DA GNIII PROJECT

RAPPORT DE LA TROISIEME SOUTENANCE

D'HALLUIN Florent
DURAND Renaud
LEC Laurent

Table des matières

1 Introduction

Nous voici donc arrivé à la troisième soutenance. Malgré les pertes subies - émeutes des banlieues, grippe aviaire, tout ça... - notre équipe est toujours debout, ou plutôt encore assise à coder. L'arrivée d'un troisième membre[1], a permis de mieux partager les tâches non réalisées par Nicolas Aycardi et Alexandre Bénière au sein du groupe.

Cette troisième soutenance n'est pas une épreuve mineure avant la soutenance finale, qui viendra d'ici environ deux mois. Elle détermine avant tout notre capacité à produire et rassembler divers éléments - affichage, physique, son, contrôles... - qui pourront être utilisés au cours des semaines à venir pour finaliser le projet.

Le résultat obtenu, dans le fond, est donc sensiblement ce qui composera le corps même du jeu que nous souhaitions développer. Nous allons introduire au long de ce rapport les différents éléments nouveaux qui permettront d'ici juin de finaliser ce merveilleux produit qu'est *Da Gniii Project*.

[1] Le malheureux retrait du CPE nous empêche d'embaucher plus, ni même d'exploiter Renaud.

2 Rapport

2.1 Le Son

Introduction

Nous voulions utiliser DirectSound au départ, mais aux vues de l'absence quasi totale d'exemples de sources en Delphi, notre choix s'est réorienté vers la bibliothèque FMod qui est très simple d'utilisation et qui contient tout une flopée de fonctions très pratiques.

Avec FMod il est assez Simple de charger un son, par exemple si l'on veut charger le son "*mon_son.wav*" on procéde ainsi :

```
FSOUND_SAMPLE_LOAD(channel,'mon_son.wav',Mode de Lecture,0,0) ;
```

Pour un mp3 c'est un peu différent car il s'agit d'un stream et le charger en mémoire demanderait trop de place. On utilise alors :

```
FSOUND_STREAM_OPEN('mon_mp3.mp3',Mode De Lecture,0,0) ;
```

Une fois chargés c'est tout aussi simple de les jouer :

```
FSOUND_PLAYSOUND(Channel, Mon Sample) ;
FSOUND_STREAM_PLAY(Channel, Mon Stream) ;
```

Pour détruire les son :

```
FSOUND_SAMPLE_FREE(Mon Sample) ;
FSOUND_STREAM_CLOSE(Mon Stream) ;
```

Il est bien sûr possible de jouer simultanément le même son sur plusieurs channels différents.

Principe

Tous les samples dont nous avons besoin sont au lancement du jeu ou de la partie chargés dans une liste. Si l'on veut jouer un son il suffit d'accéder à l'emplacement du son dans la liste et de le jouer.

Comme toutes les parties du projet, les évènements sonores sont contrôlées par un Thread. Le Thread contient une boucle qui actualise les informations dans la liste de sons en train d'être joués en fonction des informations transmises par le moteur physique ainsi que la position du joueur.

Quand on joue un son, un channel lui est attribué. Pour pouvoir gérer le son - le mettre en pause, actualiser sa position dans l'espace - il est nécessaire de créer une nouvelle liste dans laquelle figurent les sons en train d'être joués. Dans le cas des stream, ceci ne libérant pas le channel une fois terminés ; il est nécessaire de tester si sa lecture est terminée et dans ce cas, on libère le channel. Quand nous n'avons plus besoin des sons, c'est à dire à la destruction du thread, ou quand un MP3 est fini, une procédure les détruit.

Son 3D

FMod permet de gérer facilement un environnement sonore en 3D. La physique fournit en permanence au thread la position du joueur et, pour chaque son, leur position dans l'espace. Ainsi le volume et les balances sont directement contrôlées par FMod lui même.

Ainsi on peut positionner notre joueur dans l'espace grâce à la procédure

```
FSOUND_3D_LISTENER_SETATTRIBUTES(position, vitesse,
      orientation) ;
```

Pour le son il s'agit de :

```
FSOUND_3D_SETATTRIBUTES(Channel du son, position,
      vitesse) ;
```

On peut aussi déterminé jusqu'à quelle distance le son est à pleine puissance et jusqu'à quelle distance il sera audible grâce à cette dernière fonction :

```
FSOUND_3D_SETMINMAXDISTANCES(Channel, DistanceMin, DistanceMax) ;
```

Conclusion

Le résultat est assez convaincant avec FMod, puisqu'en un minimum de fonctions il nous a permis de réaliser entièrement, ou presque, ce que nous souhaitions obtenir.

