TBT

Proposition d'architecture - Révision 4238

Olivier Gournet
victor@towbowltactics.com

Pascal <Toweld> Bourut
toweld@free.fr

7 juin 2006

Table des matières

PRÉAMBULE

TowBowlTactics (TBT) [Team, 2006] est une adaptation électronique du jeu de plateau Blood Bowl (BB), édité par Games Workshop (GW). Il existe à ce jour plusieurs adaptations permettant de jouer à ce magnifique jeu. Une des première a été réalisé par GW, mais se fait vieillissante et ne respecte pas trop l'esprit du jeu de plateau. D'autres tentatives ont été effectué, comme avec RudeBowl [Team, 2005], 3dBowl [Soft, 2005] ou pyBloodBowl [Tokiros, 2005], mais ces projets ont soit avorté, soit ne sont plus activement maintenu. La référence actuelle est JavaBB [SkiJunkie, 2006], utilisé par tous les joueurs en , mais souffre de quelques défauts. Le graphisme est assez austère, et les versions actuelles ne sont pas *open source*, ce qui rend plus difficile l'ajout d'extensions. L'objectif[1] à long terme de TBT est de combler ce trou, en proposant une adaptation de Blood Bowl libre, jolie, extensible, en essayant le plus possible de respecter le jeu plateau.

Ce document établit une nouvelle architecture pour TBT en repartant de zéro. Il contient la ligne directrice qu'il a été décidé de suivre suite à de nombreuse discussions sur le forum et quelques sessions IRC, la modélisation du programme ainsi que des précisions sur des parties de code qui le méritent.

Ce document peut être considéré comme le document de référence du projet. Néanmoins, il peut contenir des erreurs, et son contenu peut encore être remis en question si quelqu'un arrive, propose une idée géniale et est prêt à la mettre en oeuvre[2]. Par contre, ce document n'aborde le problème que sous l'angle *technique*. Il n'a pas pour vocation d'être un manuel utilisateur, et fournit très peu de détails ; les graphismes, les effets sonores, l'ambiance de jeu, la réponse à la vie, l'univers et le reste [Wikipedia, 2006]...

Les auteurs tiennent à remercier tuxrouge, jibone, jgi, poituiu, Elpopotam, Cedric, krys, Vanhu, Tupad, trem, darkside, et tous les oubliés, pour toutes les remarques pertinentes et diverses contributions.

[1] Ce que l'on espère, il n'est pas dit que ca marchera :)
[2] Entendez par là, inutile de débarquer, de proposer : ¨Et si on faisait un truc super générique qui gère toutes les jeux de GW¨, et de partir en courant.

GLOBALITUDES

2.1 Direction du projet

L'ancienne version de TBT (la 0.5) est vraiment bien coté utilisateur, un peu moins niveau développement. La structure, beaucoup trop monolithique, liée à un manque de documentation chronique, a pas mal freiné le dev. L'objectif, dans un premier temps, est de d'obtenir la même chose que la 0.5, au niveau du jeu et de l'interface graphique, mais en le faisant un peu mieux[1] pour pouvoir aller plus loin ensuite.

Voici un résumé des fonctionnalités et grandes lignes que nous nous sommes efforcé de garder en tête tout en rédigeant ce papier :

- **Réalisme**. Il faut que quelque chose puisse fonctionner en peu de temps, et soit assez souple pour permettre toute sorte d'améliorations par la suite. Entendez par la qu'il est hors de question de faire une liste exhaustive de toutes les demandes et possibilités, pour se retrouver avec une *TodoList* propre à démoraliser la meilleure volonté.
- **Structuration par modules**. Il devrait être ainsi possible, sans duplication de code, d'avoir une version solo tout-en-un, une architecture client/serveur, la possibilité de remplacer un coach humain par un coach IA.
- **Réseau**. Gros manque de la 0.5, pourtant c'est un élément essentiel. Il est ici prévu dès le départ.
- **Portabilité** Unix/Windows. Les choix initiaux (C++/SDL) nous permettent d'assurer une sortie sur ces deux OS, la suite se verra dans les outils et bibliothèques utilisés.
- **Sécurité**. Un problème des jeux libres est que chacun peut modifier le code pour s'octroyer quelques pouvoirs supplémentaire. On va s'efforcer de minimiser cet effet, notamment lors de matchs joué en ligue (plus de détails dans la partie 7.1).
- **Bonne lisibilité** de cette proposition d'architecture. Heu non, ça c'est rapé :)

Le jeu de règle retenu est le LRB 4.0 (Living RuleBook), disponible en français et en anglais sur le site. Dans le futur, il n'est pas exclu que plusieurs jeux de règles soient supportés, même si aucune méthode pour cela n'a encore été retenue de manière définitive (voir la partie 10).

[1] Et avec de la documentation...

