Travail d'Étude et de Recherche

Axel BAUDET
Mathieu SCHNOOR

Université de Nice Sophia Antipolis
Mai 2009

Table des matières

Introduction **4**

Objectifs **5**

0.1 Type de jeu 5
0.1.1 Multijoueur et en réseau 5
0.1.2 Temps réel et physique 5
0.2 Exigences techniques 7
0.2.1 C++ 7
0.2.2 Plugins 7
0.2.3 OpenGL 8
0.2.4 Portabilité 8

1 Conception **10**

1.1 Plugins 10
1.1.1 Chargement dynamique d'objet 11
1.1.2 Fermeture d'un plugin 12
1.1.3 Restrictions 12
1.1.4 Modification de comportement 12
1.2 Réseau 13
1.2.1 Protocole 13
1.2.2 Implémentation 15
1.3 Méta-serveur 16
1.3.1 Description 16
1.3.2 Protocole 17
1.4 Moteur de jeu 17
1.5 Serveur 19
1.5.1 Organisation 19
1.5.2 Threading 20
1.6 Client 21
1.6.1 Core 21
1.6.2 Évènements 23

1.6.3 Rendu graphique 24

2 Réalisation 26

2.1 Moteur graphique 26

2.1.1 Rendu typographique 26

2.1.2 Moteur 3D 28

2.2 Efficacité du moteur physique 31

2.3 Synchronisation 32

2.3.1 Approche Novice 32

2.3.2 Approche Hacker 33

2.3.3 Approche Master 34

2.4 Validation de la couche réseau avec PUNIT 35

3 Résultats 39

3.1 Éditeur 39

3.2 Client 40

3.2.1 Gui 40

3.2.2 Jeu 41

Conclusion 45

A Protocole 46

A.1 Server side 46

A.2 Client side 48

B Statistiques 52

Introduction

Aujourd'hui en 2009, l'industrie du jeu vidéo a atteint un poids comparable à celle de la vidéo, et entreprend de la dépasser : 32 milliards de dollars de recettes en 2008 contre 28 pour les ventes de DVD et Blu-Ray confondues [1].

L'enjeu que représente le contrôle d'un tel marché laisse de moins en moins de place au développement de jeux « à l'ancienne », tel qu'il était massivement pratiqué au début de l'ère vidéo-ludique. Dans la logique des grands noms de l'industrie du jeu vidéo, les nouveaux titres suivent de très près les évolutions matérielles, et les conditionnent même, souvent à une seule fin : afficher des environnements plus détaillés, de meilleures textures et toujours plus de pixels à l'écran.

Cette quête du réalisme, voire souvent photo-réalisme – en elle même passionnante – se fait malheureusement parfois au détriment du jeu en lui même. Du temps où les développeurs disposaient de ressources très limitées, il fallait essayer de trouver des moyens de captiver le joueur en lui faisant oublier la pauvreté de l'environnement et la faiblesse des graphismes par la richesse ou la subtilité des mécanismes de jeu. Aujourd'hui, l'inverse se produit : l'évolution des capacités graphiques et le temps nécessaire au développement d'un moteur graphique, des environnements et des modèles y évoluant devient la tache la plus longue, limitant ainsi les évolutions du reste de l'application.

En opposition à cette doctrine du *jeu screenshot* apparaît depuis quelques années une scène du jeu alternative, ou *indie* : des concepts simples, efficaces et réalisés sans tenter d'en faire trop ; c'est ce qui a fait le succès par exemple du récent *World of Goo* [2].

C'est cet esprit que nous avons voulu recréer pour ce TER en étant à la fois dans une démarche d'apprentissage des techniques nécessaires à la réalisation du projet, mais également de produire un jeu moderne, performant et agréable.

1. http://www.numerama.com/magazine/copier/11832-Le-chiffre-d-affaires-du-jeu-video-depasse-celui-des-DVD-et-Blu-Ray.html
2. http://2dboy.com/games.php

Objectifs

0.1 Type de jeu

Si certains aspects particuliers du gameplay ou du moteur graphique ont été décidés au cours du processus de création du jeu en fonction du temps, de leur faisabilité ou de ce qu'ils apportaient au jeu, certains aspects fondamentaux ont été fixés dès le début.

0.1.1 Multijoueur et en réseau

La caractéristique principale du jeu que nous avons entrepris de créer est celle d'être multijoueur, et en réseau. C'est une contrainte forte qui va déterminer ce qu'il sera possible de faire dans le jeu, limiter certaines idées inapplicables à un tel environnement, mais aussi conditionner certains choix techniques ou encore imposer la création de certains utilitaires particuliers.

L'architecture de l'application est en conséquence celle du client / serveur. Un seul serveur par partie auxquels se connectent plusieurs clients.

Pour l'implémentation de la couche réseau, nous avons choisi l'utilisation directe de l'API des sockets Unix, au lieu de tenter de passer par une couche intermédiaire. Cela implique la conception d'un protocole réseau adapté et efficace, ainsi que son implémentation du coté client comme du coté serveur.

Enfin, la création d'un jeu en réseau sur une architecture client / serveur implique la création d'un méta-serveur : un serveur qui va référencer tous les serveurs de jeu actifs afin que les nouveaux clients puissent trouver un endroit ou jouer.

0.1.2 Temps réel et physique

Un jeu en réseau en tour par tour, ou dans lequel l'action est assez lente, comme un RTS [3], se révèle beaucoup plus tolérant aux latences ou à une mau-

3. Real Time Strategy

vaise implémentation du réseau qu'un jeu temps réel où l'action est rapide et où chaque décision du joueur doit provoquer une réaction immédiate.

Dans notre projet de jeu, plusieurs joueurs évoluent simultanément et en temps réel dans un environnement 2D : ils se déplacent sur le plan horizontal sur un terrain où ils peuvent rencontrer des obstacles, et doivent combattre les autres joueurs en leur tirant des projectiles.

On pourra citer en référence dans ce type de jeu TeeWorlds [4] ou Soldat [5], eux même inspirés par Liero [6], un cousin des premiers Worms [7] en temps réel. La différence principale se situe au niveau du point de vue isométrique, et non pas en pure 2D de coté, et de l'utilisation du moteur physique.

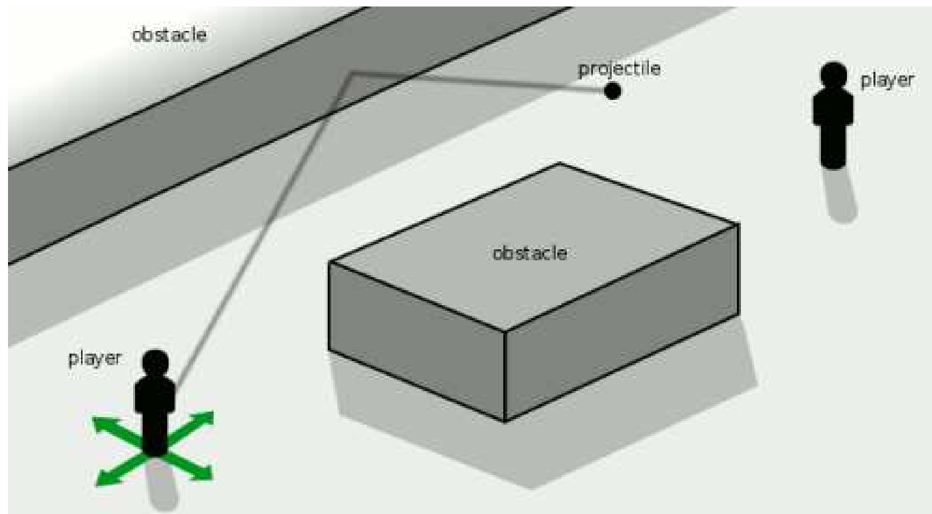


FIGURE 1 – Concept de jeu

Les joueurs, les projectiles et les obstacles sont des éléments du moteur physique. Les obstacles sont statiques, mais les joueurs et les projectiles sont des objets mouvants qui ont une masse, une vitesse et une direction, et qui lorsqu'ils rentrent en collision produisent un choc élastique : leur nouvelle direction et vitesse est dépendante de la vitesse, masse et direction des deux objets entrant en collision.

Cette définition est intéressante d'un point de vue du jeu en lui-même, mais il se pose alors le problème de sa mise en oeuvre dans le code qui n'est pas triviale, et ensuite de son application sur le réseau, qui supposera soit une

4. http://www.teeworlds.com/
5. http://www.soldat.pl/en/
6. http://en.wikipedia.org/wiki/Liero
7. http://en.wikipedia.org/wiki/Worms_(1995_video_game)

grande précision, soit une excellente capacité de prédiction : un tel moteur de jeu va typiquement être très exigeant d'un point de vue synchronisation sur le réseau. Ce point sera abordé en détail plus tard.

0.2 Exigences techniques

Au delà du système de jeu, nous avons au lancement du projet quelques exigences techniques vis-à-vis du projet.

0.2.1 C++

Le choix du langage semble principalement se poser entre deux langages, C++ et Java. Si nous avons opté pour C++, c'est pour trois raisons :

- Préférence personnelle : la gestion manuelle de la mémoire sans Garbage Collector, le choix du passage de paramètres par référence ou copie, le choix des optimisations faites à la compilation ou de l'édition de lien ; le programmeur ne peut pas le faire en Java, mais doit être beaucoup plus rigoureux en C++ pour pouvoir prétendre le faire correctement. C++ force le programmeur à être attentif à des détails qu'il aurait négligés dans une implémentation Java, nous pensons que c'est un avantage quand le choix est fait sciemment.
- Apprentissage : depuis le début, notre formation à l'Université se concentre sur Java. Des cours de programmation objet au génie logiciel, l'accent est très fortement mis sur ce langage, et très peu, en revanche, sur son aîné C++. Faire un projet conséquent en C++ est l'occasion de nous diversifier afin de pouvoir apprécier les avantages et inconvénients.
- Performances : un jeu vidéo est une application où les performances sont critiques. Nous nous plaçons volontairement bas niveau pour des raisons de contrôle des performances et de l'optimisation qu'il sera possible de pratiquer. En utilisant C++, nous sommes sûr d'avoir tous les outils à notre disposition pour faire un jeu le plus rapide et le moins gourmand possible en ressources.

0.2.2 Plugins

Tous les logiciels populaires récents disposent d'une interface de plugins plus ou moins évoluée, qu'ils soient écrits en Java (Eclipse) ou en C++ (Firefox).

Disposer d'un gestionnaire de plugins simple et puissant nous a donc semblé un pré-requis important, qu'il soit utilisé pour des fonctions secondaires,

comme afficher des nouveaux widgets dans l'interface graphique, ou encore des capacités intrinsèques au coeur du programme, comme le moteur de jeu ou le moteur graphique.

La possibilité d'utiliser le gestionnaire de plugins pour changer, à chaud, de moteur graphique, semble à priori séduisante.

0.2.3 OpenGL

Plutôt que de tomber dans la facilité d'utiliser moteur graphique tout fait comme de nombreux existent déjà (Ogre [8], Irrlicht [9]), nous avons choisi de programmer notre moteur graphique entièrement nous même, exclusivement avec l'API OpenGL.

Le choix de se placer dans une implémentation bas niveau, de limiter au minimum les exigences matérielles et de ne pas privilégier la complexité des graphismes à la richesse du jeu nous a conduit tout naturellement vers ce choix, avec également la volonté d'apprendre et comprendre les mécanismes de bas niveau mis en jeu.

Ce choix nous permettra de contrôler finement les performances de cette partie de l'application, plutôt que d'utiliser un moteur tout fait qui en fera forcément beaucoup plus que ce que nous en attendons, limitant ainsi les performances.

Le moteur implémenté sera à priori un moteur 2D.

0.2.4 Portabilité

Le dernier pré-requis technique est la portabilité.

Développant notre application sous GNU/Linux, nous sommes conscients que la cible principale, les joueurs, n'utiliseront pas ce système d'exploitation. Ayant exclu la possibilité de développer directement notre projet sous MS-Windows, nous devons donc garantir sa portabilité, en utilisant uniquement des librairies garantissant cette propriété, et en programmant selon les normes ISO C99.

Toutes les librairies utilisées sont portables :

- l'API Sockets
- les threads POSIX
- la librairie multimédia SDL [10]
- la spécification OpenGL

8. http://www.ogre3d.org/
9. http://irrlicht.sourceforge.net/
10. http://www.libsdl.org/

- les quelques autres qui seront définies plus tard dans l'évolution du projet
- API de chargement dynamique de code pour le plugins

Lors du début du projet, un de nos objectifs sera, parallèlement au développement sous GNU/Linux, de fournir un portage fonctionnel pour la plate-forme MS-Windows.

Chapitre 1

Conception

Seront présentés dans ce chapitre les choix de conception relatifs aux différentes parties des applications développées.

