DA GNIII PROJECT

RAPPORT DE LA PREMIERE SOUTENANCE

INFOSUP B1

BENIERE Alexandre
D'HALLUIN Florent
LEC Laurent

Table des matières

1 Introduction

Welcome to the report on the current state of DGP's development. I have been personally approved by Hexen Team to introduce you to their brand new concepts in game designing. Please get a drink and sit comfortably. Now relax, and enjoy the ride.

As you probably know, DGP is a new kind of FPS videogame. Basically, it has all the characteristics of this type of game, but not fully developed and in a much weirder way. Anyway we still have to work on it because it seems to be important to some important people's important eyes. Somehow, everyone doesn't agree on that, but as you read these lines, you probably have already heard about some acceptable casualties.

Because the whole washing-up is always much more than half the washing up, we have to be less greedy. Still there is a lot of fun to come. After all we are supposed to be addicted.

I'm sorry for those who couldn't understand a single word, but there is no way I can get rid of that nasty EAccessDenied exception in the French language module.

2 Rapport

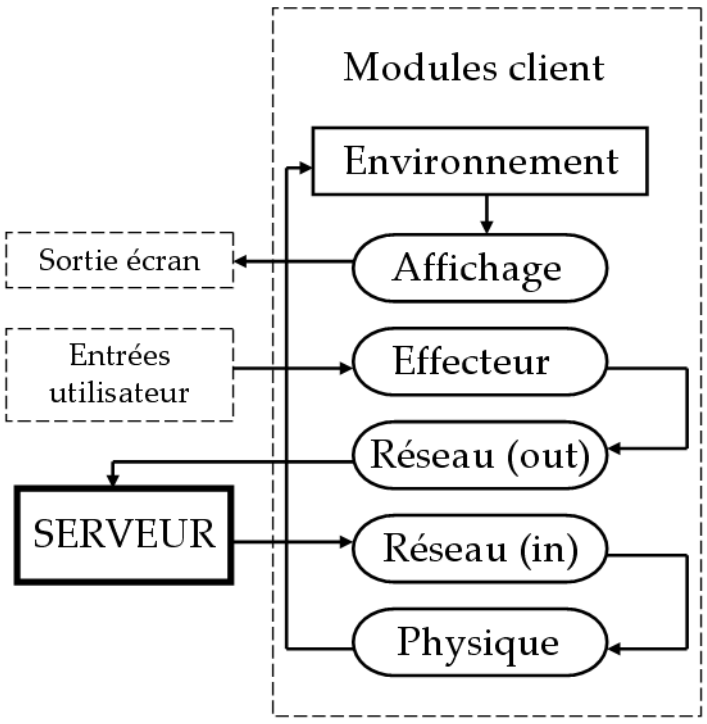
2.1 Architecture du projet

Introduction

Pour faciliter le jeu en réseau, l'architecture du projet a été pensée de manière souple : Deux applications de base, client et serveur, mais qui peuvent être instanciées sur la même machine. Chaque application est composée de modules qui tournent en parallèle, pour assurer une fluidité globale. Ainsi, si l'affichage souffre de ralentissements, le module responsable de la mise a jour des coordonnées physiques n'est que peu affecté. Ceci est possible grâce à l'utilisation de *threads*, sous-routines d'un programme exécutées de manière quasi indépendante du programme principal. De manière générale, chacun des modules possède son propre thread. Ces modules intéragissent grâce à l'utilisation de files de messages qui permettent de maintenir la fluidité recherchée.

Client

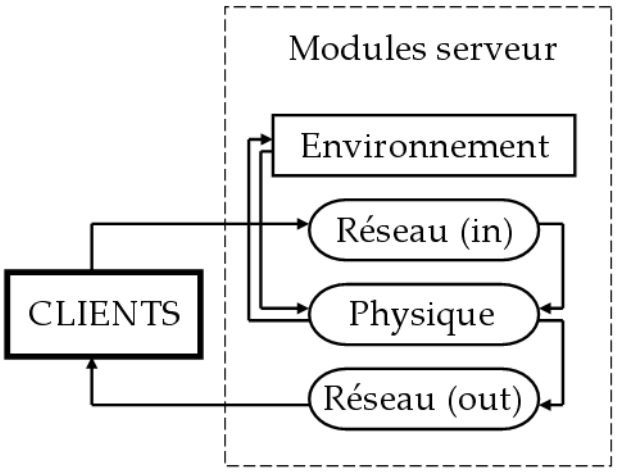
L'application cliente se compose comme suit :