2.2 Affichage - le HUD

Introduction

Le “HUD” est l’interface entre le joueur et la partie. Il permet de connaître en temps réel de connaître sa vie, son énergie, le stock de munitions, le nombre de joueurs encore en vie... bref si l’on est dans la merde ou pas !

Les sprites

Pour afficher les différents items on aurait pu constamment les afficher devant la caméra, mais il y a bien plus intelligent : repasser en 2D.

Direct X permet un affichage très simple de la 3D, et ce, à l’aide des “sprites”. Il était un temps pas si lointain ou Mario n’était pas un ensemble de triangles, mais juste une image de quelques pixels appliquée à une surface, à savoir la télé.

Le principe reste le même, mêeme si depuis depuis la technique a évolué. Nous appliquons donc nos textures sur une surface 2D, notre écran. Le système est donc adapté à ce que nous comptions faire, à savoir afficher un HUD digne de ce nom.

Les données physiques comme le niveau d’énergie ou le nombre de munitions sont stockées dans une variable connue par l’objet HUD, en effet un pointeur vers cette variable est une propriété de cet objet. Il reste ensuite à afficher les différents éléments. On peut de plus afficher du texte, en utilisant une texture par caractère.

Les inconvénients

Bien qu'*a priori* ce système ne demande rien de particulier puisque très facile à mettre en place, il nécessite néanmoins une importante place dans la mémoire vive. En effet, chaque texture est mise en mémoire afin d'être utilisée.

De plus, la qualité des textures ne peut pas être maximale, comme en bitmap par exemple ; cela prendrait beaucoup trop de place. De fait, on utilise un format assez pratique puisque gérant la transparence : le PNG. La qualité est très bonne et la transparence peut alors être pleinement exploitée.

2.3 Gameplay

Introduction

Entre la seconde soutenance et la suivante (troisième, donc...), beaucoup de travail a été effectué niveau gameplay, c'est à dire qu'une fois les procédés techniques mis en place, le but du jeu est alors de le faire (le jeu, donc). 'Fin bref, tâche pas si facile nécessitant bien souvent des modifications conséquentes de la structure interne du projet. Si peu facile, d'ailleurs, qu'on a voulu faire un système adaptable, quelque chose qui permettrait de ne pas repasser une douzaine d'heure à chasser les erreurs de typo. Et donc, On scripte !

Notions de Mod

Bien bien bien, on va faire simple : Nous avons déjà présenté lors de la seconde soutenance un système de chargement de map, qui permettait de charger polygones, textures, et d'instancier des objets dans l'environnement. Aujourd'hui, on fait encore plus fort. Un fichier *.gmd [2] contient une mine d'informations permettant faire tourner le jeu de manière totalement personnalisable :

Structure d'un fichier de mod

- **Index :** On y décrit ce qui suit, nombre de modèles, textures, sons à charger, nombre de classes déclarées, d'armes, ainsi que d'autres paramètres spécifiques
- **Texture :** On liste simplement les textures utilisées en leur associant un index
- **Mesh :** De même pour les modèles
- **Sound :** ... et pour les samples audio
- **Class :** architecture de base d'un type d'objet, reprise dans les déclarations d'arme et de classe de joueur
- **Weapon :** les différentes 'armes' (moui, bon, les machins qui lancent d'autres trucs, bref pas d'quoi torturer un canard, ca reste une simple démo technique)
- **Player :** LA classe joueur. Oui, parce qu'on peut avoir plusieurs classes de joueur, avec des comportements qui peuvent être complètement différents (celui qui tient le fusil, et celui.. qui code *tsk*)

[2] *Gros Mangeur de Dattes*

```
class {
  // Allows multiple declarations

  // i - class ID (starts at 0)
  2,
  // s - class name (just so there is one)
  "MusicPlay",
  // f - mass (unused atm)
  0,
  // b - affected by forces
  0,
  // f - cchoc : energy reflected (0 - 1+)
  1,
  // i - Model ID (has to be declared previously)
  0,
  // i - texture ID (may not be used later on)
  4,
  // i - maxlife
  100,
  // f - radius
  95,

  // i - number of init events;
  0,

  // i - number of collision with objects events
  2,
  0001MusicPlay("Croaker-tempnis.mp3"),
  0001PhyKillProj(other),

  // i - number of collision with polygons (map) events
  0,

  // i - number of ending events;
  0,

  // i - Ending parameter, useless...
  0;

}
```

Oh, me diriez-vous, c'est bien beau, mais j'ai rien compris... c'estquoidonc ça ?

```
0001MusicPlay("Croaker-tempnis.mp3"),
```

ou ça ?

```
0001PhyKillProj(other),
```

Darn ! Mokay, la suite.