2.2 Licence

Le code est source libre, ca c'est un point acquis. Le svn est librement accessible en lecture[2]. Tout le monde peut soumettre des modifications soit en nous les envoyant, soit en demandant à TuxRouge un accès en écriture sur le svn. Au niveau de la licence la difficulté vient du fait qu'il s'agit de l'adaptation d'un jeu qui n'est pas libre de droits. **GW!** (**GW!**), contactés à l'époque du premier tbt avait authorisé la poursuite du projet à la condition qu'on arrête tout développement s'ils nous le demandent. Pour jouer à TBT "légalement", il faut posséder le jeu de plateau. Nous essayons de reprendre le moins possibles d'éléments propres à GW. C'est pourquoi nous allons refaire les graphismes des figurines et de l'interface. Une race est la propriété de **GW!**, il s'agits des skavens. Pour les règles, c'est plus compliqué et je ne voudrais pas écrire trop de bêtises, donc, à voir ...

2.3 Cas d'utilisation

En dehors des développeurs, il y a deux types de personnes qui peuvent avoir à utiliser TBT. Les joueurs, pour pouvoir y jouer, et les administrateurs système, pour installer et faire tourner des serveurs de jeu, notamment lors de ligues.

Plaçons nous du point de vue admin sys. Ca doit être aussi simple que de récupérer/installer le serveur, modifier un fichier de configuration, et le lancer depuis une console. Pas besoin d'interface graphique. Éventuellement, pouvoir agir sur le serveur une fois lancé.

Plaçons nous du point de vue du joueur. Il vient de télécharger le jeu, de le décompresser/installer, et aimerait y jouer. Pour lui, zéro configuration, il doit juste lancer l'interface graphique, et les quelques options de configuration (résolution de la fenêtre, ...) doivent se faire depuis là.

Ensuite, il doit pouvoir lancer une partie de plusieurs manières différentes :
- En solo, contre une AI, sur une même machine.
- A deux joueurs humain sur une même machine.
- Créer une partie, sur une LAN ou internet, pour qu'un autre joueur humain puisse le rejoindre.
- Rejoindre une partie créée sur une LAN ou internet par un autre joueur humain.
- Rejoindre en tant que spectateur une partie déjà créé et/ou déjà débutée, quelque part sur une LAN ou internet.
- Se connecter à un serveur publique (ou de ligue) existant sur le réseau, et pouvoir y créer/joindre/assister à des matchs.

[2] https://projects.nekeme.net/projects/tbt/wiki/CompilationDeveloppeur

Dans (presque) tous les cas de figure, ce fonctionnement sera appelé *mode réseau*, *mode client/-serveur*, ou encore *network mode* (voir la figure 2.5). Il y aura toujours trois processus (programmes) lancés : un serveur, et deux clients. Même si le joueur a décidé d'héberger une partie, un serveur sera lancé sur sa machine et recevra les connexions. Même si il a décidé de jouer seul sur sa machine contre une AI, un serveur et un autre client seront lancés dans son dos depuis l'interface graphique. Aussi bizarre que cela puisse paraître, ce cas d'utilisation *solo* rentre dans la catégorie *mode réseau*.

Le lecteur attentif n'aura pas raté la faille de ce système. Il y a un cas où ça ne fonctionne pas, quand deux joueurs humain décident de jouer sur la même machine. On ne peut pas lancer deux interfaces graphique sur la même machine. Il faut n'en lancer qu'une seule, et s'arranger pour que cette interface graphique puisse passer d'un joueur à l'autre en fonction du jeu (puisque c'est un jeu tour à tour, on peut se le permettre). Ce mode de fonctionnement spécial est appelé *mode standalone* (voir la figure 2.5), et ce cas sera son unique cas d'utilisation.

2.4 Les outils/bibliothèques utilisés

Ci contre un rapide résumé des divers outils et technologies retenues, et qui seront utilisés. Ces choix sont définitifs. En vrac :

- Le langage retenu pour le coeur du programme est le **C++**. Non négociable. Par contre, il sera possible d'écrire certaines parties *externe* dans d'autres langages.
- L'interface graphique principale utilise **SDL**, et est codé en **C++**. Il sera possible d'écrire d'autres interfaces graphiques plus tard, dans d'autres langages et avec d'autres bibliothèques.
- Le format d'échange standard, pour la description des équipes, du fichier de configuration, ..., est **XML**.
- La bibliothèque permettant de manipuler ces fichier XML sera **xerces-C++** [projects, 2006].
- La partie réseau utilisera les bonnes vieilles sockets, le protocole de communication sera refait à la main, suivant les besoins.
- L'environnement de développement principal est GNU/LINUX, utilisant le compilateur **g++** (>= 3.3). Une compatibilité avec l'environnement de développement **VC++7**, sous Windows, tentera d'être maintenu[3].
- La configuration du projet sous Linux se fait à l'aide des **autotools**.
- Le contrôle des sources se fait à l'aide **subversion**.