Ces choix de conception sont présentés non pas dans l'ordre d'importance mais dans l'ordre de dépendances : le système de plugins sera présenté en premier, car l'implémentation du protocole en dépend, de même que le client et le serveur dépendent de l'implémentation réseau.

On présentera dans cette section l'application de manière globale et complète dans l'approche qui a été faite pour la construire, pas les moyens techniques précis mis en oeuvre, qui seront détaillés de manière moins exhaustive et lorsque cela présente un intérêt, dans le chapitre suivant.

1.1 Plugins

Le gestionnaire de plugin, plutôt que d'être une sucrerie permettant l'accès à des fonctionnalités très optionnelles comme dans Firefox, est dans le projet un élément fondamental.

Ce gestionnaire de plugin doit pouvoir gérer des composants internes à l'application : le moteur de jeu, le moteur de rendu, etc. Il est donc important qu'il soit fiable et générique. Le gestionnaire de plugins permet :

- Chargement de fichiers ELF dynamiques [1]
- Grâce à un point de sauvegarde, recharger à chaud n'importe quel plugin, par exemple lors d'une mise à jour
- Fermeture propre d'un plugin bien formé, ou destruction forcée
- Pouvoir threader le plugin s'il le permet, afin qu'il puisse exécuter des appels bloquants

1. Executable and Linking Format, *.so

Chaque plugin est un fichier ELF compilé à partir d'une classe C++ dérivée de la classe Plugin de base fournie par l'API de plugin du projet (Fig. 1.1), selon un design pattern semblable au *Template Method*.

```
class Plugin {
  public:

  Plugin();
  virtual ~Plugin();

  virtual bool init() = 0;

  virtual bool mainfunc() = 0;

  virtual bool threaded();
  virtual bool canResume();

  virtual void loadstate(void *lstate);
  virtual void *savestate();

  virtual void askStop();
  virtual bool hasStop();
  virtual bool hasAskedStop();

  virtual void stop();
  virtual bool resume();
};
```

Figure 1.1 – classe Plugin générique

1.1.1 Chargement dynamique d'objet

Le Plugin Manager charge des classes C++ en guise de plugin : chaque classe doit étendre `Plugin` afin d'avoir le squelette minimal que le Manager demande pour pouvoir lancer une instance du Plugin. Les plugins bloquants, comme par exemple une boucle d'acceptation de connexions entrant sont automatiquement lancés dans un nouveau thread. On peut aussi, si l'on veut, redéfinir ce comportement, et faire threader des plugin non bloquant et inversement.

Le Manager sait aussi stopper, relancer, fermer et tuer un plugin. On peut alors, avec des plugins *résumables*, sauvegarder l'état courant d'un plugin lors du déchargement et le récupérer lors du rechargement d'un autre. Cette méthode est toutefois limitée car la sauvegarde et la restauration doivent être programmées manuellement. En effet, si l'on prend par exemple le plugin du serveur qui accepte les connexions ; il ne doit pas fermer la socket serveur lorsqu'il enregistre son état et qu'il est déchargé. Cela provoquerait la déconnexion de tous les clients actuellement connectés : il n'y a donc pas de sauvegarde automatique possible pour ce genre de plugin.

1.1.2 Fermeture d'un plugin

Lorsque l'utilisateur le désire, ou lors de la destruction du Plugin Manager, ce dernier peut alors fermer un Plugin proprement (responsabilité au programmeur). Si le programmeur n'a pas permis au plugin de pouvoir se fermer (avec par exemple une boucle infinie sans condition d'arrêt), le Manager peut alors tenter de tuer et décharger le plugin avec une annulation de thread brutale. Il est conseillé alors au programmeur de faire un plugin qui puisse se fermer proprement car l'annulation de thread peut occasionner une erreur de segmentation.

1.1.3 Restrictions

Comparé au ClassLoader de java qui ne sait pas décharger une classe déjà chargée ou même charger deux classes du même nom (conséquence de la convention de nommage en java), le Plugin Manager peut lui charger des librairies ayant le même nom située à des endroits différents et aussi recharger des classes ayant le même noms très facilement. Le Plugin Manager peut aussi décharger bien plus facilement des classes.

Les avantages du Plugin Manager en C++ sont donc les mêmes que ses inconvénients et sont liés directement au langage d'implémentation : beaucoup de liberté et flexibilité mais cependant aucune vérification possible, et donc, tout est la responsabilité du programmeur (boucles, erreur de segmentation, mémoire non libérée, etc.).

1.1.4 Modification de comportement

La modification de comportement est ce qui permet d'accroître les possibilités d'un plugin afin qu'il puisse entièrement s'intégrer au code de manière transparente et générique.

Certains plugins sont de type passif : le plugin chargé via la librairie `libAccept.so`, par exemple est en écoute permanente de connexion, et lorsque celle ci arrive l'enregistre dans la liste des clients (si le serveur n'est pas plein, et s'il n'y a pas eu d'erreur) afin d'être utilisé autre part dans le code. D'autres sont actifs, comme par exemple `libComm.so`, qui boucle sur les envois et réceptions réseaux en permanence, ou `libStep.so` qui, lui, boucle constamment sur le moteur du jeu.

La modification de comportement, elle, vise un type de plugin plus spécial. Lorsqu'un plugin est chargé via le Plugin Manager, il n'y a aucun moyen de le distinguer génériquement d'un autre afin de lui faire accomplir une tâche spéciale interne au serveur et déjà bien définie. On peut alors déclarer un prototype de méthode comme par exemple :

```
int (*proto_OnAttack)(Player *, Player *, Weapon *)
```

Et déclarer la méthode correspondante :

```
proto_OnAttack Attack = NULL;
```

Le plugin pourra alors maintenant, contrairement aux autres, modifier le comportement du programme et de cette méthode, en y enregistrant l'adresse de la sienne directement (en respectant le prototype associé) (Fig. 1.2).

1.2 Réseau

1.2.1 Protocole

Le protocole de communication est utilisé pour échanger des messages entre le serveur et ses différents clients.

Le but recherché pendant la conception du protocole est la simplicité et la rapidité de traitement ; il est fréquent de voir certains protocoles passer la majeure partie de leur temps à parser les requêtes émises par le client, et inversement.

Les protocoles usuels reçoivent généralement des requêtes dont la trame approche :

```
TYPE [Paramètres]
```

Comme par exemple le protocole IRC [2]. Cette structure très simple correspond bien à ce qui est recherché ; mais deux choix s'opposent alors en ce qui concerne les performances :

2. `http://www.ietf.org/rfc/rfc1459.txt`

```
double (* proto_Operation)(double a, double b);
proto_Operation Operation = NULL;

[Plugin sumOp.so]
double sumOp(double a, double b) {
  return return a + b;
}
programme.Operation = &sumOp;

Operation(2, 3); // == 5

[Plugin divOp.so]
double divOp(double a, double b) {
  return return a / b;
}
programme.Operation = &divOp;

Operation(6, 3); // == 2
```

Figure 1.2 – Modification de comportement par Plugin

- Un rapide, qui consiste à évaluer lettre par lettre le type de la requête entrante (ce qui serait rapide, mais peut lisible dans le code)
- Un plus lent, qui consiste à évaluer mot par mot le type de la requête (ce qui serait plus lisible, mais moins intéressant)

Nous avons donc la simplicité du protocole, mais cependant, pas les performances.

La solution est amenée par une utilisation intelligente d'une fonctionnalité de C/C++ : le pointeur de fonction. Il est possible d'assigner des pointeurs de fonctions à des chaînes de caractères entières, mais nous avons choisi de le faire sur des entiers, ce qui correspond plus à la logique de la méthode utilisée.

Le protocole prend donc la forme :

```
<x,y> [TYPE] [Paramètres]
```

Où `x` et `y` sont des nombre allant de 0 à 255 (un octet). Il suffit alors d'appeler la fonction enregistrée dans `Protocole[x][y]` où Protocole est une table de pointeurs de fonctions pour assigner une fonction à une requête. Deux octets supplémentaires sont donc ajoutés à chaque requête pour reconnaître la requête entrante *en temps constant*, là où d'autres algorithmes moins

performants auraient reconnus cette dernière en temps linéaire, en fonction du nombre de requêtes connues.

L'argument `[TYPE]` de la requête devient obsolète selon ce schéma, mais il est tout de même conservé afin de pouvoir garder une trace lisible des requêtes effectuées en cas de débugage, mais également à pouvoir tester l'implémentation du protocole, comme décrit plus loin.

Le protocole de communication complet est disponible dans la partie Annexe.

1.2.2 Implémentation

Le choix du protocole de transport de données n'est pas trivial, chaque méthode comportant son lot d'avantages et d'inconvénients. pour les communications clients / serveur, notre choix s'est porté sur `TCP`.

Ses avantages sont les suivants :
- Intégrité des données
- Accusé de réception
- Le mode connecté garanti l'identité de l'envoyeur dans une certaine mesure
- ordonnancement des paquets

Tous ces éléments sont essentiels ; un message non transmis risque de briser la cohérence du jeu, le mode non connecté force l'implémentation de la couche réseau à comporter un système d'authentification de l'émetteur, et l'ordre des paquets est important, par exemple quand il s'agit d'initier une nouvelle partie.

`UDP`, qu'on préfère généralement pour les jeux en raison de sa rapidité, ne fournit rien de tout cela. Chaque paquet est reçu une fois sans ordonnancement, les routeurs n'auront alors pas à attendre que les `n` premiers paquets arrivent pour envoyer le paquet `n+1`. En plus de cela il n'aura pas besoin de demander la ré-émission des paquets perdus.

Choisir UDP pour sa vitesse reviendrait à lui ajouter par dessus un mécanisme d'accusé de réception et de contrôle d'intégrité pour les messages importants, et à contrôler l'ordre de certaines séquences de requêtes, ce qui reviendrait à implémenter en quelques jours certaines des caractéristiques d'un protocole vieux de 28 ans : rien ne garantit que UDP + une sur-couche incomplète sera plus rapide et performant que TCP.

Les problèmes de congestion du réseau qui font échouer TCP dans les jeux, au demeurant, s'appliquent également à UDP, même s'ils sont exacerbés dans TCP à cause de l'ordonnancement des paquets : si le réseau est lent ou perd des paquets arbitrairement, TCP va provoquer une latence excessive en délayant les paquets suivant le paquet perdu, mais UDP ne résoudra rien

car il perdra lui aussi le paquet et il faudra probablement également le réémettre. Quand internet ne marche plus, aucun protocole ne peut nous aider, il faut jouer tout seul.

Dans le cas où l'avenir donnerait tort à ce choix de conception, la grande modularité de la couche réseau permet de changer l'implémentation de l'envoi des messages très facilement, sans aucun impact sur le code du client ou du serveur.

1.3 Méta-serveur

1.3.1 Description

Ce serveur secondaire est destiné à faciliter le référencement des parties présentes sur tous les serveurs de jeu publics lancés.

Le but est évidemment de faciliter l'accès des parties aux joueurs : il permet un référencement clair, précis avec des informations telles que le nom de la carte lancée couramment sur le serveur, le nombre de joueurs connectés actuellement, l'adresse et le port de destination pour le client afin qu'il puisse s'y connecter.

Les serveurs esclaves rafraîchissent leur état par intervalle donné régulier afin que le serveur maître tienne à jour une liste exhaustive de ces derniers : c'est le *Heart Beat*.

Le système de *Heart Beat* a été implémenté en tant que plugin, améliorant la modularité et permettant du côté serveur, très facilement (en désactivant/-réactivant le plugin) de passer automatiquement de mode privé à public. Sa simple conception permet d'étendre facilement toute nouvelle information que l'on voudrait (ou que l'on va) ajouter par la suite, comme par exemple le mode de jeu ou la version du serveur (afin de prévenir d'éventuels problèmes de compatibilité).

Un des plugins effectue aussi une modification de comportement comme évoqué dans la section Plugin afin d'automatiser la purge des serveurs inactifs (fermés ou rendu privés) et, la purge des serveurs inaccessibles. Il est évident qu'un serveur contactant le serveur maître et se trouvant sur un réseau privé ne peut être contacté par un quelconque client se trouvant hors du réseau en question. Un serveur se trouvant sur le même réseau que le serveur maître, même si ce dernier est accessible de l'extérieur, ne sera en mesure de donner une autre IP qu'une IP privée au serveur maître ; il y a pour cela un système de gestion d'IP externes. Les serveurs peuvent s'ils le désirent spécifier un nom de domaine pour simplifier la retenue de leur adresse pour la raison citée plus haut. Le plugin est alors intéressant afin d'éliminer les adresses qui

auraient été, intentionnellement ou pas, mal écrites.

Un mécanisme d'exportation automatique (le Plugin HTTP) de la liste des serveurs actuellement lancés est également disponible. Cette liste, sérialisées sous format XML, peut être récupérée via un script (par exemple celui en PHP que nous fournissons), afin d'afficher la liste des serveurs publics. On peut alors, via cette liste, créer des statistiques avancées sur les serveurs en récupérant les scores internes au serveur (pseudonyme, score, temps de connexion, utilisation des maps).

