où un risque d'erreur élevé, si l'on ne prend aucune précaution lors de la programmation des ces objets.

Voici une illustration :

Deux threads A et B possèdent chacun un pointeur _p vers le même emplacement mémoire, ou est stocké un entier.

```
procedure ThreadA.Execute;
begin
  _p^ := _p^ + 1;
end;

procedure ThreadB.Execute;
begin
  _p^ := _p^ * 2;
end;
```

On ne voit apparemment ici qu'une seule opération, qui consiste pour le thread A, à ajouter 1 au contenu de l'emplacement _p, et pour le thread B, à multiplier cette valeur par 2. Pourtant, il y a en fait trois opérations dans chaque procédure :

1. la lecture de la valeur
2. le calcul du résultat
3. l'écriture de la nouvelle valeur

Ce qui peut engendrer le cas suivant :

```
 - Execution du thread A -
Thread A : Lecture du contenu de _p : 10;
 - Execution passe au thread B -
Thread B : Lecture du contenu de _p : 10;
Thread B : Calcul de la nouvelle valeur : 20;
Thread B : Ecriture de la nouvelle valeur : _p^ := 20;
 - Execution passe au thread A -
Thread A : Calcul de la nouvelle valeur : 11;
Thread A : Ecriture de la nouvelle valeur : _p^:= 11;

Résultat attendu : _p^ = 21;
Résultat optenu :  _p^ = 11;
```

Ansi, il a fallu redoubler d'attention au niveau du partage de la mémoire, notamment lors de l'échange de messages : un message doit être rédigé totalement avant d'être ajouté à la pile du thread récepteur (opération atomique : assignation d'un pointeur)

Comment les utilise-t-on en pratique dans le projet

Stade actuel

En plus du processus principal, qui gère la création, la destruction de l'application et une boucle principal de traitement de messages windows, nous nous servont des threads suivants :

```
procedure Main;
var
  PhysiqueThread:  TPhysique;
  ReseauSPThread:  TReseauSP;
  AffichageThread: TAffichage;
  EffecteurThread: TEffecteur;
```

- PhysiqueThread : Moteur physique dont le fonctionnement a été détaillé précédemment.
- ReseauSPThread : Réseau simulé qui envoie des messages de rafraichissement et ordres d'action au thread physique, mais n'utilise pas de connection avec un ordinateur distant : c'est une serveur local pour un seul joueur.

- AffichageThread : Boucle d'affichage qui parcours l'environnement et affiche les objets qu'il contient.
- EffecteurThread : Boucle de contrôles, qui gère les ordres de déplacement, de tir, et l'appel au menu.

Limite théorique

Bien que le temps manquera pour le mettre en application, on peut imaginer un modèle comme celui-ci, reposant sur un ordre de priorité :

- Application principale
- Thread physique principal
- Thread reseau client
- Thread reseau serveur
- Thread effecteur
- Thread d'affichage
- Thread physique secondaire (lié à l'affichage)
- Thread de traitement du son
- Thread(s) d'intelligence artificielle

2.4 Contrôles

Introduction

Le système de messages Windows offrait un environnement facile à utiliser mais présentait de gros inconvénients : les temps de latence. Nous nous sommes donc tournés vers une alternative performante et finalement aussi simple, à savoir DirectInput.

Principe

L'effecteur est un thread rapide, prévu pour etre exécuté jusque 100 fois par seconde. Bien que cette limite soit un peu exagérée, il faut noter que le déplacement de la caméra dans un FPS est primordiale pour avoir une vision fluide de l'environnement. Ainsi nous avons fait le choix de dissocier le rafraichissement de caméra du rafraichissement par le moteur physique du joueur lui même. L'effecteur a donc la charge de mettre à jour la direction de la caméra, et de prendre note des actions du joueur sur le clavier et la souris.