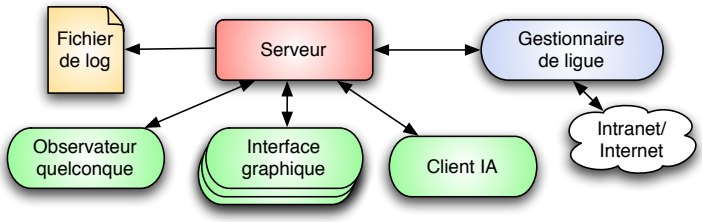


FIG. 2.1 – Schéma global de l'environnement TBT

2.5 La foire aux modules

La version proposée est beaucoup plus modulaire que la précédente. Au lieu d'un seul répertoire source produisant un seul binaire, on a ici un ensemble de composants. Ceux-ci vivent dans des répertoires source différents, sont compilés en bibliothèque statique et assemblés selon les besoins. Nous aurons besoin de :

- **Outils** : Parser xml, Logger, ...
- **Transfert de données** : S'occupe du transfert de messages entre les clients et le serveur. (classes `Cx` et dérivés).
- **Arbitrage** : Composant servant à arbitrer les décisions des coachs, et à lancer les dés. Ce composant existe en un seul exemplaire pour une partie. (classe `SRules`).
- **Règles client** : Contient toutes les informations mises à jour par l'arbitre sur la situation de jeu. Permet de faire le lien entre le coach et l'arbitrage, également permet de filtrer les ordres ¨triviaux¨ des coachs : ceux dont les choix sont déterministes et qui n'ont pas d'incidence sur l'autre coach. (classes `GameClient` et `CRules`).
- **Serveur** : Qui s'occupe de la gestion des différents clients, spectateurs, et pourra interagir avec un gestionnaire de ligue. (classe `Server`).
- **UI (user interface)** : En bout de chaîne, interagit avec un coach. L'UI peut être :
  - **GUI (graphical user interface)** : Interface graphique, s'adressant à un joueur humain.
  - **CLI (command line interface)** : Interface console, s'adresse à un humain un peu geek qui préfère la console, ou utilisable facilement par des scripts.
  - **AI (artifical intelligence)** : Coach géré par un ordinateur.

Faites bien attention à la terminologie utilisée, qui porte souvent à confusion. Référez vous au glossaire pour plus de détailsA.

A noter également qu'en mode de fonctionnement *client/serveur*, il sera possible de lier le serveur à un gestionnaire de ligue, comme décrit dans la partie 3.

[3]Mais dépendra vraiment de la bonne volonté des développeurs.

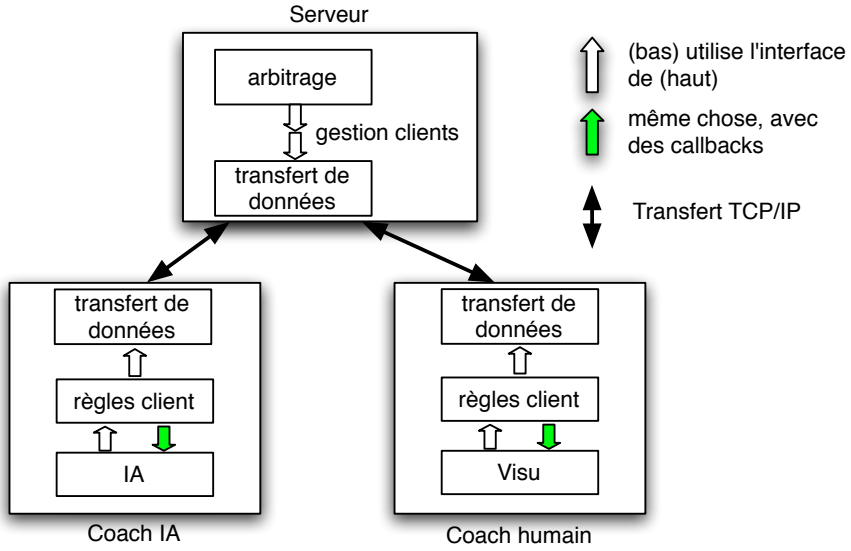


FIG. 2.2 – Emboitement des modules, en configuration *client/serveur*.

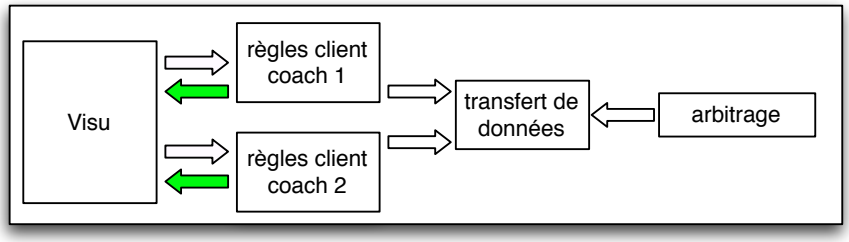


FIG. 2.3 – Emboitement des modules, en configuration *standalone*.

2.6 Fonctionnement général

La figure 2.6 décrit le processus de traitement d'un message, à travers les différents modules. Le protocole mis en jeu lors du parcours de ce message, ainsi que les interfaces des différentes classes sont explicitées plus en détails dans les parties suivantes.