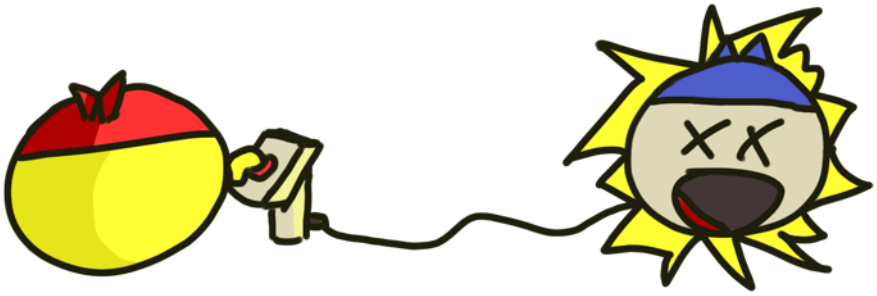
- *Affichage* : Responsable du parcours de l'environnement et de l'affichage des objets sur l'écran.
- *Effecteur* : Se charge de récupérer les entrées clavier - souris et de transmettre les ordres au serveur par l'intermédiaire du module réseau. C'est la boucle principale du programme, chargée également d'initialiser et de détruire les autres modules.
- *Réseau* : Responsable des échanges avec le serveur.
- *Physique* : Permet d'exécuter les ordres du serveur sur l'environnement local.

Serveur

L'application serveur sert principalement à centraliser et à coordonner les actions de tous les joueurs. Elle contient l'environnement physique de référence, et empèche les désynchronisations. Elle est batie sur ce modèle :



- *Réseau* : Capable de traiter les messages provenant de tous les joueurs. Des ordes de priorité doivent être mis en place pour garantir une bonne jouabilité.
- *Physique* : Agit sur l'environnement de référence et exécute les ordres de tous les joueurs pour les valider et les transmettre tous.



2.2 Le réseau

Puisque le jeu développé est avant tout un jeu convivial, une gestion du réseau s'imposait.

Tout d'abord il a fallu choisir entre les différentes librairies offertes par Delphi : *Indy* ? *ICS* ? *Winsocket* ? Après avoir effectué différent test d'emission et de réception, c'est *Winsocket* qui s'est avéré le meilleur choix puisque son utilisation permet un gain de quelques dizaines de millisecondes.

C'est donc avec l'utilisation des *Socket Windows* que sera développée l'architecture réseau.

Principe

Le réseau permettra de faire dialoguer les différents joueurs. Ces messages sont donc stockés dans une file et peuvent être consultés le moment venu.

Pour l'instant il ne s'agit que d'une ébauche du système final, mais il permet de se rendre assez bien compte du résultat intégral.

Les messages seront des types record d'entiers. Ils permettront de stocker une id de la personne qui envoie (id du joueur qui se déplace par exemple) mais aussi des messages plus complexes qui seront interprétables par une décomposition du nombre en tranches d'un certain nombre de chiffres.

Une gestion des priorités sera elle aussi mise en place afin de traiter les messages plus importants tels que la physique, avant d'autres qui le sont moins, tels que la lecture des sons.

Malheureusement même si les bases y sont, nous ne sommes pas en mesure d'en présenter plus pour l'instant.

2.3 Le parseur de fichiers

Introduction

Le parseur de fichier est directement lié au chargement de cartes pour le jeu - voire peut-être à leur génération si le temps nous le permet. En effet, les cartes, ou maps, sont stockées sous la forme de fichiers textes **.map*.

Pour cela, nous avons choisi de développer un langage scripté, à la syntaxe C-ienne ({ , ; " /* */ // etc...) que l'on appellera le DGL, *Da Gniii Language* - ou *Le Langage qui Grince*.

Présentation du DGL

Le DGL est un langage de script. Son interprétation permettra le chargement des objets et leur placement qui seront utilisés par le moteur graphique pour l'affichage de la carte, et par le moteur physique pour les relations entre les objets et l'environnement.

Sa syntaxe est simple. On utilise des *tokens* faisant office de constructeurs. Ce sont ces derniers qui indiquent au parseur quelle opération va être effectuée. On peut les assimiler à des identifiants de fonctions.