On suit donc tout d'abord la progression de la souris en coordonnées relatives. Et c'est bien là l'avantage de Direct Input. Si l'on obtenait les coordonnées en absolu, il y aurait quelques soucis quand le curseur arriverait au bord de l'écran (même si on est en dual screen...) Puisque Direct Input gère le déplacement en relatif, nous n'avons pas ce problème car nous connaissons exactement la quantité de mouvement de la souris suivant les axes X et Y. Nous y avons bien entendu ajouter la gestion des boutons gauche, droit, de la molette ainsi que de boutons auxiliaires. Concernant le clavier, l'état de chaque touche (enoncée ou non) est récupéré dans un tableau d'une centaine de shorint correspondant à chacune des touches.

```
// Quit
if (Controls^.Quit.Keys[0]^ = 128) or
   (Controls^.Quit.Keys[1]^ = 128) then
begin
  Gniii^.continue := False;
end;
```

Bien que le parcours d'un tableau de cette taille ne soit que peu gourmand en ressources, le système de pointeurs permet un controle astucieux des commandes, et une personalisation des controles tout ce qu'il y a de plus aisée : Le comportement de chacune des action est défini dans le programme, et à chaque action est associée une adresse mémoire, celle contenant l'état de la touche qui lui est assignée dans les paramètres de configuration, adresse qui se trouve appartenir au tableau dans lequel est mis à jour l'état du clavier par DirectInput. On peut donc facilement modifier les contrôles, même en temps réel.

2.5 Chargeur de maps

Introduction

Le parseur de maps présenté à la première soutenance a été prévu pour charger des maps, ce que nous avons commencé à faire pour cette soutenance. Il nous fallait un langage et un parseur flexibles permettant de gérer facilement les maps ainsi que leur évolution au fil du projet. Par conséquent, nous ne montrons qu'une petite partie de ce que le parseur devrait nous permettre de faire pour la suite du projet.

Principe

A la dernière soutenance, nous avions un parseur de maps qui parsait un langage, le *DGL*. Ce langage consistait à donner un constructeur que nous faisions suivre de ses paramètres, et il y avait autant de lignes de paramètres que d'appel du constructeur.

Cela n'a pas changé. Nous disposons désormais de cinq constructeurs :

```
index
texture
mesh
object
triangle
```

Le premier permet de connaître à l'avance le nombre de textures et de meshes chargés en mémoire. Pourquoi connaître ce nombre ? Tout simplement parce que l'affichage fonctionne avec des tableaux de pointeurs vers ces mêmes objets. Puisque les tableaux dynamiques de Delphi ne sont pas performants et

puisqu'une liste chaînée aurait pris trop de ressource, le tableau est statique. Il faut donc connaître à l'initialisation du tableau le nombre d'éléments qu'il va contenir.

Le constructeur *texture* lui, va appeller une procédure permettant de charger cette texture en mémoire. On lui assigne un ID afin de l'identifier et on spécifie bien évidemment le fichier source, et la procédure du loader de maps va charger cette texture en mémoire et un pointeur vers cette texture va alors être ajouté dans le tableau de textures.

A l'instar de *texture*, *mesh* va permettre de charger des modèles meshes en mémoire. De même que pour les textures, on lui assigne un ID et on indique le fichier contenant le modèle, et une procédure va alors se charger de l'ajouter en mémoire.

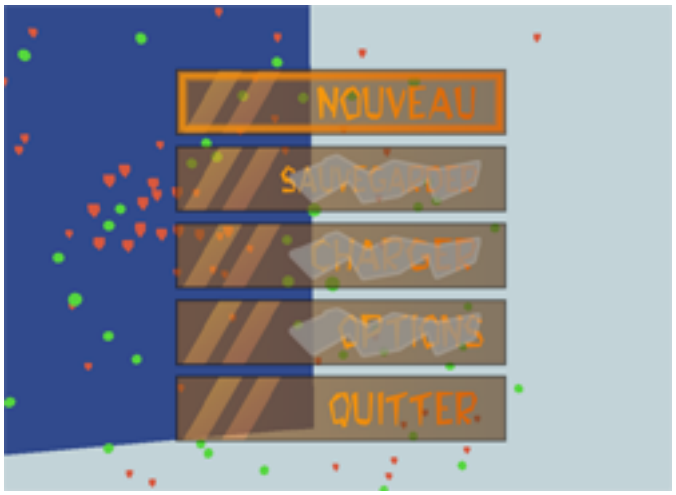
Les deux derniers constructeurs ne font que charger en mémoire des ressources. Des textures et des meshes seuls ne sont pas utilisables, pour cela on va ajouter deux autres constructeurs.