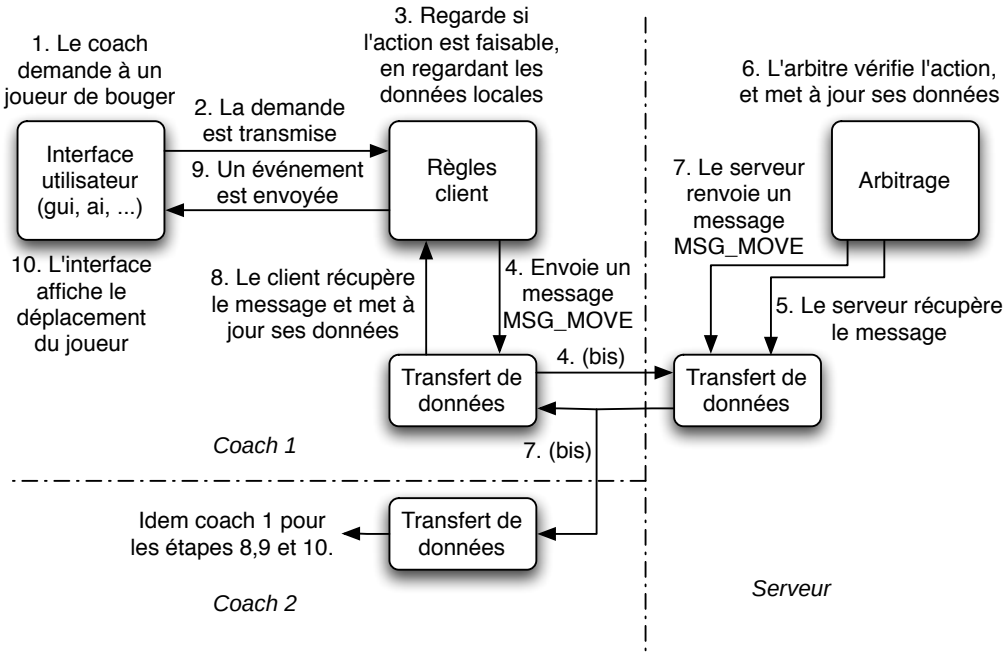


FIG. 2.4 – Détails du processus de traitement d'un message.

GESTIONNAIRE DE LIGUES

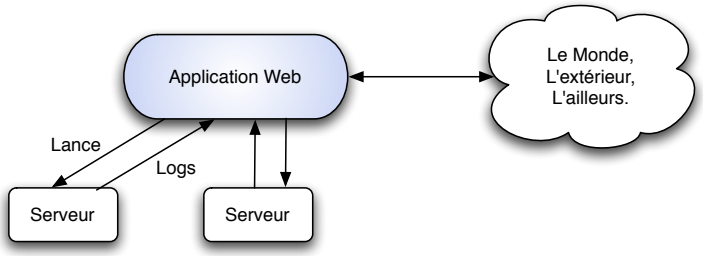


FIG. 3.1 – Gestionnaire de ligue dans son environnement

Pas grand chose à en dire. Ça doit être une application totalement séparée, qui doit fournir les services suivant :

- Générer plusieurs matchs, des poules, ...
- Lancer ou se connecter à des serveurs avec les bons paramètres.
- Lire et analyser les logs du serveur.
- Gérer les équipes entre les matchs.

Idéalement, ça serait un site web, soit développé *from scratch* (php/mysql, rails, ...), ou une adaptation d'un projet déjà existant (par exemple, FGS[1]). Il pourrait être développé totalement en parallèle de TBT, ou encore avant ou après le reste de l'appli[2], et ne dépend que du format de log. Pour le lancement des serveurs, l'appli web peut forker, appeler un autre cgi qui s'occupera d'exécuter un serveur, se connecter à un démon, ... On pourrait même imaginer une architecture distribuée derrière, mais pour l'instant considérons simplement qu'il est capable de lancer un serveur.

Pour l'instant, il n'est proposé aucune modélisation plus poussée de cette partie, qui dépendra majoritairement de la technologie choisie. Le gestionnaire de ligues ne fait actuellement pas du tout parti de nos priorités. D'abord, avoir un jeu qui marche pour jouer une partie. Ensuite, on verra.

[1] http://www.nekeme.net/index.php/FGS
[2] ou pas.

LE MODULE TRANSFERT DE DONNÉES

En réalité, il y a très peu de chose à dire sur ce module. Il doit fournir une interface permettant d'envoyer et de recevoir des paquets. Dans cette partie, on se permettra une petite entorse au plan en prenant de l'avance et en décrivant l'utilisation des paquets ainsi que le protocole de communication, bien que ces deux éléments ne fassent pas à proprement parler partie du module transfert de données.

Ce module est codé en C++ et utilise TCP/IP (d'autres extensions, comme le support http/https sont envisageable), et possède aussi une implémentation *kludge* permettant la transmission de paquets dans la configuration *standalone*.