1.3.2 Protocole

La couche réseau serveur / serveur maître utilise `UDP` ; lorsque le serveur envoie un battement de coeur au serveur maître, cela ne fait aucune importance si ce dernier le reçoit ou pas, si le paquet n'arrive pas à destination, le suivant dans les quelques secondes à venir y arrivera. Il est aussi intéressant dans ce cas d'utiliser l'`UDP`, car si le serveur maître est inaccessible ou non joignable pour diverses raisons, le mode non connecté de l'`UDP` procure une robustesse supérieure dans ce cas que le mode connecté en TCP : les paquets seront tout simplement *droppés*.

Enfin la couche réseau client / serveur maître de la récupération de la liste des serveurs a elle été faite en TCP. Dans ce cas, la recherche de performance est sans objet et n'a donc pas été recherchée : on peut se permettre un d'utiliser `TCP` pour garantir que l'affichage de la liste des serveurs toujours complet.

1.4 Moteur de jeu

Le moteur de jeu est une des briques de base de l'application. Il n'y a aucune notion d'appel réseau ou d'affichage au sein du moteur, il ne fait que gérer des données indépendemment de leur représentation, et est complètement séparé du reste de l'application avant d'y être intégré.

Le moteur a plusieurs fonctionnalités :

- Chargement de la zone de jeu (ou *map*)
- Interactions physiques
- Gestion des actions des joueurs

Le moteur de jeu ne gère pas le joueur en lui même, mais un `Character`, que contrôle le joueur. La contrôleur réseau seul connaît le joueur et le `Character` associé, mais le moteur de jeu ne gère qu'au final un liste d'objets physiques (`Collidable`) pouvant entrer en collision, et grâce à la liaison

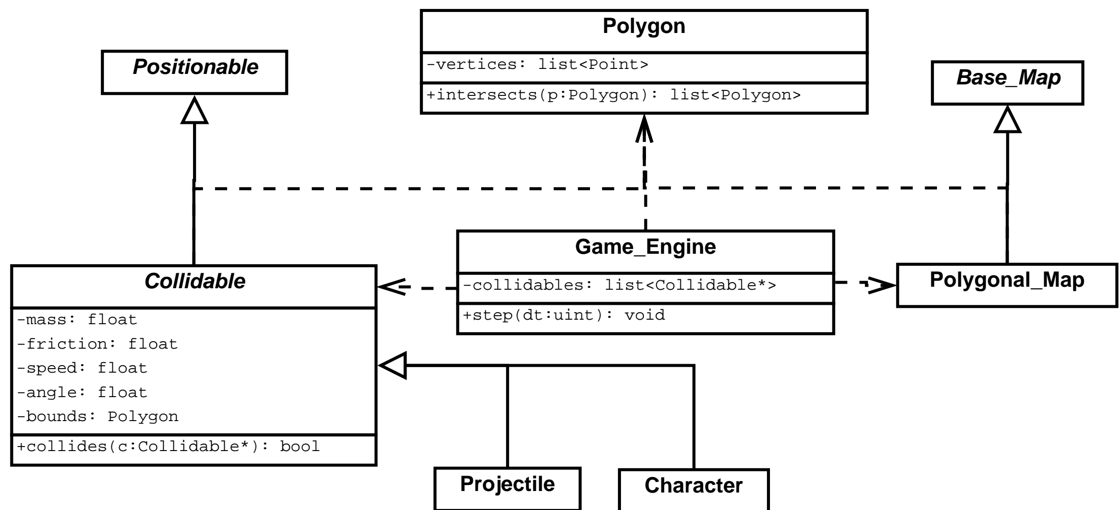


Figure 1.3 – Organisation du moteur de jeu

dynamique de C++ [3], les types concrets s'y référant.

L'utilisation typique du moteur de jeu est d'intégrer dans une boucle un appel à `Game_Engine : :step(dt)`. Cette opération va faire évoluer tous les objets en mouvement connus du moteur, les `Collidable`, d'une distance proportionnelle à leur vitesse et au temps écoulé depuis le dernier appel à la fonction. Éventuellement, les objets vont changer de direction et de vitesse en cas de collision.

Pour détecter une collision entre deux objets, on les représente dans le moteur physique comme une liste de points : un polygone. La collision entre deux objets mouvants ne nécessite pas le recours au polygone englobant : en considérant les objets en mouvement comme approximativement sphériques, un simple test en fonction de leur distance et de leur rayon permet de détecter la collision, et d'y appliquer un choc élastique (Fig. 1.4).

Le recours au polygone englobant est justifié pour les collisions avec la zone de jeu : représentée dans la classe `Polygonal_Map` comme une liste de polygones, on considère alors qu'il y a collision lorsque une arête de l'un des polygones est en intersection avec une arête de l'autre polygone. Le choc n'est cette fois ci pas élastique par souci de simplification ; obtenir une simulation physique précise n'est de toutes façons pas le but, mais simplement d'en donner l'illusion.

3. dynamic_cast<Character*>(Collidable*) par exemple

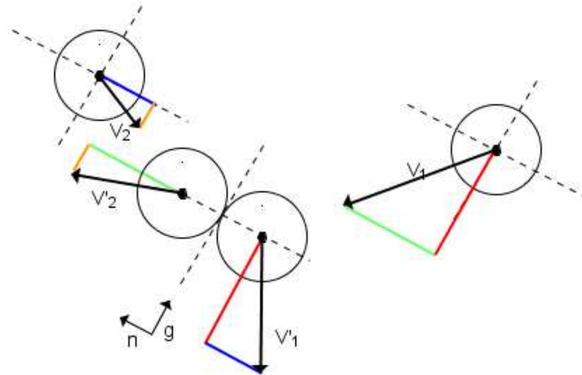


Figure 1.4 – Choc élastique

1.5 Serveur

1.5.1 Organisation

Le serveur de jeu principal est très fortement lié au gestionnaire plugins, et charge 6 plugins au démarrage (Fig. 1.5.1).

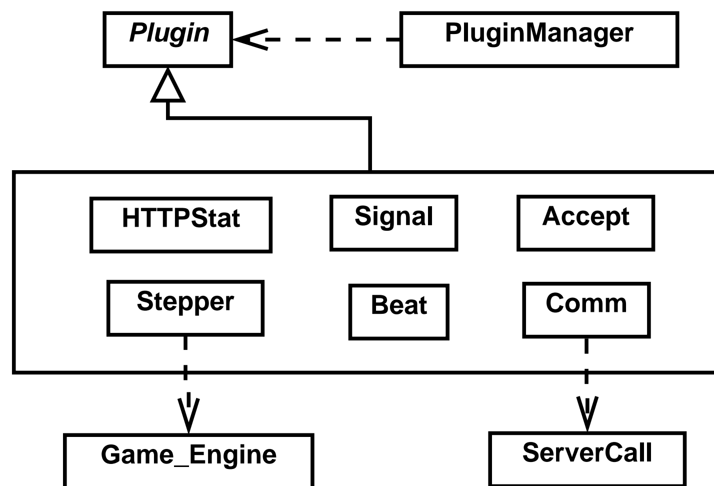


Figure 1.5 – Organisation du serveur de jeu

Certains de ces plugin exercent des fonction maîtresses au sein du serveur :

- Signal : Gestion des signaux : interruption ou destruction d'un plugin, fermeture du serveur
- Accept : Boucle d'acceptation des connexions des nouveaux clients

- Comm : gestion des appels réseau, interfacée avec le contrôleur réseau principal coté serveur, `ServerCall`
- Beat : *Heart Beat* ; inscription au Master Server
- Stepper : exécution du moteur de jeu
- HTTPStat : mise à jour de statistiques du serveur (nombre de joueurs, map, etc) sérialisées en HTTP sur un port spécifique

Cette organisation tout en plugins rend le serveur extrêmement flexible et modulaire. En remplaçant certains plugins vitaux comme `Comm` ou `Stepper`, on peut changer complètement de serveur sans modifier de code ! Il est tout à fait possible d'un point de vue théorique de faire de ce serveur un serveur pour un autre jeu communiquant avec un protocole différent.

1.5.2 Threading

Le serveur s'exécute sur 2 threads distincts :

- Le thread d'acceptation des clients : accept.
- Le thread d'envoi et de réception des données : send/recv.

L'envoi et la réception des données est placé dans un seul thread pour plusieurs raisons ; sans ça, pour `n` clients connectés, on aurait :

- `n` threads : un pour la réception et l'envoi pour chaque client, il est effectivement un peu inapproprié pour des entrées/sorties d'avoir 1000 threads pour 1000 clients connectés.
- $n * k$ accès simultanés et concurrents sur k ressources ; en effet, à partir du moment ou l'on thread les clients, ce n'est plus seulement une liste de client qu'il faut synchroniser mais chaque ressource potentiellement accessible en écriture par les clients. Ces ressources concernent en particuliers les informations propres aux clients (pseudonyme, statistiques, position, scores, etc...) et les ressources globales.

Les accès simultanés sont non seulement contre-performants dans ce genre de serveur *actif* et *dynamique* car les états des joueurs (bloqués, ralentis, plus de munitions) changent souvent mais parce qu'également, cela implique, lorsque c'est plus performant qu'une version mono-thread, une très bonne compréhension et gestion des mutex et autres principes qui n'empêche pas les programmeurs les plus avertis, de commettre des erreurs d'interbloquage. En plus de ces deux threads, il existe d'autres threads dépendant du nombre de plugins ajoutés au serveur. Bien entendu, ces derniers ont accès au mutex sur la liste des clients, et aux ressources associés.

1.6 Client

Le client est bâti autour de trois composants principaux :
- l'interface graphique utilisateur, permettant le listing des serveurs via le méta-server, la configuration du client ou la personnalisation du personnage contrôlé par le joueur
- le thread d'évènements, où est lancé le contrôleur réseau qui va recevoir des informations du server, ou les récupérer depuis les entrées de l'utilisateur
- le thread graphique où est construit le contexte OpenGL, et qui utilise une version locale du moteur de jeu pour afficher l'état courant

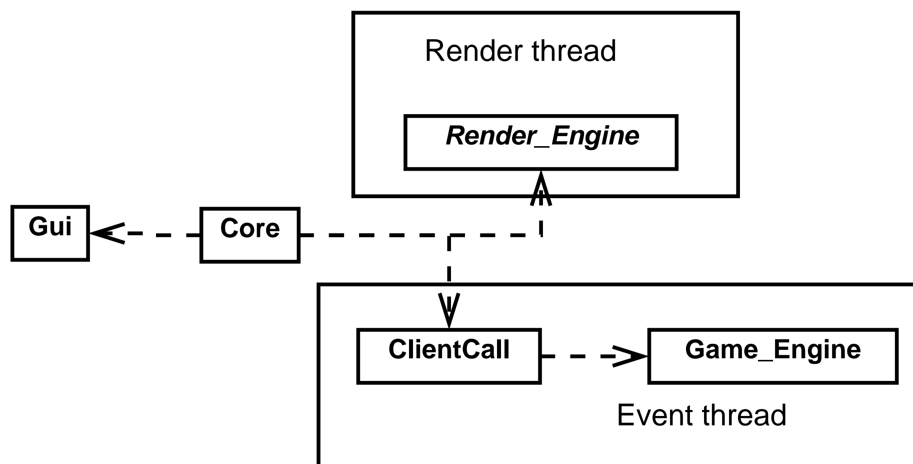


Figure 1.6 – Organisation générale du client

1.6.1 Core

Le `Core` est le composant principal du client. Il va rassembler tous les autres composants et les faire interagir ensemble.

En premier lieu, il va lancer le Gui : cette interface graphique va récupérer des informations depuis le client, principalement le serveur de jeu sur lequel il veut se connecter. Lorsque le Gui disparaît, il est détruit, et ne réapparaîtra que si le joueur quitte le serveur auquel il s'est connecté. Cela simplifie le protocole réseau en empêchant le joueur de changer trop sensiblement d'état, mais également le moteur de rendu, qui n'a pas besoin d'afficher un Gui à l'intérieur de son propre contexte, en devant lui rediriger ses entrées d'évènements.

Une fois le Gui disparu, le Core va lancer deux threads : le thread de rendu graphique, et la boucle d'évènements principale. Pour leur permettre de communiquer, le Core va utiliser la classe `Command_Queue`, qui est une implémentation du pattern *Command* (Fig. 1.7).