Le premier, *object* va ajouter dans la zone de jeu, et par conséquent dans les différentes structures algorithmiques le manipulant (tel que l'octree) un nouvel objet. On indique donc ses coordonnées dans le fichier map et une procédure le charge. Cet objet n'est pas réellement concret, pour cela on le représente par un modèle mesh. Le constructeur connaît donc l'ID du mesh qui représentera cet objet. Afin de gérer les collisions, ses dimensions sont elles aussi indiquées dans la map.

Enfin, *triangle*, le dernier constructeur, va permettre d'afficher des triangles, texturés ou non. Pour cela on spécifie les coordonnées des trois sommets ainsi que la texture que portera cette forme et on l'ajoute à nouveau là où il est nécessaire de l'ajouter pour gérer l'affichage et les collisions. Ce constructeur devrait évoluer pour permettre de dessiner des formes plus complexes, si bien sûr elles deviennent indispensables.

Conclusion

Le système marche donc parfaitement. Il ne demande qu'à être pleinement exploité par le moteur physique qui gérera alors entièrement les textures, les lumières, les matériaux... et peut-être, qui sait, les événements scriptés de tout bon jeu !



2.6 Menus

Principe

Bien que son potentiel ne soit pas encore totalement exploité, le système de menu se décompose sous la forme d'un arbre, dont chaque noeud possède une action associée (pointeur de procédure) et une liste de sous-menus, ainsi qu'une liste de paramètres qui lui sont propre. Une classe encapsule les sous arbres pour permettre une navigation simplifiée : On garde l'adresse du menu ouvert et l'on affiche à chaque rafraichissement de l'écran ses sous-menus, grâce à l'utilisation de textures transparentes (alpha blending) et d'un sprite général sur lequel on dessine.

Spécificités

```
TMenuAction = procedure of object;

TMenuItem = record
  Caption:     string;
  TextureFile: string;
  Texture:     IDirect3DTexture9;
  Ratio:       single;
  Available:   Boolean;
  Highlighted: Boolean;
  Location:    pTDMaillon;

  SubMenus:    TDChaine;

  Action:      TMenuAction;
end;
```

La structure d'arbre du système de menu permet une initialisation facilitée et adaptable par exemple à un système de script, qui permettrait de modifier en temps réel le menu en fonction par exemple de l'état du joueur à l'écran ou des actions à sa disposition.

Une partie cependant de cet arbre est précalculée afin d'optimiser toujours plus la vitesse de l'application. C'est le cas notamment du chargement des textures de chaque item, qui se fait donc grâce à un algorithme de parcours d'arbre, récursif cette fois par rapport à l'octree utilisé dans l'environnement, étant donné qu'il n'est destiné à être exécuté qu'une seule fois, et que le risque d'erreur de programmation est bien moindre.

2.7 Site web

Introduction

Comme annoncé dans le cahier des charges, le site web est un vecteur - non non pas ceux qu'on trie dans tous les sens - important d'informations. Nous souhaitions donc un site clair, concis et ergonomique. On y retrouve ici toute l'ambiance du projet avec nos désormais connus personnages - faudra s'efforcer de leur trouver un nom.

Concernant l'aspect technique, le site utilise bien évidemment le XHTML 1.1 et le CSS 2.0, tout en respectant les normes éditées par le W3C. Nous nous sommes servis aussi du PHP pour faciliter la génération des pages web. Les news sont gérées en MySQL, le reste du site étant statique, il ne demande pas d'utiliser la puissance des bases de données - même si l'on s'en servira probablement pour les fichiers et les liens.

Avant de rentrer plus en détails sur chaque partie du site, il est continuellement consultable à cette adresse : http ://webdgp.free.fr/ - bonne visite !