Ces constructeurs sont séparés par des blocs d'accolades. Dans ces accolades, on y trouve des instructions, chacune se terminant par ';'. Elles viennent compléter le constructeur qui accompagne le bloc, ce sont en quelque sorte les arguments qui sont progressivement passés au constructeur.

Ce langage supporte en outre deux types de commentaires : celui du langage C (/* */) et le commentaire de fin de ligne apporté par le C++ (//).

Un petit exemple de script

```
/* addition de deux nombres */
addition {
         30.5, 11.5; // additionne 30.5 et 11.5
         4, 5; // additionne 4 et 5
}

/* concatenation de deux strings */
concatenation {
         "Da Gniii", " Project"; // donne "Da Gniii Project"
}
```

Les constructeurs seraient ici "addition" et "concatenation" qui s'appliqueraient aux couples 30.5,11.5 et 4,5 pour l'addition et aux strings "Da Gniii", " Project" pour la concaténation.

Principe

Pour parser le fichier, il faut d'abord le récupérer dans une variable. L'erreur à ne pas commettre serait de stocker l'intégralité du fichier dans cette même variable, pour des raisons de mémoire.

Tout d'abord il ne sert á rien de récupérer les espaces (sauf pour une exception), les retours à la ligne et le contenu des commentaires.

Le fichier est lu caractère par caractère. Puisque les commentaires commencent, et même se ferment en utilisant deux caractères on va utiliser une file de deux caractères *ch1* et *ch2*. Par exemple, si *ch1* contient "/" et que *ch2* contient "*" alors on active le commentaire de ce type et on ne bufferise plus les caractères lus jusqu'à ce que *ch1* contienne "*" et *ch2* "/".

Puisque le langage fonctionne avec des constructeurs et des instructions, on peut travailler avec deux buffers qui les contiennent. On va tout d'abord récupérer le constructeur dans *current* jusqu'à rencontrer un *token* "{" La suite est en effet un ensemble d'instructions. C'est ensuite au tour du bloc d'être traité. Pour cela on va récuperer son contenu jusqu'à un *token* ";". Et cela jusqu'à ce qu'il n'y ait plus d'instructions, c'est à dire dès que *ch1* contient le token "}". Et on recommence jusqu'à arriver à la fin du fichier.

Dès que l'on récupére une instruction, on peut appeler une fonction qui va traiter cette instruction selon le constructeur courrant. Cette fonction, selon le constructeur fournit, appelle une autre fonction qui va parser l'instruction. Pour cela, il suffit de connaître l'ordre des arguments pour les typer, et de les séparer puisque chaque argument est espacé du *token* ",".

Future évolution

Même si pour l'instant le parseur ne peut qu'additionner des nombres ou concaténer des chaînes, le principe y est et il ne reste plus qu'à ajouter des constructeurs et les fonctions qui vont avec. Il permettra par exemple de charger des triangles, des mesh, des textures et peut-être plus tard de scripter des événements.

2.4 Moteur graphique

A la vue des possibilités offertes, nous avons choisi d'utiliser *Direct X* pour de nombreux aspects du projet. Pour la 3D, nous utilisons donc *Direct 3D*.

Pour l'instant nous ne sommes pas allés très loin dans son utilisation. Nous ne faisons que l'initialiser pour y afficher des modèles - ou mesh, des polygones et pour appliquer des transformations basiques comme les rotations, translations et homothéties.

L'utilisation générale de *Direct X* est maîtrisée, mais il faudra attendre la prochaine soutenance pour une réelle utilisation de cette puissante librairie.

De plus, le moteur graphique actuel souffre d'un gros défaut : il affiche exactement tout ce qu'on lui demande. La prochaine version du moteur graphique permettra, à l'aide d'un octree ou d'un BSP, de n'afficher que ce qui est visible et ainsi permettre un gain significatif de performances.

2.5 Moteur physique

Le but du moteur physique est de mettre à jour l'environnement, c'est à dire les objets (joueurs, projectiles) se trouvant dans la zone de jeu. Il doit non seulement être rapide (fréquence de 20 rafraîchissements par seconde environ), mais aussi être suffisamment réaliste pour ne pas perturber le joueur. Ainsi, il devra être capable de gérer les collisions entre deux objets, mais également la destruction d'objets hors-zone, ou la création de nouvelles instances (projectiles). Actuellement, les collisions n'ont pas encore été implémentées, mais le moteur est capable de déplacer des objets et de mettre à jour leur registre (dans l'octree) si besoin est. Ainsi, ajouter une gestion de collisions simple (sphère - sphère ou sphère - surface) ne devrait pas poser de problèmes.