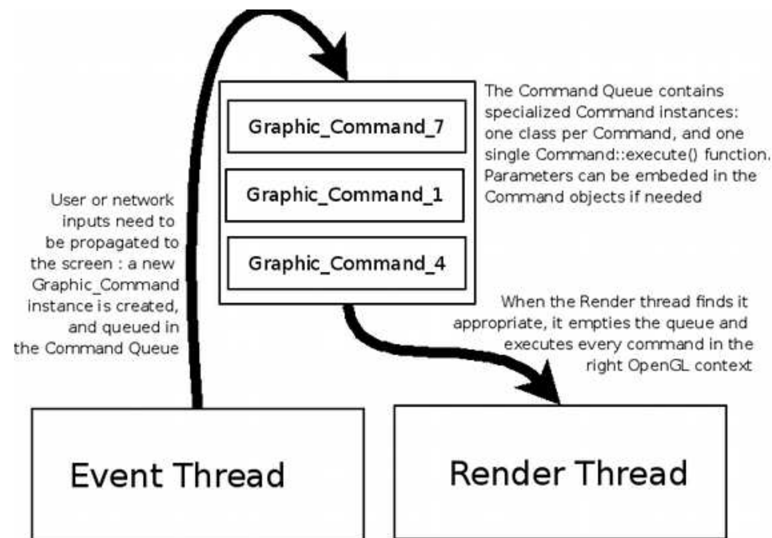


FIGURE 1.7 – Communication inter thread du client

L'utilisation de ce mécanisme ne devrait pas être nécessaire dans la mesure où les deux threads partagent la même mémoire et devraient pouvoir y accéder sans conflit, pourvu que l'accès aux ressources soit bien compartimenté avec des mutex. Malheureusement, OpenGL tolère très mal qu'on interagisse avec son état depuis un thread séparé ; le thread d'évènements ne peut donc pas directement modifier l'état du thread de rendu.

Cette *file de commandes* permet, depuis la boucle d'évènements, d'envoyer des commandes spécialement construites pour le moteur de rendu, afin qu'il les exécute dans son propre contexte, au moment où il a accès aux ressources. Voici par exemple la commande qui permet d'effectuer une capture d'écran :

```
class GC_Screenshot : public Command {
public:
  const char *path;

  GC_Screenshot (const char *p) :
    Command("GC_Screenshot"),
    path(p) { }
```

```
  ~GC_Screenshot() { }

  void execute (void *args) {
    Render_Engine *eng = (Render_Engine*) args;
    screenshot(eng->get_width(),
               eng->get_height(), path);
  }
};
```

Les arguments sont encapsulés dans l'objet `Command`, et exécutés ensuite par la `Command_Queue` où elle a été ajoutée :

```
  if (!is_empty()) {
    Command *c = queue.front();
    queue.pop_front();
    c->execute((void*)engine);
  }
```

1.6.2 Évènements

Le thread d'évènements embarque deux composants principaux : une copie locale du moteur de jeu, et le contrôleur réseau côté client.

D'une part, l'utilisateur va effectuer des actions qui seront transformées depuis ses périphériques d'entrée (clavier, souris) en actions sur le contrôleur réseau qui les transmettra au serveur, et d'autre part, le serveur va envoyer des messages dont l'effet sera appliqué sur la copie locale du moteur de jeu, et propagé à l'écran par le moteur de jeu qui dispose d'une référence sur les données gérées par le moteur local.

Le client peut interagir depuis 3 états distincts :

- Spectateur : il ne participe pas au jeu mais peut l'observer, et demander à rejoindre la partie
- Joueur : il interagit avec le jeu coté serveur et l'effet de ses actions est propagé aux autres clients
- Console : le client dispose d'une console qui lui permet de discuter avec les autres joueurs (publiquement, par équipe ou de façon privée), ou alors d'interagir avec le serveur de jeu en utilisant la `range` d'administration du protocole, accessible après authentification : changement de map, exclusion d'un joueur, etc.

1.6.3 Rendu graphique

Généricité

Le moteur graphique, écrit entièrement avec l'API OpenGL, a été conçu le plus modulaire possible afin d'être chargeable à volonté dans le contexte du gestionnaire de plugins. La classe abstraite `Render_Engine` fournit les mécanismes de base du rendu, avec l'affichage des messages, de la console, ainsi que d'un OSD [4] basique pour informer le joueur des dernières actions, de messages importants, ou encore son état de santé ou des objets qu'il possède actuellement ; toutes ces informations sont rendues par le moteur abstrait indépendemment du moteur concret instancié dans le `Core`.

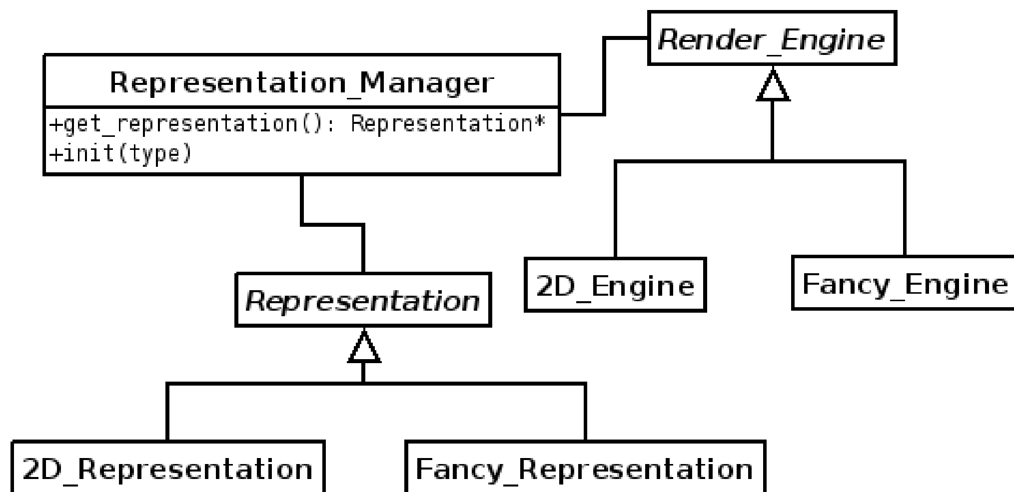


FIGURE 1.8 – Conception du moteur graphique

Représentation

Afin de pouvoir supporter cette généricité, la façon dont sont représentés les objets du jeu dans le moteur graphique doit être dissociée du-dit moteur ; on ne va pas charger les mêmes textures dans un moteur 3D utilisant la perspective ou dans un moteur en vue 2D isométrique.

Ainsi, pour accéder à la représentation d'un élément donné depuis le moteur graphique, on demande au gestionnaire de représentation (Fig. 1.7) d'accéder à une représentation concrète liée au moteur graphique actuellement utilisé.

4. On Screen Display

Ce gestionnaire de représentation est initialisé une fois en fonction du moteur chargé, selon le design pattern *Abstract Factory*, et on peut ensuite accéder directement à la bonne instance de `Representation` via le pattern *Singleton*.

Cette conception permet de dissocier entièrement les données de leur représentation, mais également de séparer différentes représentations en fonction du moteur, ou encore de réutiliser la même représentation pour plusieurs moteurs différents, et inversement.

Chapitre 2

Réalisation

Dans ce chapitre, on présentera en détail la réalisation technique de certains aspects du projet dignes d'intérêt, ainsi que du processus de tests mis en oeuvre pour valider l'implémentation de la couche réseau.

2.1 Moteur graphique

Le moteur graphique est l'une des parties les plus copieuses côté client, probablement après la partie synchronisation qui est abordée plus loin.

2.1.1 Rendu typographique

La surprise est assez déplaisante lorsqu'on remarque que OpenGL ne dispose d'aucun moyen standard de faire du rendu typographique, ou dit plus simplement, d'afficher du texte. On dispose alors de plusieurs options :

– Afficher directement aux coordonnées de l'écran une image matricielle, ou *bitmap*. Assez simple puisqu'OpenGL fournit les primitives, on parle alors de *Font Rasterization*. Malheureusement, cette technique est très lente : les données de l'image doivent être calculées par le `CPU`, puis renvoyées au `GPU`, de plus, cette technique ne permet ni d'étirer, ni de faire pivoter le texte.
– Utiliser une librairie qui va se charger de tout pour nous, sans trop rallonger la liste des dépendances ou alourdir le client ; SDL_pango [1] pourrait faire l'affaire mais n'est pas fait pour interagir directement dans un contexte OpenGL : il faudrait sans cesse convertir des surfaces SDL en textures OpenGL, ce qui se révèle impraticable en terme de performances.

1. `http://sdlpango.sourceforge.net/`

- Police géométrique : dessiner chaque glyphe en utilisant les primitives de base d'OpenGL : lignes et triangles, principalement. Les avantages sont nombreux : étirement, rotation, extrusion, éclairage ou même texturage des glyphes. Par contre, le nombre de triangles à afficher pour une simple chaîne de caractères peut se révéler très important : encore une fois, les performances sont mauvaises, sans compter que concevoir une telle police de caractère peut se révéler long et difficile.
- Quadrilatères texturés : chaque glyphe est un *Quad* auquel on plaque une texture représentant le caractère voulu, préalablement chargé en mémoire. Les implémentations OpenGL actuelles sont particulièrement efficaces pour ce genre d'opération, cette technique est par conséquent extrêmement efficace. De plus, on peut ainsi appliquer à chaque glyphe la couleur qu'on veut, les tourner dans n'importe quel sens et évidemment les étirer dans la mesure permise par la résolution des textures utilisées.