News

La section "News" permet de rapidement se tenir au courant des dernières news concernant le site et le projet, elles sont classées par date de manière à toujours maintenir l'actualité en haut de la page.

DGP

Cette seconde partie permet de mieux situer le projet pour l'utilisateur. Le type de projet et le ton sont donnés via un long texte, qui était d'ailleurs le texte d'introduction du cahier des charges. Les captures d'écran viendront s'y ajouter dès qu'elles seront suffisament significatives pour le visiteur.

Fichiers

Afin de permettre aux visiteurs, et futurs INFOSUP de contempler (ou pas) notre travail, la section “Fichiers” accueille nos différents rapports au format PDF, auxquels viendront s’ajouter les exécutables et sans doute quelques bonus si le temps nous le permet.

Team

La partie “Team” sert à mettre en avant les différentes personnes (bon ok on est que deux) qui s’investissent dans le projet. Des informations viendront

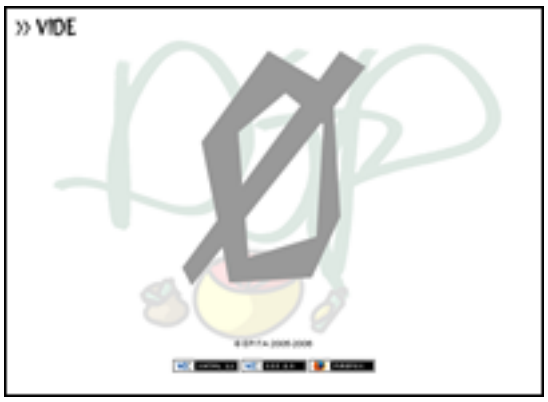
peut-être s'ajouter, si l'inspiration nous vient (et nous connaissant, ça va pas être triste).

Liens

La section "Liens" présente différents liens, classés selons différentes catégories et classés par ordre alphabétique de contenu. On y trouve des liens consacrés à la programmation Pascal Object dans une première section intitulée "Delphi". Ensuite, nous avons mis quelques liens en relation avec Direct X. La section "Divers" comprend les liens ne pouvant être classés ailleurs. Et enfin, la section "Autres projets" contient des liens vers d'autres projets de la promotion, cela va de soit. Les liens seront ajoutés au fur et à mesure de leur découverte.

Vide

La section vide, quant à elle... est vide.



Conclusion

Le site est donc terminé. Nous effectuerons sans doute quelques retouches, et bien sûr il sera continuellement mis à jour avec des captures d'écran, des news et les prochains rapports de soutenance. Il sera en outre toujours disponible l'année prochaine pour les prochains SUPs à la recherche d'inspiration !

3 Nouvelle répartition du travail

Légende : ø Pas abordé − Commencé + Bien avancé ⊗ Terminé

3.1 Seconde soutenance

	Alexandre	**Florent**	**Laurent**
Moteur graphique	?		+
Moteur physique	?	+	
Son et graphismes	?	−	−
Réseau	?	−	−
Gameplay	?	−	
Site web	?		+

Le réseau n'a pu être approfondi comme il était annoncé à la soutenance précédente. Nous avons préféré à place nous concentrer sur les menus. Il ne s'agit donc pas d'un retard mais d'un déplacement.

3.2 Troisième soutenance

	Alexandre	**Florent**	**Laurent**
Moteur graphique	?		+
Moteur physique	?	+	
Son et graphismes	?	+	+
Réseau	?	+	+
Gameplay	?	+	+
Site web	?		⊗

3.3 Dernière soutenance

En théorie, tout devrait être terminé.

4 Conclusion

Le travail est donc bien avancé et les principaux mécanismes sont en place. Notre objectif pour la troisième soutenance est de relier tous ces composants via un système élaboré de messages qui permettra une meilleure communication entre les différents modules. Il nous faudra aussi préparer les différents éléments graphiques et sonores qui pourront être implémentés dans le jeu. Il reste donc encore pas mal de travail à fournir, mais nous devrions pouvoir aborder la troisième et avant-dernière soutenance avec un projet déjà bien abouti.