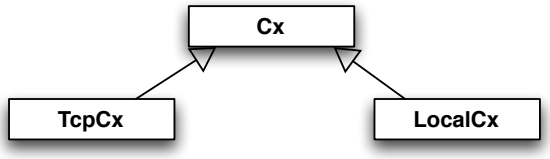


FIG. 4.1 – Hiérarchie du réseau.

4.1 Interface

Ce composant est en haut de la chaîne. Il ne dépend de personne, donc peut fournir une interface propre sans se préoccuper des autres. Voici son interface, en C++[1], correspondant à la hiérarchie de la figure 4.

```
class Cx {
  virtual void send(Packet*);
  virtual Packet* receive();
  virtual void poll();
};
class TcpCx : public Cx {
```

[1] Y aurait-il un meilleur formalisme ?

```
  void connect(host, port);
  void listenAt(port);
};
class DirectCx : public Cx {
  virtual void send(Packet*);
  virtual Packet* receive();
  pktList queue_;
};
```

Listing 4.1 – Interface (simplifiée) des classes composant le réseau.

La classe `DirectCx` propose une implémentation *quick&dirty*, en stockant les paquets dans une liste lors de l'appel de la méthode `send()`, et les restituant à l'appel de la méthode `receive()`. Elle n'accepte que deux instanciations, une pour le serveur et une pour le client, et est protégée contre les accès concurrents de processus léger.

4.2 Base

Les transferts de données se font à l'aide de la classe `Cx`, qui possède les méthodes nécessaires pour envoyer ou recevoir des données. L'objet transféré, à travers la classe `Cx`, est l'objet `Packet`. Il sera transféré tel quel, en mode binaire.

```
struct Packet {
  int token;
  int client_id;
  int data_size;
};
```

Listing 4.2 – Structure d'un paquet

- `token` : Signification du message. Plusieurs tokens possibles sont décrits à la table 4.2.
- `client_id` : L'uid du client duquel le message provient / auquel le message est destiné.
- `data_size` : Taille totale du paquet.

Tel quel, la classe `Packet` est un peu inutilisable. En fait, elle sera héritée pour chaque type de message, chacun rajoutant ses propres attributs. A noter : pour effectuer une conversion simple (passer outre les incompatibilités little/big endian), tous les attributs doivent être des entiers, codés sur 4 octets. Pour faire passer des chaînes de caractères, il faut déclarer un tableau d'entier, et utiliser des méthodes pour faire la conversion.

Pour déclarer un nouveau paquet, il faut utiliser les macros définies dans `common/PacketHandler.hh`. Celle ci permet une gestion plus facile des paquets arrivant dans le moteur, comme décrit dans la section **??**.

4.3 Protocole

Cette partie décrit le fonctionnement en mode *réseau*, le mode *standalone* étant similaire. Au lancement, le serveur écoute sur un port défini à l'avance, et attend 2 connexions de la part de coach actifs (humain ou IA), avant de lancer la partie. Les spectateurs peuvent aussi se connecter durant cette phase, ou à n'importe quel moment de la partie. Exemple de communication lors de la connexion d'un client :

```
Serveur                         Client
                    <-          CX_INIT
CX_ACCEPT           ->
CX_SENDUID          ->
```

CX_INIT est envoyé par le client pour initier une connexion, et contient entre autre le motif de cette demande (spectateur ou acteur), le nom du coach, …. Le serveur pourra alors lui renvoyer CX_ACCEPT, suivit d'un CX_SENDUID contenant l'identifiant unique que le coach s'est vu attribué, ou bien CX_DENY.

Token	Cl.	Srv.	Description
CX_INIT	X		Demande de connexion
CX_ACCEPT		X	Ouverture de connexion acceptée
CX_DENY		X	Ouverture de connexion refusée
MSG_CREATEOBJECT	X	X	Création un objet (cf sec. 6.1.2)
MSG_UPDATEOBJECT	X	X	MAJ d'un objet (cf sec. 6.1.2)
MSG_DELETEOBJECT	X	X	Destruction d'un object (cf 6.1.2)
MSG_NEWTURN		X	Début d'un nouveau tour
MSG_TURNOVER		X	Fin du tour en cours
MSG_ILLEGALPROC	X		Demande d'interruption provenant de l'adversaire
MSG_TIMEOUT		X	Dépassement du temps lors d'une interruption
MSG_CHAT	X	X	Bavardage en ligne
ACT_MOVE	X		Un joueur bouge
ACT_BLITZ	X		Un joueur blitz
ACT_PICKBALL	X		Un joueur récupère une balle

FIG. 4.2 – Liste (non exhaustive) des tokens. Les colonnes `Cl.` et `Srv.` indiquent respectivement que le token peut être envoyé d'un client ou du serveur.

Provenant d'un monde où les gens se voient en vrai, et bien que ce soit un jeu tour à tour, les concepteurs ont introduit la faculté au coach qui glande la possibilité d'interrompre le tour de l'autre, ou de devoir lancer des dés, histoire de ne pas s'endormir. Par exemple, lors de l'oubli du

marqueur de tour, quand le second coach essaie de bloquer un joueur plus fort que lui, lors de certaines cartes spéciales, les sorciers, . . .Ces possibilités devront être reproduites.