La technique choisie et implémentée dans `Render_Engine` est la dernière, *texture-base fonts*, en utilisant une images de $32 * n$ pixels de largeur et $4 * n$ de hauteur (où n est la largeur de la police en pixels), obtenant ainsi une texture pour chaque caractère de la table `ASCII` (Fig. 2.1).

```
 ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

Figure 2.1 – Exemple de police 14pt

En utilisant une *font map* pour le remplissage des caractères et une pour le contours, on peut ainsi utiliser n'importe quelle couleur de glyphe en lui mettant un contour noir pour qu'il soit visible où qu'il soit affiché (Fig. 2.2).

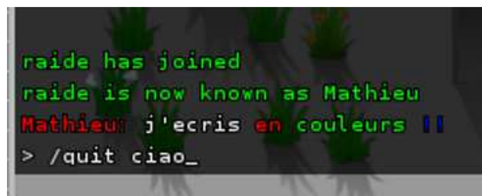


Figure 2.2 – Rendu de la console

Les limitations actuelles de cette technique sont assez évidemment le fait que seuls les caractères `ASCII` sont gérés. Une version internationalisée de-

manderait non seulement de devoir créer des images à peu près 3 à 4 fois plus grandes pour contenir la majorité des caractères locaux, mais également d'avoir un système fiable permettant d'associer à un évènement clavier le bon caractère, ce qui n'est pas trivial dès lors qu'on sort du standard ASCII.

2.1.2 Moteur 3D

Comme mentionné plus tôt dans ce document, le projet s'orientait initialement vers la création d'un moteur exclusivement 2D. Le premier prototype de moteur graphique était en 2D, et cela convenait. C'est quand il a fallu ajouter au moteur l'affichage des personnages et leur animation, que le vrai problème s'est posé.

La création d'un moteur 2D s'était en premier lieu imposée pour des raisons de simplicité : il apparait beaucoup plus aisé d'écrire un simple moteur 2D isométrique qu'un moteur 3D où il faut créer des modèles 3D, les charger, les éclairer, lisser et texturer pour ne pas subir de `Polygon Aliasing`[2] trop important, et évidemment les animer avec un squelette auquel on attribue des groupes de vertices pondérés.

Seulement, le moteur 2D vient lui aussi avec son lot de désagréments : il faut passer par un système de sprites, ce qui créé des artefacts graphiques plus important que le polygon aliasing : les changements de direction lorsqu'on passe brusquement de la sprite orientée sud à celle orientée sud-est, par exemple. En outre, créer de belles sprites requiert également de créer un modèles 3D et de l'animer avant de rendre séparément quelques images clefs, ou d'être un dessinateur de talent.
Comme la conception très modulaire du système de rendu permet facilement de changer de moteur graphique à l'envie, nous avons expérimenté la création d'un moteur 3D pour pouvoir peser le pour et le contre. Ce fut, étonemment, beaucoup plus facile et agréable que prévu.

Chargement des modèles

Après avoir édité quelques modèles 3D grace au logiciel Blender[3], se pose le premier problème : comment les charger dans le moteur 3D.

Après quelques tentatives infructueuses avec différents formats, le format `3DS`[4] est adopté : la librairie libre `lib3ds`[5] arrive à extraire correctement les fichiers générés par le script d'export disponible dans Blender.

2. Artefact consistant à percevoir les polygones qui forment un modèle 3D lors du rendu
3. http ://www.blender.org/
4. Format binaire du logiciel propriétaire `3DSMax`
5. `http://www.lib3ds.org/`

Cette librairie, bien que très complète, ne gère par contre que le chargement : elle est capable de lire le fichier binaire, d'en extraire à peu près tout ce qu'il contient pour placer les informations dans ses stuctures de données.

Il est nécessaire, à partir de là, de mettre en place un mécanisme pour réorganiser ces données de façon à être exploitables par le moteur de rendu 3D, et surtout, de faire le rendu effectif avec OpenGL.

La classe `Model_3ds` créée dans ce but, n'extrait que 3 informations du modèle : les vertices de chaque triangle du modèle, la couleur du `material` qui leur est associée, et le vecteur normal de chacune de ces faces.

Afin de ne pas subir de polygon aliasing ou d'effet *3D moche*, aucun éclairage n'est utilisé pour le rendu. Ansi, chacune des faces du modèle est rendu intégralement dans la couleur qui lui est associée, rendant chaque modèle chaque modèle plus ressemblant à un dessin 2D à la main qu'un modèle 3D fait de polygones.

Pour accentuer cet air “cartoon”, on utilise en plus la technique désormais célèbre du *cel-shading*.

Cel-shading

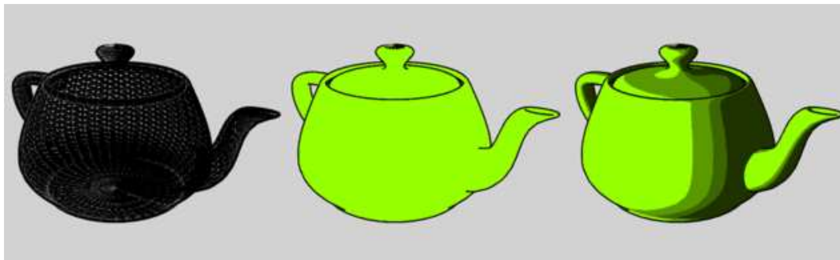


Figure 2.3 – Rendu en cel-shading

Comme représenté sur la figure 2.3, le cel-shading est décomposable en 3 étapes :

- Rendu du contours en dessinant tous les triangles dont les normales sont orientées dans la direction opposée à celle de la caméra, en mode fil de fer, et dans la couleur du contour voulue (utilisation de la technique du *backface culling*)
- Rendu de tous les triangles faisant face à la caméra (*frontface culling*) dans la couleur du matériau de base
- Application d'une texture à 1 dimension servant d'index pour calculer la valeur d'ombrage avec le produit scalaire du vecteur normal au vertex en cours, et du vecteur d'éclairage (normalisés).

Le seul problème rencontré lors de l'implémentation de cette technique vient des données fournies par `lib3ds`. En effet, le calcul de la valeur d'ombrage de chaque vertex demande le vecteur normal à ce vertex, mais `lib3ds` ne fournit qu'une normale par face, donc 3 vertices. Rendu en l'état, cela produit un important aliasing, qui doit être réduit en recalculant les normales de chaque vertex en effectuant la moyenne des normales de toutes les faces voisines.

Optimisations

La première version du moteur de rendu utilisant le Cel-shading s'est révelée excessivement gourmande en CPU. Un profilage à plat en étudiant les graphes d'appels donnés par `Gprof` révèle que la fonction calculant les coordonnées des vertices de `Model_3ds` prend 90% du temps CPU. Son implémentation, naïve, effectue successivement les 3 rotations dans l'espace tridimensionnel requises pour les animations, mais en utilisant des calculs trigonométriques passant par le CPU. C'est inutile, puisqu'OpenGL permet d'appliquer ces transformations directement en les calculant sur le GPU avec des matrices de projection. En passant tous ces calculs sur le GPU grace à l'utilisation intelligente de l'instruction `glRotatef()`, la consommation en temps CPU coté client a été divisée par 4.

Décors

L'aire de jeu, la *map*, est représentée comme décrite dans le chapitre précédent comme une liste de polygones. Lors du rendu, cela rend la zone de jeu un peu vide, puisque ces polygones sont actuellement non texturés. Afin de remplir l'espace, un système de décors a été ajouté. Ces décors sont directement inclus depuis le descripteur de la map, et consistent en un nom, une position dans l'espace, et un polygone pour les collisions. Le moteur 3D, au nom du décor, va associer une texture qu'il récupère via les gestionnaire de représentation. Cette texture est plaqué directement sur un *Quad*, face caméra, de manière à épouser les formes du polygone qu'il représente dans le moteur de collision et donner l'illusion qu'il s'agit d'un objet 3D de même nature que les personnages. Cette astuce est possible grace à la vue orthogonale : pas de perspective, une simple texture représentera à l'écran le même volume d'où qu'on la regarde. Cette technique permet d'afficher des décors complexes pour un total de 1 polygone et 4 vertices, tout en s'intégrant parfaitement dans l'environnement 3D, ce qui est extremement efficace.

Dernières optimisations

Il reste objectivement peu de choses à optimiser pour que le moteur 3D soit parfaitement efficace. La dernière petite touche d'optimisation appliquée à cette partie du client est une technique de *pruning*. Tous les objets et polygones affichés dans le moteur de rendu sont nécessairement triés selon leur ordonnée. En effet, pour pouvoir obtenir un *alpha-blending*[6] correct, il faut dessiner les objets selon leur éloignement de la caméra, à cause des limitations du tampon de profondeur fourni par OpenGL qui ne conserve que peu d'informations sur ce qui a été précédemment dessiné. Ainsi, tous les objets sont déjà triés. Grace à cette propriété, on peut faire sans aucun coût supplémentaire une recherche dichotomique sur les objets à afficher, pour trouver en temps logarithmique quel est le premier objet sur l'axe des ordonnées à apparaitre dans le champ de la caméra, itérer jusqu'au dernier et arrêter là la boucle.

2.2 Efficacité du moteur physique

Le moteur physique a été l'une des toutes premières fonctionnalités du projet. Les algorithmes les plus connus pour les problèmes de collisions entre polygones s'appliquant à des polygones convexes dont l'intersection est initialement nulle, aucun algorithme existant n'a été initialement trouvé répondant directement au problème : l'ère de jeu est englobée par un grand polygone concave dans lequel évoluent les polygones correspondant aux objets du jeu. Pour appliquer les théorèmes classiques de séparation des convexes à ce cas particulier, il faudrait préalablement commencer par diviser les polygones concaves en polygones convexes, augmentant la complexité intellectuelle et informatique du problème.

L'implémentation de base du moteur de collisions a donc été faite le plus naïvement du monde : tester chaque arête une à une, pour une complexité $O(n^2)$. Après tout, comme l'a dit Donald Knuth :

"We should forget about small efficiencies, say about 97% of the time : premature optimization is the root of all evil."

Puis, lorsqu'il est venu le temps de faire du profilage de code, après être passé par les optimisations du moteur graphique, on est revenus sur ce moteur physique aux calculs sub-optimaux.

6. Consiste à mélanger deux pixels selon leur transparence

Puisque la technique du pruning a été appliquée avec succès sur le moteur graphique, elle peut être appliquée au moteur physique : tous les polygones statiques peuvent être triés selon l'axe des ordonnées, lors de leur chargement. Ensuite, quand il faudra tester une à une les arêtes, on testera chacun des objets mobiles contre chacun des objets immobiles, permettant ainsi d'appliquer la même dichotomie que précédemment. On ne s'y méprend pas : la complexité en temps reste bien $O(n^2)$ dans le pire des cas. Ou plus précisément $O(k * n^2)$. Si ni n ni la classe de complexité n'a changé, la constante k, elle a été très fortement réduite.