2.6 L'octree

Introduction

Un jeu de type FPS doit être capable de gérer une multitude d'objets physiques et leurs intéractions. Bien qu'un monde dans lequel les balles et autres projectiles ne blessent personne serait fantastique, il est difficile d'imaginer un jeu vidéo négligeant délibérément cet aspect. Ainsi, il nous a fallu nous lancer dans l'énorme tâche qu'est la gestion des collisions. En terme simple, gérer les collisions dans un projet tel que le notre revient à :

- Prendre tous les objets un par un.
- Chercher tous les objets qui lui sont proches.
- Regarder si par hasard il y a collision avec l'un d'entre eux.
- Agir en conséquence (destruction, arrêt, rebond,...).

Seulement, la notion de proximité est tous simplement inexistante en informatique. Il faut l'implémenter d'une manière ou d'une autre, en gardant à l'esprit que le nombre de calculs, et donc la performance globale de l'application, décroît rapidement. Nous avons donc cherché une structure capable de donner, pour tout objet de notre espace de jeu, ce qui se trouve dans une certaine zone autour de cet objet, et qui donc est susceptible d'intéragir

avec lui. On peut distinguer trois structures susceptibles de répondre notre attente : le tableau, dont l'inconvénient majeur est qu'il nécessite un espace mémoire contigu important ; le tableau chainé, qui perd les avantages de l'accès direct ; et l'arbre, pour lequel les temps d'accès aux feuilles est fixe, mais qui demande un espace mémoire plus important. Pour pallier les problèmes d'espace mémoire, nous avons donc opté pour le bonzaï.

Le principe de l'octree

Il ferait mauvais genre de rappeler ici le principe d'un arbre, donc venons-en directement aux spécificité de l'octree. Contrairement aux arbres binaires, chaque sous-arbre ne possède non plus deux, mais huit fils.

Dans une représentation en 3D, il est facile de visualiser l'architecture d'un tel arbre : La zone de jeu, un simple cube, sera divisé en huit cubes de taille identiques qui formeront ses huit fils. Chacun d'eux sera ensuite divisé en huit, et ainsi de suite, jusqu'au niveau de précision désiré. Nous travaillerons ici sur un octree complet, dont toutes les feuilles sont au même niveau. Ainsi, la zone de jeux est divisée en une multitude de petits registres qui sont les feuilles de notre octree. Dans ces registre sont stockés des pointeurs vers tous les objets dont ils contiennent le centre. Ainsi, récupérer les voisins d'un objet donné revient lister tous les objets présents dans le même registre que celui-ci, et les registres adjacents si besoin.

L'implémentation

Pour des raisons pratique, notre implémentation a été adaptée à des besoins spécifiques, notamment la recherche d'une optimisation de temps de parcours. Ainsi, chaque feuille de l'octree connait la position de toutes les feuilles adjactente. De plus, l'orientation objet étant particulièrement gourmande en ressources, que ce soit vitesse ou temps de calcul, elle n'a été conservée que pour la structure globale : les millions potentiels de sous-arbres, ansi que les registres situés dans les feuilles (listes chainées), ont été recodés de manière indépendante de l'implémentation objet. Pour des structures de ce type, et présentes en si grand nombre, les procédés de vérification inhérentes à l'objet représentent une perte de performance de plus de 30%, pour un résultat absolument identique. Des procédures de création et de destruction ont bien évidemment été écrites afin de ne permettre aucune fuite de mémoire.

3 Nouvelle répartition du travail

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

3.1 Première soutenance

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

3.2 Seconde soutenance

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

3.3 Troisième soutenance

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

3.4 Dernière soutenance

En théorie, tout devrait être terminé.

4 Conclusion

Pendant les longues nuits passées à se creuser la cervelle sur des algorithmes de parcours d'arbre itératifs, nous avons beaucoup appris. Tant d'heures de code permettent de renforcer la confiance en soi... Et en le debuggant (enfin ce truc là. Comment c'est déjà ?).

De grandes avances ont ainsi pu être achevées, même s'il reste encore beaucoup de chemin à parcourir.

En des termes plus simples... Delphi, ca poutre ! Mais parfois aussi, ca pique... M'enfin on s'y est fait. Reste plus qu'à le commencer, le projet, maintenant.