Le protocole est ainsi complètement orienté traitement par *flot*, où chaque message provenant d'un client est traité en temps réel puis une réponse apportée, soit positive, soit négative. Il doit *toujours* y avoir une réponse, pour ne pas laisser le client dans l'expectative. On suppose le protocole comme sûr (pas de paquet perdu), ce qui est heureusement le cas avec TCP, ou avec nos *fake methods*.

Un tour est limité à 4 minutes. Le serveur devra s'assurer que cette limite n'est pas franchi, en déclenchant un compte à rebours au début du tour d'un coach. Les seules pauses permise seront pendant l'interruption d'un autre coach. Les temps de calcul du serveur et les temps de transfert réseau ne seront pas pris en compte, parce qu'ils devraient toujours être inférieur à 0.2 secondes pour une action, temps pendant lequel le coach peut réfléchir.

Un exemple de tour du coach 1 pourrait se faire de la manière suivante (attention, ce n'est vraiment qu'indicatif) :

```
Serveur                  Client 1                Client 2
MSG_NEWTURN       ->
                  <-     ACT_MOVE
ACT_MOVE          ->
MSG_UPDATEOBJECT  ->
                  <-     ACT_BLITZ
ACT_BLITZ         ->
                                          <-     MSG_ILLEGALPROC
MSG_ILLEGALPROC   ->
MSG_TIMEOUT       ->
                  <-     MSG_TURNOVER
MSG_TURNOVER      ->
```

Pour commencer, le serveur indique dans un paquet `MSG_NEWTURN` que c'est au coach 1 de jouer. Le coach *souhaite* déplacer un joueur, en fait la demande au serveur, qui accepte en renvoyant le message `ACT_MOVE` suivit d'une mise à jour de l'objet joueur en question. Ensuite, le coach 1 veut effectuer un blitz. Le serveur refuse, et renvoie un message `ACT_BLITZ` indiquant que l'action a été refusée. Ensuite, vu que le marqueur de tour n'a pas été bougé, le coach 2 se lance dans une procédure illégale. Là, comme je ne savais pas quoi mettre, disons que cette procédure timeout. Ensuite, le coach 1, ayant fini de jouer, demande la fin de son tour avec `MSG_TURNOVER`, répété par le serveur.

LE MODULE XML

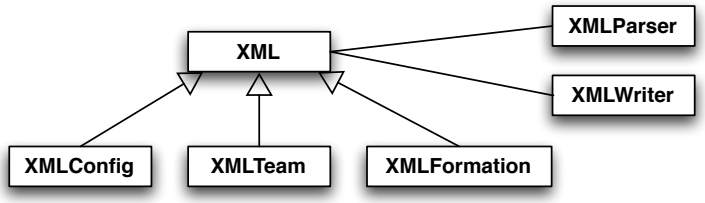


FIG. 5.1 – Modélisation du module `XML`.

Ce module ne mérite sûrement pas une section à lui tout seul, car il n'y a pas grand chose à en dire. Il est là pour encapsuler la bibliothèque de manipulation XML *xerces* afin de fournir une interface très simple aux autres modules, permettant de s'abstraire totalement de la manipulation de noeuds, de fichiers, et de la mémoire dans le reste du programme.

A l'utilisation, c'est plutôt simple. Incluez[1] et instanciez une variable de type `XMLConfig`, `XMLTeam` ou `XMLFormation` selon le type de document que vous voulez utiliser, et utilisez ses méthodes pour récupérer/modifier des attributs. Cet objet peut être vu comme une mini *base de donnée*, il est mis à jour dynamiquement en fonction des modifications. Et la fin, vous pouvez le sauvegarder dans un fichier. Plus d'informations et des exemples de code sont disponibles dans la documentation doxygen.

Ce module est déjà quasiment complètement codé. Il reste néanmoins quelques améliorations, comme une vérification complète de l'utilisation mémoire, ainsi que de l'ajout d'une TDT pour permettre la vérification des documents ainsi que de fournir des valeurs par défaut dans le cas où tout l'utilisateur aurait omis quelques champs. Et, bien entendu, le débugger[2].

[1] Ne faites pas d'include sur `XMLWriter` et `XMLParser`, ils ne sont utilisés qu'en interne
[2] Non... vraiment ? :)

LE MODULE RÈGLES : ARBITRAGE ET RÈGLE CLIENT

6.1 La partie de code commune

6.1.1 Pourquoi donc ?

Les composants *arbitrage* et *règles clients* ont beau être deux entitées complètement distinctes, des similitudes existent, ce qui implique du code dupliqué si on ne fait rien. On vient déjà, dans la partie précédente, de factoriser le code réseau, on va ici tâcher de factoriser les structures de code commune, par une classe de base, `Rule` (cf fig. 6.1.1). De cette partie émerge une bibliothèque, nommons la `libcommon.la`.