Avec l'algorithme présenté ci dessus, le profilage révèle que la détection des collisions entre polygones occupe entre 5 et 10% du temps CPU, pour un total d'environ 300 vertices dans le système physique. Contre toute attente, l'algorithme naïf légèrement amélioré se révèle donc suffisant, compte tenu de ce qui est attendu de lui. Il ne sera donc en l'état pas amélioré, même si la possibilité de le faire plus tard en utilisant des algorithmes de partitionnement de l'espace est laissée.

2.3 Synchronisation

La synchronisation de l'état du jeu entre le serveur et les clients est sans doute la partie la plus délicate de tout le projet.

C'était évidemment à attendre compte tenu de la définition du jeu et de la nature du moteur physique, très prompt à introduire des imprécisions à cause des coefficients élastiques et des déplacements avec friction.

2.3.1 Approche Novice

La première définition de l'approche de synchronisation consistait à avoir un moteur de jeu de chaque côté du réseau. Un sur le client, un sur le serveur. Évidemment, le moteur de jeu du serveur avait toujours raison, cependant, beaucoup de liberté était laissée au moteur coté client, tout en exigeant de lui de calculer rigoureusement les mêmes choses que le moteur coté serveur, et en lui demandant évidemment de conserver une précision maximale.

Pour cela, les deux entités ont été alignées de manière à avoir exactement le même *tick rate* : c'est à dire qu'une itération sur le moteur côté serveur prend exactement le même temps qu'une itération sur le moteur côté client. Sur ce même modèle, périodiquement, selon un délai paramétrable, toutes les positions des clients étaient communiquées de façon à corriger d'éventuelles déviations (Fig. 2.4).

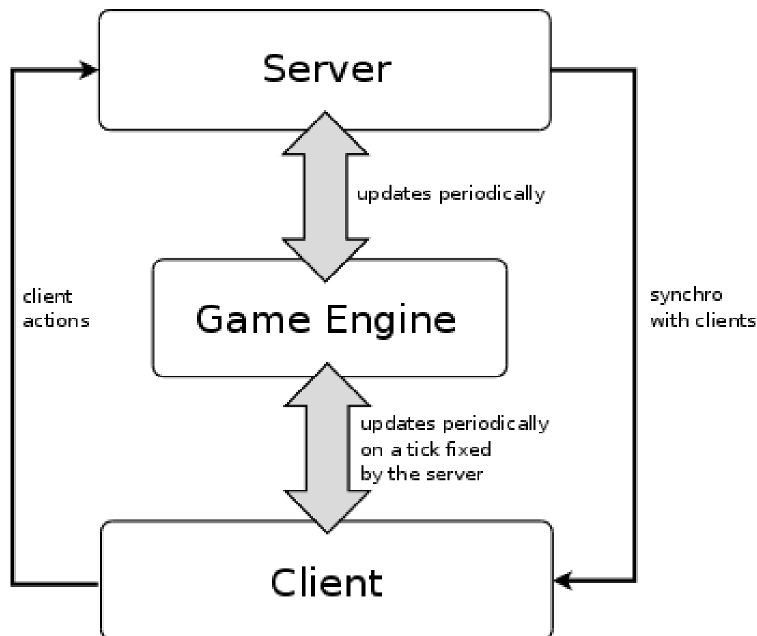


Figure 2.4 – Synchro client / serveur, première version

2.3.2 Approche Hacker

Ce modèle a fonctionné à peu près correctement jusqu'à ce que les entités soumises aux collisions élastiques et aux déplacements avec frottements se mettent à être controlées par le joueur. Assez souvent, des imprécisions apparaissent, et le résultat immédiat est ce fameux effet de *warp* qu'on retrouve à différents degrés dans énormément de jeux en réseau : mis à jour par le serveur, un client saute brusquement d'une position à une autre : il y a eu phénomène de *désynchronisation* suivi d'un recalage brutal et très déplaisant pour le joueur.

Ce phénomène est dû à une erreur de conception simple : la mise à jour des positions, ou resynchronisation, est faite à un moment arbitraire, ou déterminé par un timer : mise à jour toutes les 2 secondes. Cela ne tient pas, puisqu'il y aura forcément des cas où $2s - \epsilon$ vont s'écouler avant la resynchronisation.

Une longue observation du phénomène permet de déterminer précisément quand a lieu la désynchronisation : lors des collisions. La solution est alors évident ; plutot que de mettre à jour les positions à un moment arbitraire, les positions sont systématiquement mises à jour lors de chaque collision. Cela permet d'une part de recaler immédiatement une imprécision due à une

divergence d'angle entre le moteur physique local et le moteur distant, mais également de corriger une imprécision accumulée éventuellement plus tôt à un moment où l'utilisateur risque le moins de le remarquer : une collision entraine un changement brusque de direction et de vitesse. Dans la majorité des cas, si on en profite pour recaler la position du joueur, il n'y verra rien.

2.3.3 Approche Master

La solution précédente n'a malheureusement pas tout résolu : il restait encore cette désynchronisation marginale, un cas dégénéré qui nous a semblé indétectable, ou tout simplement dû au hasard, et que nous avons surnommé le *teleport*, synonyme de désenchantement et désolation.

Non seulement le *teleport* semblait se produire dans des conditions aléatoires, mais en plus il n'occasionait pas des petites resynchronisations telles qu'on avait pu les observer lors de la précédente phase de synchronisation, mais véritable des déplacements instantanés d'un bout à l'autre de l'écran.

Ce bug a probablement persisté plusieurs semaines avant que la cause en soit détectée. Pour résumer, il est dû à une erreur au tout début de la phase de synchronisation : s'il est très utile d'avoir un moteur de jeu local afin de faire de la prédiction et ne pas devoir renvoyer en permanence les nouvelles positions, il est absolument nécessaire que le moteur de jeu local n'ait pas les mêmes propriétés ni rôles que le moteur de référence distant.

La resynchronisation, en effet, se fait lorsque le moteur de jeu du serveur détecte une collision. Dans des cas très rares, cependant, le moteur de jeu local détectait une collision, mais pas le moteur de jeu distant. Résultat, pas de synchronisation, car aucune collision du point de vue du serveur. Ainsi, la copie locale repartait dans une nouvelle direction alors que le moteur de référence faisait toujours évoluer l'objet dans la même direction qu'avant. Le client reste donc désynchronisé jusqu'à ce que le serveur lui trouve une nouvelle et véritable collision, auquel cas le client est brusquement recalé, souvent très loin de là où il croyait être.

La solution à ce problème est de mieux définir les rôles des moteurs distants et locaux. Le moteur distant, de référence, garde le sien. Le moteur local, en revanche, n'effecture désormais plus aucun test de collision et ne peut plus, de lui même, décider de modifier son état courant. C'est le moteur distant, via un appel réseau, qui devra systématiquement lui communiquer son nouveau comportement.

Évidemment, même après cela, les problèmes de désynchronisation occasionels susbistent. Mais, beaucoup moins handicapants, ils sont alors dûs à de petites erreurs de programmation, ou des congestions temporaires du réseau,plus à une réelle lacune de conception.

2.4 Validation de la couche réseau avec PUNIT

`Punit` est le nom donné au système de test conçu *ad-hoc* pour le projet. il permet de tester de façon unitaire la validité et les propriétés de non-regréssion de la couche réseau.

Le but de PUNIT ainsi est de vérifier par une ou plusieurs batteries de tests si les communications via le réseau entre le serveur et les clients sont cohérentes et ne font pas échouer de manière anormale l'un ou l'autre bout de la connexion.

Cettre structure de test permet, entre autres :
- D'exécuter plusieurs tests, et récupérer leur résultat.
- D'associer un envoi de données à une ou plusieurs réceptions.
- De charger plusieurs jeux de tests via plusieurs fichier
- D'accepter des modificateurs spéciaux (+, *, ?, !, W) afin de déterminer si une requête :
  - Doit être exécutée un nombre de fois aléatoire au maximum ou au minimum une fois.
  - Ne doit pas être exécutée
  - Provoque un avertissement lors d'une exécution (requête/fonctionnalité devenue obsolète)
- De reconnaître une requête via des expressions régulières avancées
- De générer un rapport complet et formaté du résultat de l'exécution

Ce système de tests unitaires de l'implémentation de la couche de communication de notre réseau a été implémenté en PHP [7].

En effet, vu la complexité de mise en place d'un framework de tests unitaires pour le réseau, il était assez évident que le C/C++ et même le Java ne pouvaient subvenir à nos besoins : dans le temps imparti, il aurait été impossible de mettre en oeuvre un tel framework de test en utilisant l'un de ces langages, tant il leur manque la flexibilité d'un langage de script.

L'avantage de PHP, outre sa flexibilité, est le fait qu'il a énormément de fonctionnalités natives. L'ajout d'une simple librairie permet d'y inclure le support des Sockets UNIX, et il est très facilement déployable : sur n'importe quel serveur GNU/Linux possède un serveur PHP. De plus, le fait que PHP soit à la base un processeur hypertexte en fait l'outil idéal pour deux raison :
- Excellent moteur d'expréssions régulières, puisque l'une des fonctions principales du langage est de transformer ou parser du texte
- Génération presque instantanée de rapports formatés en HTML et servis sur le Web, puisque PHP est conçu pour générer des pages Web en HTML

7. PHP Hypertext Processor, `http://php.net/`

Les inconvénients d'utiliser PHP pour ce framework sont peu nombreux. Le seul défaut théorique de PHP pourrait être ses mauvaises performances inhérente à sa nature de langage non compilé, mais le fait de ne l'utiliser que pour des appels réseau, qu'il fait au travers de librairies compilées, rend la charge du script quasi-nulle.

La programmation de ce testeur en PHP a donc été très rapide, pour un résultat extrêmement satisfaisant. Il serait bien sur pas adapté pour un quelconque autre protocole que le notre, mais c'est aussi l'avantage de PHP que d'être modifiable très facilement, rapidement, et par à peu près n'importe qui.

Pour utiliser `Punit`, il suffit d'inclure le fichier `tester.php` présent dans le dossier `conf/data/scripts/tester/` de l'abre SVN du projet (qui utilise les autres fichier php du dossier), et de lancer `Punit` après avoir enregistré les fichiers de règles. Chaque fichier de règle peut être exécuté sur un serveur différent du précédent, cela permettrait par exemple de tester plusieurs versions du serveur de jeu, sur plusieurs machine distantes, ayant des charges réseau et CPU hétérogènes. On peut aussi, simuler une latence plus forte, en modifiant une simple variable. Elle n'est cependant pas réglable pour chaque requête interne aux règles, mais cela pourrait se faire simplement.

VAR NAME	DESCRIPTION	DEFAULT
FILE	un fichier de règles	None
HOST	Le nom d'hôte ou se trouve un serveur de jeu	"localhost"
PORT	Le port d'écoute de ce dernier	1024
SLEEP	Le temps de latence du serveur distant	10 (msec)
VERBOSE	Si le résultat doit être affiché	false