Scripting

Chez Hexen (DGPTeam) nous sommes de grand indécis (si, si...) et donc, afin de garantir la flexibilité du projet, on a mis en place un système de scripting (comme les pros !), permettant de contrôler l'environnement de manière intuitive sans trop s'empoigner l'encéphale, en gros, un truc bien utile pour la suite, qui se révèlera fort pratique dans le futur.

TEffect

Le script repose principalement sur l'utilisation d'effets assignés à un évènement d'un joueur, d'une classe d'objet, d'une instance particulière de cette classe ou d'une quelqu'autre structure. Un effet se présente ainsi :

```
TGenEffect     = procedure (_arg: pointer);

TEffect = record
  arg: pointer;
  siz: word;
  fnc: TGenEffect;
end;
```

Ce qui signifie en gros que c'est une structure qui sait lancer une procédure et lui indiquer où se trouvent en mémoire ses arguments d'appels, sans n'avoir aucune idée de ce que cette procédure effectue.
On peut ainsi regrouper ces structures dans des listes d'effets et les effectuer à la suite de manière automatique. Il suffit de savoir créer l'argument correspondant à un effet donné. Ainsi donc, un effet particulier de présente sous la forme d'une fonction de création, d'une procédure d'exécution et d'un type d'argument.
Par exemple :

```
Type
  pTObjValClcData = ^TObjValClcData;

  TObjValCLCData = record
    obj: pointer; // pointeur vers pTOPhy
    val: integer;
    clc: pTDChaine;
  end;

// endommage l'objet concerné
procedure ObjDamage(_arg: pointer);
begin
  with pTObjValClcData(_arg)^ do
  begin
    pTOPhy(obj^)^.life := pTOPhy(obj^)^.life - val;
    if pTOPhy(obj^)^.life <= 0 then
    begin
      clc^.FAddItem(pTOPhy(obj^));
      pTOPhy(obj^)^.idq := True;
    end;
  end;
end;


function  ObjDamageCreator(Gniii: pTGniii;
                _cd: pTCollData; arg:string): pTEffect;
var
  o: pTObjValClcData;
begin
  new(o);
  if LowerCase(getString(arg)) = 'other' then
    o^.obj := @(_cd^._o2)
  else
    o^.obj := @(_cd^._o1);
  o^.val := getInteger(arg);
  o^.clc := _cd^.clc;
  new(Result);
  Result^.arg := o;
  Result^.siz := SizeOf(TObjValClcData);
  Result^.fnc := ObjDamage;
end;
```

Qui permet, par exemple lors d'une collision, d'endommager un des objets concerné.

Le parseur général utilisé par l'application se charge de récupérer le nom de l'effet et ses paramètres à partir du fichier de mod, puis un crible alphabétique détermine la fonction de création d'effet à appeler.

Conclusion

Le système de scripting permettra, lorsque les dernières pierres de l'architecture du projet seront en place, de créer un jeu digne de ce nom, et ce, sans avoir à se replonger dans des suites de lignes de code rébarbatives (parfois même frustrantes)

2.4 Site web

Le site web était déjà quasiment fini à la seconde soutenance. En effet, rien n'a été changé mis à part quelques détails mineurs ne nécessitant par une description ici.

Néanmoins il continue d'être actualisé dès qu'un événement majeur ou une mise à jour est apportée au site. Les illustrations ont été ajoutées dans la parties "fichiers" et sont donc désormais téléchargeables.

3 Nouvelle répartition du travail

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

3.1 Troisième soutenance

	Renaud	**Florent**	**Laurent**
Moteur graphique			+
Moteur physique		+	
Son et graphismes	+		+
Gameplay		+	
Reseau	-		
Site web			⊗

3.2 Dernière soutenance

En théorie, tout devrait être terminé.

4 Conclusion

Un rapport de soutenance léger et sans détours compliqués, qui on l'espère, pourra sans problème être digéré. Beaucoup de choses ont été améliorée, refaite de sorte qu'il y a peu de choses nouvelles et réellement perceptibles. Elles sont pourtant là et leur importance n'est pas à négliger.

Da Gniii Project avance sérieusement sans pour autant entraîner une omnubilation à tendances suicidaires. Plus l'on progresse, plus nos pas se font grands. Le jeu fini sera sans nul doute bien plus limité que ne l'étaient nos ambitions de départ, mais notre créativité y gagne une part de réalisme, qui permet peut-être de mieux appréhender ce qu'il faut en tripes et en foie de veau pour mener un projet à son terme.

Nous avons donc quasiment terminé ce qui servira de moteur au jeu lui même, il ne nous reste plus qu'à apporter quelques modifications à nos sources et le tout sera exploitable pour commencer à réaliser notre jeu.