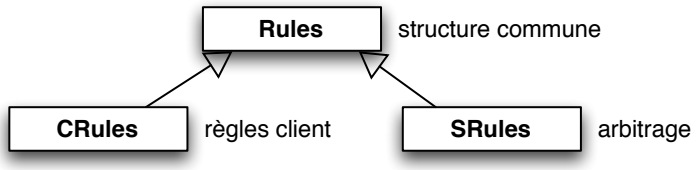


FIG. 6.1 – Hiérarchie de la classe `Rules`.

6.1.2 Les classes représentant les règles

Les classes candidates (liste non exhaustive, rassemblé lors d'une inspection rapide de l'ancien code) à ce passage sont :

- `Ball` : Le ballon.
- `Public` : Le public autour du terrain de jeu, événement aléatoire et incontrôlable.

- `Map` : Représente le terrain de jeu. Elle possède quelques méthodes interessante, comme un pathfinding.
- `Player` : Un joueur de BB, n'importe lequel. Faut pas les oublier, ces petites bêtes là !
- `Team` : Container pour joueur.

Dans ces classes ne devront se trouver que la structure de donnée, et quelques méthodes communes aux deux parties, règle client et arbitrage. Dans chacune des deux parties, si il y a besoin de rajouter une méthode spécifique, hériter de la classe en la préfixant par **S** pour le serveur et **C** pour le client, et ajouter/surcharger tout ce dont on a besoin. Cette convention de nom devrait être suffisamment claire et explicite.

6.1.3 Les méthodes de transmission de messages

Explication des mécanismes permettant l'envoi et la réception de paquets. Attention, on va s'enfoncer un peu dans le code. Le fonctionnement est entièrement symétrique pour l'arbitrage ou les règles clients (en fait, le code gérant cela se trouve dans la partie commune).

Globalement, vous avez une méthode `sendPacket()` dans la classe `{S,C,}Rules` qui vous permet d'envoyer des paquets à l'autre bout. Pour la réception, il y a un système de callback. Il faut déclarer au début quelle méthode recevra quel type de message, et cette méthode sera appelée avec le paquet dès qu'il se présentera. Les callbacks peuvent également être définit en fonction de l'état de la partie.

Comme il est pratiquement certain que vous n'avez pas tout compris[1], on va le refaire au ralenti sur un exemple concret. Prenons le placement du ballon au début de la partie. Un client choisit son placement, le serveur le fait dévier et le renvoie aux clients. Pour bien suivre le parcours des messages à l'execution, vous pouvez suivre la figure 2.6. Bien qu'elle décrive le déplacement d'un joueur, le traitement est quasi identique.

```
DECLARE_PACKET(MSG_BALLPOS, MsgBallPos, Ball)
  int row;
  int col;
END_PACKET

class Ball {
  ...
```

Listing 6.1 – (*Partie commune*). Déclaration du paquet, et de son contenu.

```
CBall::CBall(CRules* sr)
  : cr_(cr)
{
```

[1] Avez-vous lu, d'abord ? :)

```
 cr_->handleWith(new PacketHandler<MSG_BALLPOS>(this, &CBall::onBallPos));
}
```

Listing 6.2 – (*Règle clients*). La méthode textttonBallPos() recevra le message du serveur.

```
SBall::SBall(SRules* sr)
 : sr_(sr)
{
 sr_->handleWith(new PacketHandler<MSG_BALLPOS>(this, &SBall::onBallPos));
}
```

Listing 6.3 – (*Arbitrage*). La méthode `onBallPos()` recevra la demande de placement du ballon du serveur.

```
void CBall::init(int row, int col)
{
 MsgBallPos pkt;
 pkt.row = row;
 pkt.col = col;
 cr_->sendPacket(pkt);
}
```

Listing 6.4 – (*Règle clients*). La méthode `init()` prend en parametre les coordonées qu'a choisit le joueur pour placer le ballon, et l'envoie au serveur.

```
void SBall::onBallPos(const MsgBallPos* pkt)
{
 pkt->row += rand(10) - 5;
 pkt->col += rand(10) - 5;
 sr_->sendPacket(*pkt);
}
```

Listing 6.5 – (*Arbitrage*). Le serveur recoit la demande du joueur, fait rouler un peu la balle aléatoirement, et renvoie la vraie position du ballon aux clients.

```
void CBall::onBallPos(const MsgBallPos* pkt)
{
 row_ = pkt->row;
 col_ = pkt->col;
 onEvent(eBallPosition);
}
```

Listing 6.6 – (*Règle clients*).Le client recoit les vraies coordonnées du ballon, les stocke en interne, et envoie un événement à l'UI.

6.2 Le composant arbitrage : le moteur de règle

C'est évidemment un gros morceau, il serait sage de bien modéliser son fonctionnement interne. Quelques graphiques ayant été fait lors de précédentes tentatives de refactorisation sont à regarder, notamment ceux sur les actions[2]. Ici, on récupère toutes les déclarations de classes faites dans `libcommon.la`, donc nos objets sont déjà pourvu de méthodes permettant les transferts client/serveur.