```
<?php
register_rule(FILE, HOST, PORT, SLEEP, VERBOSE);
register_rule(FILE2, HOST, PORT, 5000, true);
PUNIT();
?>
```

Figure 2.5 – Exemple d'utilisation de Punit

```
[bind]
1,1,START          => 1,1,START
1,10,PLIST         => *1,10,PLAY [0-9]+ [A-Za-z][A-Za-
                      z0-9]+&&1,11,^[A-Z]{3} [A-Z]{5}\$
1,11,PUNIT         => 1,12,JOINED && 1,3,TICK [0-9]+ &&
                      1,10,PLAY && 2,11,.*

// Can restart, does nothing bad
1,1,START          => 1,1,START

// Can reask plist
1,10,PLIST         => *1,10,PLAY && 1,11,END PLIST

// Cannot rejoin
1,11,PUNIT         => 1,4,already joined

// Changing nickname
1,12,PUNIT         => 1,4,already taken
1,12,PU NIT        => 1,4,Invalid nickname: PU
1,12,P_UNIT        => 1,4,PUNIT is now known as P_UNIT &&
                      1,4,^[0-9]+ P_UNIT
1,12,AXEL          => 1,5,P_UNIT is now known as AXEL &&
                      1,4,^[0-9]+ AXEL
1,12,PUNIT         => 1,5, is now known as && 1,4,.*

[include]
./rules/unbound.rules
```

Figure 2.6 – Exemple de règles Punit (rlogin.rules)

rules/rlogin.rules	Test cases : 19
Test succeed	Ok[18], Misc[1], Warns[0], Errors[0]

rules/rcontrol.rules	Test cases : 7
Test succeed	Ok[6], Misc[1], Warns[0], Errors[0]

rules/radmin.rules	Test cases : 34
Test succeed	Ok[32], Misc[2], Warns[0], Errors[0]

rules/rchat.rules	Test cases : 32
Test succeed	Ok[31], Misc[1], Warns[0], Errors[0]

rules/warning.rules	Test cases : 2
Test passed	Ok[0], Misc[1], Warns[1], Errors[0]
Expected	Result
[W1,1,WARNSTART]	[1,1,START]

rules/error.rules	Test cases : 2
Test failed	Ok[0], Misc[1], Warns[0], Errors[1]
Expected	Result
[1,1,WARNSTART]	[1,1,START]

Figure 2.7 – Rapport d'exécution avec Punit

Chapitre 3

Résultats

Dans ce dernier chapitre sont présentées les fonctionnalités concrètes et visibles qui ont été implémentées dans au cours de ce TER.

3.1 Éditeur

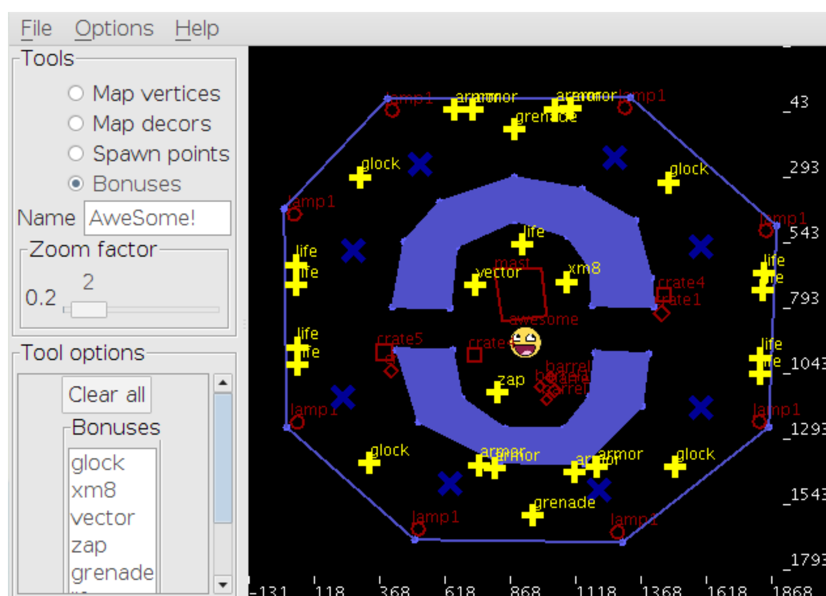


FIGURE 3.1 – Éditeur de niveau

Afin de faciliter l'édition des différentes zones de jeu, un éditeur de niveau a été développé en Java avec une simple interface Swing. Java a été utilisé

pour la rapidité developpement, sur une application où les performances sont à peu près sans objet.

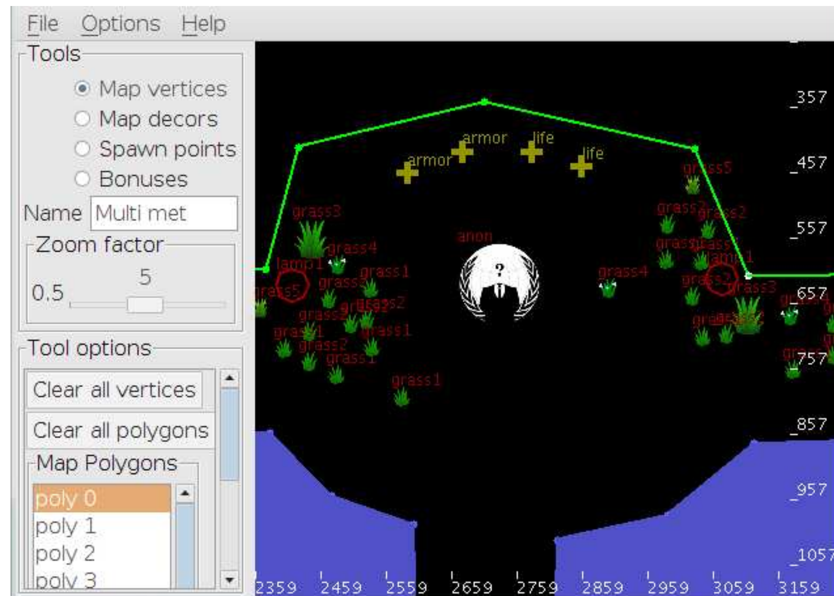


Figure 3.2 – Éditeur de niveau

Cet éditeur de niveau permet de :
- Lire un fichier .map et charger tous ses éléments
- Afficher les bonus, points d'apparition, polygones, décors
- Ajouter, enlever, déplacer à la souris des bonus, points d'apparition, polygones et décors
- Zoomer la carte et translater le point de vue
- Sauvegarder toutes les données éditées au bon format

3.2 Client

3.2.1 Gui

L'interface graphique utilisateur du client a été developpée en utilisant FLTK [1]. Les avantages de FLTK par rapport à un tookit plus connu comme wxWidgets [2], GTK [3] ou Qt [4] sont, non exhaustivement :

1. the Fast Light ToolKit `http://www.fltk.org/`
2. http ://www.wxwidgets.org/
3. http ://www.gtk.org/
4. http ://www.qtsoftware.com/products/

Figure 3.3 – Listing des serveurs obtenus du méta-server

- Très faible empreinte mémoire : le Widget de base fait 60 octets
- Conçu pour être linké statiquement, et autorisé par la licence
- Support OpenGL
- Grande maturité, projet plus vieux que GTK et Qt.

À coté de ça, les limitations se situent du coté de l'aspect général, qui s'approche plus de `NeXT` que de `OSX`, et au niveau du positionnement des widgets, qui se fait obligatoirement de façon absolue, en pixels.

Ces défauts mineurs ne compromettent en rien l'utilité qui est fait de ce toolkit pour le Gui du projet, qui a les fonctionnalités suivantes :

- Choix du serveur de jeu parmi le listing proposé par le méta-serveur (Fig. 3.3)
- Configuration de son personnage pendant le jeu : skin, couleurs, avec zone de prévisualisation OpenGL (Fig. 3.5)
- Configuration des contrôles clavier
- Réglages des options système : résolution, shading, particules, etc.

3.2.2 Jeu

À l'heure actuelle, le jeu permet un seul mode de jeu. Un système d'équipes a été implémenté coté serveur, mais pas encore coté client. Dans le mode de jeu par défaut, chaque joueur peut utiliser j'usqu'à 6 armes en ramassant

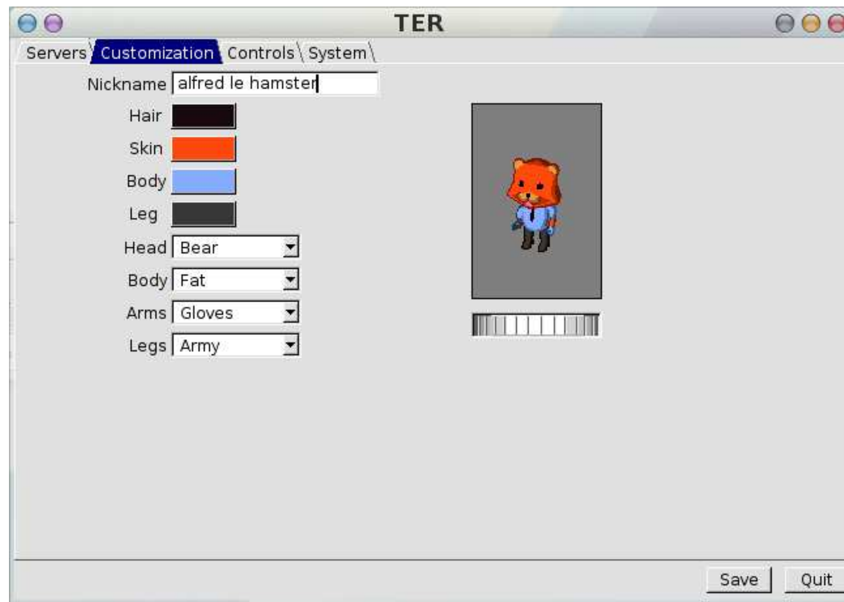


Figure 3.4 – Personnalisation du personnage

les bonus adéquats, qui réapparaissent au bout d'un certain temps, dont une pelle qui projette l'ennemi, une arme à rayon pouvant rebondir sur le décors, ou encore un lance grenade dont le projectile explose après 5 rebonds, 3 secondes ou l'impact avec un personnage, provoquant des dégats à effet de zone et un souffle poussant tout ce qui se trouve à sa portée. Enfin, les joueurs disposent d'une action spéciale, la téléportation : en gardant le clic droit appuyé, ils projettent une sphere qui peut les téléporter là où ils la dirigent, jusqu'à ce qu'ils relâchent la pression, ou que la sphère rencontre un obstacle, décors ou personnage, qu'elle projette à grande vitesse.

Les captures d'écran ci dessus ne rendent pas hommage au jeu, qui prend son intérêt joué à plusieurs, avec force effusions de particules et sous les explosions.

Cette page décrit comment construire le projet à partir des sources : `http://m1.raide.fr/ter/?p=download`

Figure 3.5 – Mode spectacteur

Figure 3.6 – En pleine partie

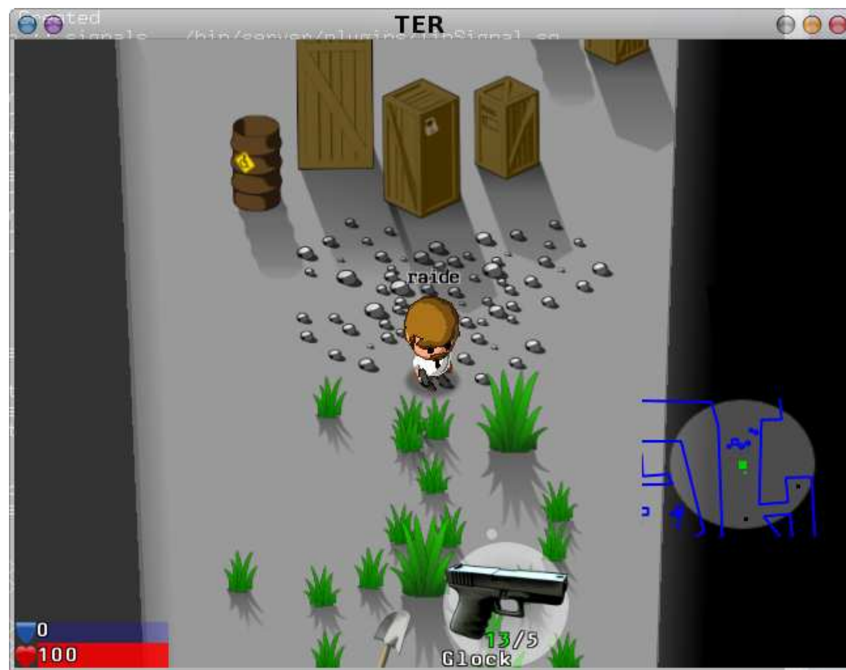


Figure 3.7 – En pleine partie

Conclusion

Au début de ce TER, nous nous étions fixés certains objectifs. La plupart ont été remplis, certains n'ont pas pu l'être à temps :

- Moteurs de rendu en tant que plugin : pour pouvoir en faire la démonstration et prouver son utilité, il faudrait pouvoir disposer d'au moins deux moteurs graphiques de bonne qualité, et assez différents. L'implémentation d'un seule étant déjà complexe, cette idée a été abandonnée
- Portage MS-Windows : annoncé pour la fin du TER, cette échéance n'aura pas pu être respectée. Ignorant tout du developpement sous MS-Windows, nous avons préféré reporter la livraison de binaires pour cette plateforme à quand nous aurions le temps de les faire ; le portage du plugin manager, par exemple, pourra se réveler difficile et demander une bonne connaissance de la plateforme.

Certains aspects, en revanche, ont été développés au delà de nos espérances initiales, avec le Meta Server et son plugin HTTP servant une liste en Xml, l'éditeur de niveau, mais encore le moteur 3D ou le framework de test en PHP.

Globalement, ce TER nous a passionés, parce que nous avons fait ce que nous voulions faire, et que nous avons couvert beaucoup de domaines de l'informatique en un seul projet : conception objet poussée, optimisation de code, protocoles réseau, algorithmique géométrique, tests, génération de Xml, graphisme 3D ou encore problématiques de concurrence.

Ce projet, nous comptons évidemment le continuer et l'enrichir, car nous avons la sensation qu'il reste beaucoup à faire. Propulsé gratuitement et sous licence `GPL`[5] ou équivalente dès que tous nos objectifs seront atteints et qu'un portage MS-Windows sera disponible, nous espérons que ce projet fera un peu parler de lui.

5. GNU General Public Licence `http://www.gnu.org/licenses/gpl.html`

Annexe A

Protocole

The general format for the protocol is :

```
[Range] [idreq] [req] '\0'.
```

Where :

```
Range = the general range of the req
idreq = the uid of the req into the general range
req   = the req.
\0    = the NULL terminated byte
```

Here is the list of implemented methods (always parsed by an entrance req) :

A.1 Server side

[Range : 00]

<N/A>

[Range : 01] Control range

<00>	TERM	The client has logged out
<01>	START	Facultative ack from the client.
<02>	PONG	A pong from the client has been received
<10>	PLIST	An exhaustive Player list has been requested by the client
<11>	JOIN nick	The client asks to join the server with the nickname 'nick'
<12>	NICK nick	The client asked for a nick change
<13>	GETMAP	Asked the current map name
<14>	GETSPAWNS	Asked the player's spawn points
<15>	SPAWN	
<20>	SKIN	The client announced his skin
<21>	SKINCOLOR	The client announced his custom colors

[Range : 02] Game range

<01>	ACT	An action occured
<02>	MOVE x y	A move has been asked <x & y>
<03>	TEAM n	A change team occured from the player
<04>	ORIENT a	The player want to change his orientation to angle 'a'
<05>	STOP	Stops a current movement
<06>	SHOOT	The player fired
<07>	CHWEAPON	The player changed his weapon
<08>	RLWEAPON	The player reloaded his weapon
<09>	INITTP	Starts teleporting
<10>	ENDTP	Apply teleporting

[Range : 03] Admin range

<01>	AUTH pass	Try to authentificate the current user with the password 'pass'
<02>	NICK n1 n2	Changes the current nickname of the player 'n1' by 'n2'
<03>	KICK nick	Kicks 'nick'
<04>	BAN id/nick	Bans 'nick'
<05>	RBAN id	Unbans the 'id' corresponding to an ip
<06>	BLIST	Asks for the banlist
<07>	MUTE id/nick	Mutes 'nick'
<08>	RMUTE id/nick	Removes a mute to 'nick'
<09>	MODERATE	Mutes all the players
<10>	RMODERATE	Removes the mute on all the players
<11>	CHANGEMODE	Changes the game mode
<12>	BALANCE	Balance teams
<13>	KILLTEAMS	Resets the teams
<20>	CHANGEMAP map	Changes the current map to map

[Range : 04] Chat range

<01>	ALL mess	Sends a message to all players
<02>	ONE nick mess	Sends a private message to 'nick'
<03>	TEAM mess	Sends a team message
<04>	ALLA mess	Sends an action to all players
<05>	ONEA nick mess	Sends a private action to 'nick'
<06>	TEAMA mess	Sends a team action

A.2 Client side

[Range : 00]

<N/A>

[Range : 01] Control range

<00>	TERM	The server has shut down
<01>	START	The client has been accepted by the server.
<02>	PING	The server requested a pong to know the client's status connection
<03>	TICK	The server sent the sync. delay (tick) to the client
<04>	MESS	The server has sent a message to the current client
<05>	WALL	The server sent a message to all clients
<06>	BANNED	The player has been banned from this server
<10>	PLAY id name	Send a logged player named 'name' with ident 'id'
<11>	ENDPLIST	The players sent is finished
<12>	JOIN	The client has joined the server
<13>	LEFT id nick	The player identified by ident 'id' and nick 'nick' has left
<14>	NICK id new	A player with id 'id' changed his nick to 'new'
<15>	MAPNAME map	The current map name of the server
<16>	MAPCONTENT	When downloading a new map
<17>	CHANGEMAP map	The server has set the map to 'map'
<18>	SPAWN id x|y	The player spawned here on the map
<20>	SKIN	Announces another player's skin
<21>	SKINCOLOR	Announces another player's skin color

[Range : 02] Game range

<table>
<tr><td><01></td><td>ACT</td><td>An action occured</td></tr>
<tr><td><02></td><td>MOVE id a</td><td>move occured to angle 'a' from a player</td></tr>
<tr><td><03></td><td>STEP id x|y|a|s</td><td>A step occured <x & y & angle & speed> of all clients</td></tr>
<tr><td><04></td><td>CHTEAM id n</td><td>The player changed team to the team 'n'</td></tr>
<tr><td><05></td><td>ORIENT a</td><td>The player has changed his orientation to angle 'a'</td></tr>
<tr><td><06></td><td>STOP id</td><td>The player stopped to move</td></tr>
<tr><td><07></td><td>POSUPDATE</td><td>Updates the position and orientation</td></tr>
<tr><td><08></td><td>SHOOT id</td><td>The player shot</td></tr>
<tr><td><09></td><td>CHWEAPON</td><td>The player changed its weapon</td></tr>
<tr><td><10></td><td>RLWEAPON</td><td>The player reloaded its weapon</td></tr>
<tr><td><11></td><td>SCORE</td><td>Score update</td></tr>
<tr><td><12></td><td>REBONUS</td><td>A bonus appeared</td></tr>
<tr><td><13></td><td>BONUS</td><td>A bonus has been taken</td></tr>
<tr><td><14></td><td>SHOT</td><td>A player has been shot</td></tr>
<tr><td><15></td><td>DIE</td><td>A player died</td></tr>
<tr><td><16></td><td>LIFE</td><td>Health has been lost</td></tr>
<tr><td><17></td><td>GREN</td><td>A grenade bounced</td></tr>
<tr><td><18></td><td>GRENEXPL</td><td>A grenade exploded</td></tr>
<tr><td><19></td><td>EXPL</td><td>An explosion occurrend</td></tr>
<tr><td><20></td><td>TP</td><td>A player teleported</td></tr>
</table>

[Range : 03] Admin range

<01>	AUTH nick	The user 'nick' got administrator privileges
<02>	NICK id nick	Player with id 'id' has changed his nickname to 'nick'
<03>	KICKED nick	The player 'nick' has been kicked
<04>	KBANNED nick	The player 'nick' has been kick-banned
<05>	UNBANNED ip	The 'ip' bas been unbanned
<06>	BLIST	The banlist begins
<07>	BAN id ip	A ban on 'ip' exists and has 'id' as identifier
<08>	ENDBLIST	End of the banlist
<09>	MUTED nick	The player 'nick' has been muted
<10>	UNMUTED nick	The player 'nick' has been unmuted
<11>	MODERATED	Moderated mode activated
<12>	UNMODERATED	Moderated mode unactivated
<13>	KILLTEAMS	Reset teams

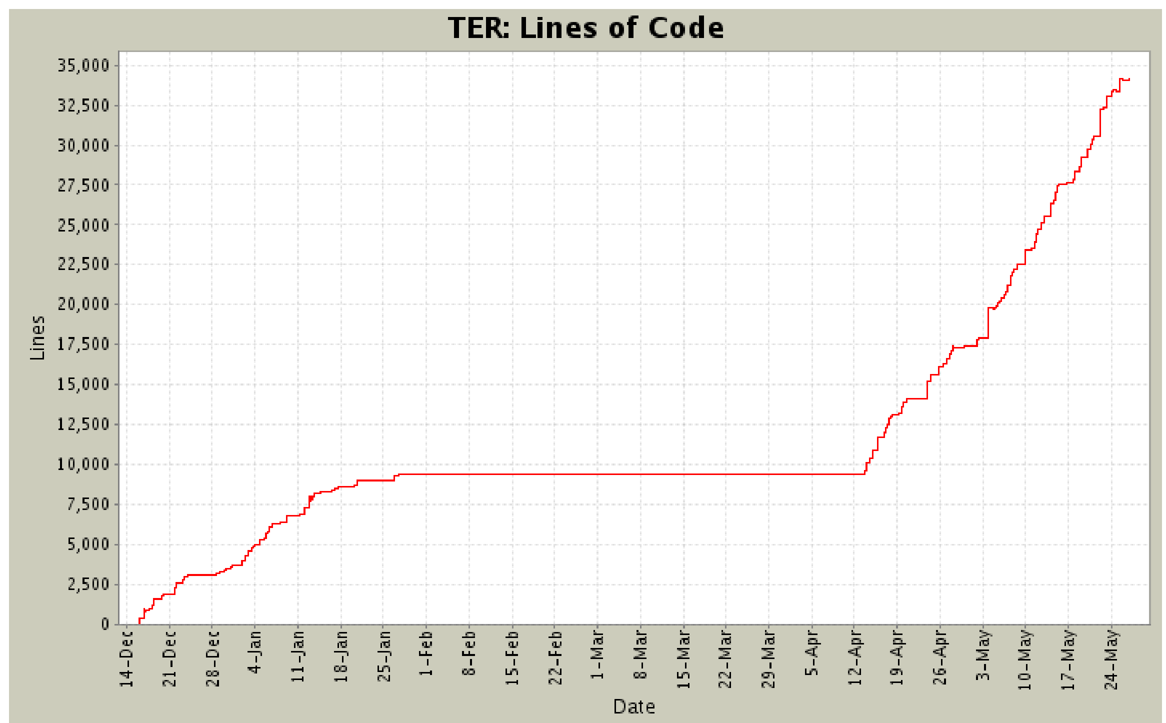
[Range : 04] Chat range

<01>	ALL nick mess	Receives a message to all players from 'nick'
<02>	ONE nick mess	Receives a private message from 'nick'
<03>	TEAM nick mess	Receives a team message from 'nick'
<04>	ALLA nick mess	Receives an action to all players from 'nick'
<05>	ONEA nick mess	Receives a private action from 'nick'
<06>	TEAMA nick mess	Receives a team action from 'nick'

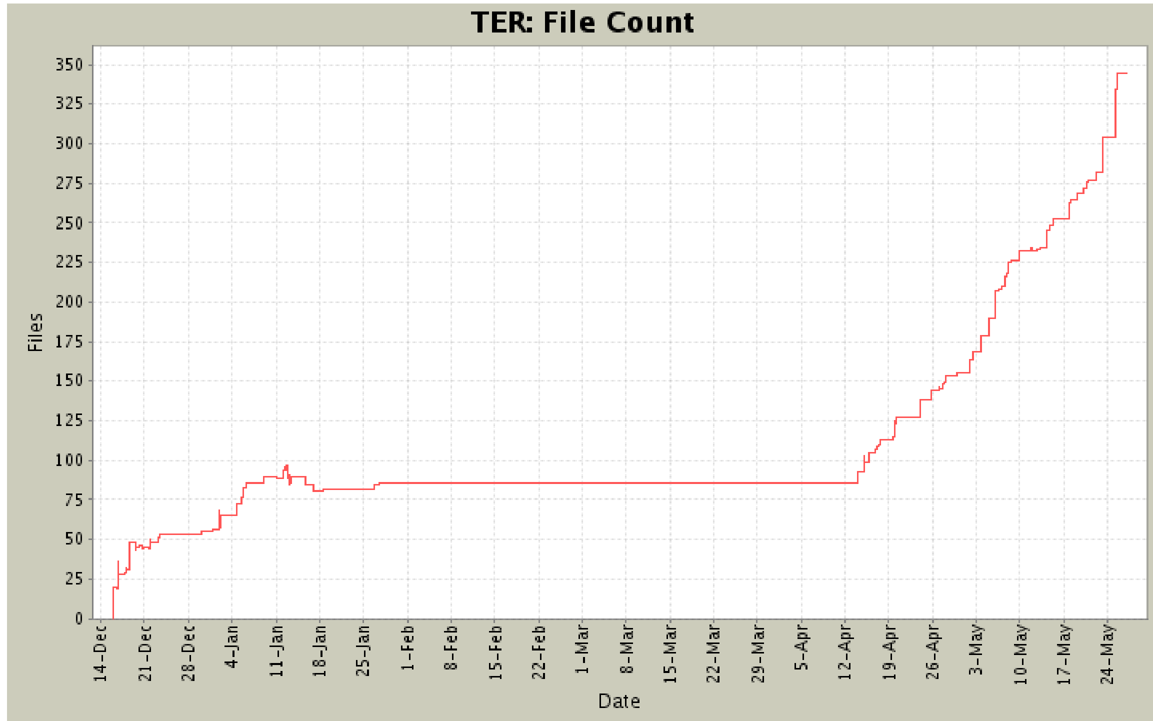
Annexe B

Statistiques

Les données statistiques suivantes ont été générées par StatSVN [1] à partir du dépôt SVN du projet.

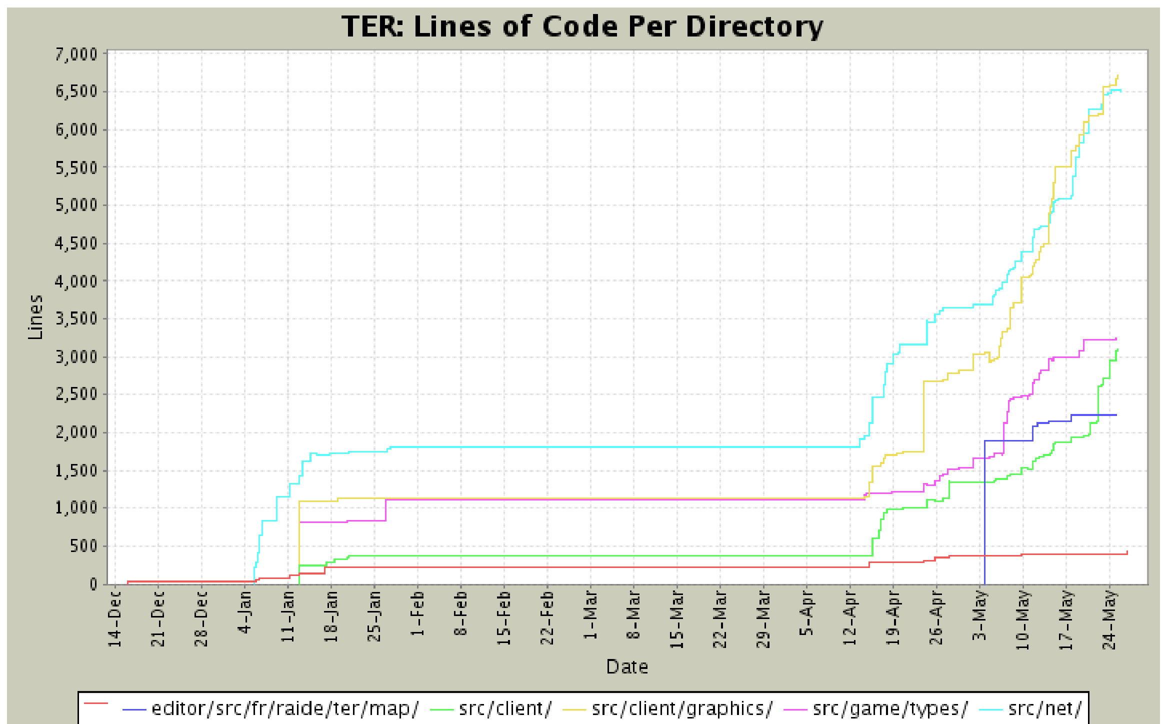


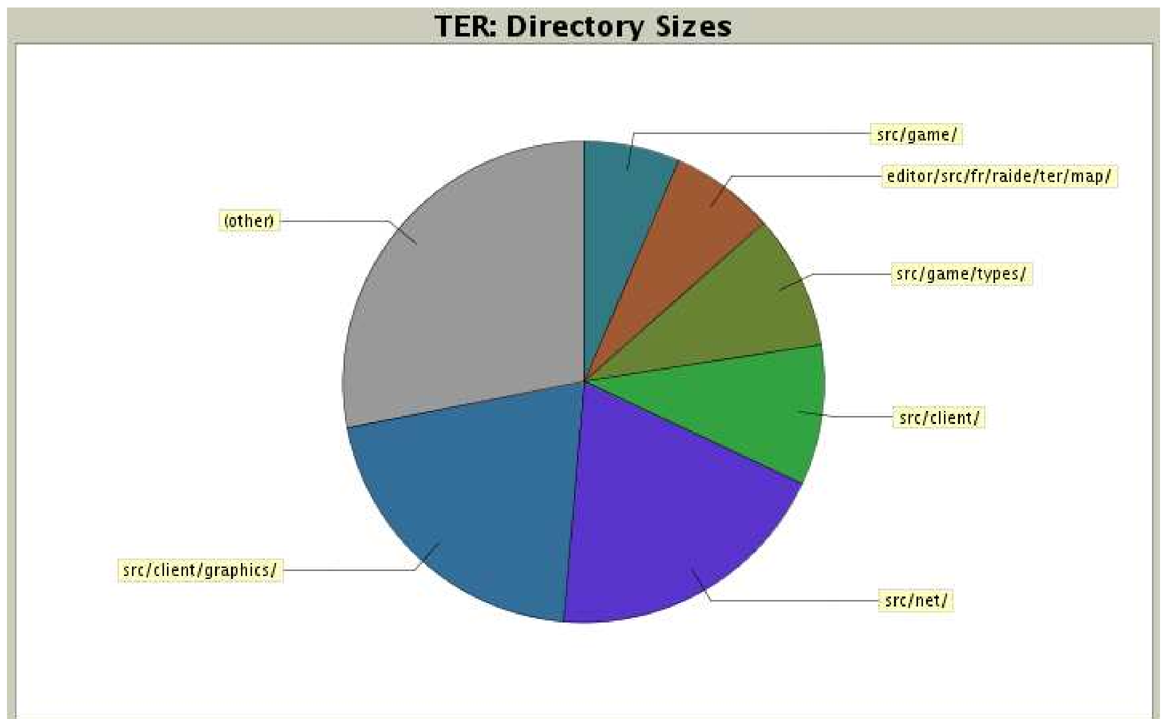
1. http://www.statsvn.org/



Type	Files	LOC	LOC per file
Totals	344 (100.0%)	31978 (100.0%)	92.9
*.cc	64 (18.6%)	18719 (58.5%)	292.4
*.hh	71 (20.6%)	8383 (26.2%)	118.0
*.java	8 (2.3%)	2241 (7.0%)	280.1
*.php	9 (2.6%)	1090 (3.4%)	121.1
*.map	2 (0.6%)	411 (1.3%)	205.5
*.txt	2 (0.6%)	181 (0.6%)	90.5
*.rules	9 (2.6%)	174 (0.5%)	19.3
*.conf	3 (0.9%)	123 (0.4%)	41.0
*.dec	25 (7.3%)	58 (0.2%)	2.3
*.xml	1 (0.3%)	33 (0.1%)	33.0
Others	3 (0.9%)	565 (1.8%)	188.3
Non-Code Files	147 (42.7%)	0 (0.0%)	0.0

Figure B.1 – File types


TER: Lines of Code Per Directory
Lines
7,000
6,500
6,000
5,500
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
14-Dec
21-Dec
28-Dec
4-Jan
11-Jan
18-Jan
25-Jan
1-Feb
8-Feb
15-Feb
22-Feb
1-Mar
8-Mar
15-Mar
22-Mar
29-Mar
5-Apr
12-Apr
19-Apr
26-Apr
3-May
10-May
17-May
24-May
Date
editor/src/fr/raide/ter/map/
src/client/
src/client/graphics/
src/game/types/
src/net/


TER: Directory Sizes
src/game/
editor/src/fr/raide/ter/map/
(other)
src/game/types/
src/client/
src/client/graphics/
src/net/