Ca c'était pour la théorie, la pratique sera certainement moins jolie. La méthode retenue est de tout coder en dur, sans externaliser une quelconque configuration de règle dans un fichier de configuration. Tant que le code reste localisé en certain endroit précis, il sera peut-etre possible, plus tard et avec le recul, de rendre ce code un peu plus extensible.

6.2.1 Tips and tricks

Le système de coordonnées retenu et utilisé est schématisé sur la figure 6.2.1, et correspond à celui utilisé pour les matrices. Le terrain est présenté de la même manière que dans TBT 0.5. Ne surtout pas utiliser `x` et `y`, qui, pour le coup, ne ferait qu'apporter une grosse confusion. Les constantes COLS et ROWS sont définies comme limite. (respectivement 15 et 26). Les tableaux, comme en C, partent de zéro jusqu'à `n-1`.

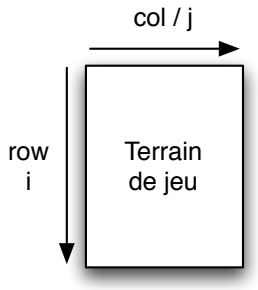


FIG. 6.2 – Système de coordonnées du terrain de jeu.

6.3 Le composant règles client

L'interface devrait ressembler à cela :

```
class CRules {
  get*(); // Récupere les objets player, ball, weather, ...
  registerCallback();
  getState();
  process();
};
```

Listing 6.7 – Interface pour la gestion des clients.

[2]Voir le fichier `action.pdf`, dans le répertoire doc sur le svn.

Cet objet est toujours dans un certain état, qu'il est possible de connaitre, par exemple INIT_GAME, PLAY_OURS, PLAY_OTHER, …. `process()` serait la fonction principale, qui irait voir si il y a des paquets à récupérer, changerait d'état, appellera des callbacks, …. A propos des callbacks, l'utilisateur peut définir un certain nombre d'évenements pour lesquels il aimerait être informé (déplacement d'un joueur, …), et notre objet `CRules` ira lui appeler la méthode demandé en temps voulu.

LE SERVEUR DE JEU

Cette partie explique le binaire serveur produit, pour la configuration *réseau*.

`Server` est la partie qui contient le `main()` et est responsable des connexions avec tous les clients (coachs et spectateurs). Elle contient plusieurs instances de `Client`, une pour chaque client connecté. Elle contient également une instance de `GameServer`, qu'elle appelle à l'occasion.

7.1 Las écurité, où ¨t'es gentil, H4k3rZ, va jouer ailleurs¨

Le code source étant disponible, n'importe qui peut s'amuser à modifier les sources, se rajouter 3 ou 4 champions invincibles, compiler le tout et exploser son pauvre petit frère qui décidément n'a pas encore tout compris à la vie.

Pour essayer de ne pas tout confondre en parlant de tout et de n'importe quoi, reprenons les cas d'utilisation :

- on veut pouvoir jouer tout seul à un ou deux sur un pc, sans réseau.
- on veut pouvoir jouer entre deux potes de confiance à distance, en se connectant à un serveur.
- on veut pouvoir jouer entre deux personnes à distance, en se connectant à un serveur, sans que ces personnes ne se fassent trop confiance.
- lors des matchs de ligue, des serveurs seront lancés, et on veut pouvoir se connecter dessus et jouer (via le gestionnaire de ligue).

Dans les cas 1, 2 et 3, les coachs sont maître de toute la chaîne du processus, et quelles que soient les techniques mises en place, ils pourront les contourner. Dans les cas 1 et 2, tant mieux, ces personnes pourront modifier leur programme dans un bon esprit et faire ce que bon leur semble. Même, c'est une attitude à encourager, ça peut déboucher sur des trucs sympa qu'il sera possible de réintégrer ensuite.

Dans les cas 3 et 4, c'est plus problématique, et il convient encore de séparer les possibilités de ¨triche¨ en deux parties :

- Le choix des équipes.
- Le déroulement du jeu en lui-même, après que le ballon soit lancé.

Concernant le choix des équipes, en mode ligue, il n'y a pas de problème, puisque les équipes sont gérés dans la continuité par le serveur de ligue. Donc pas moyen de tricher. Dans le cas d'utilisation trois, où on a envie de faire une partie avec quelqu'un qu'on vient de rencontrer sur un forum et auquel on ne fait pas confiance, il peut sortir une équipe complètement délirante[1]. Dans ce cas, comme en jeu plateau, un coach est libre de refuser la partie. Une autre possibilité serait de limiter les 2 coachs aux équipes débutantes, sans aucune expérience.

Pour le second point, le problème est très restreint. Le serveur prend toutes les décisions importantes concernant le déroulement de la partie et comme les clients n'ont pas accès entre eux, il est impossible pour un client de faire une action impossible. Celle-ci se verra rejeter par le serveur, le second client comme le déroulement de la partie n'en sera nullement affecté. Pour résumer, *l'intégrité du serveur garantie l'intégrité de la partie*. Lors des matchs de ligue, le serveur est sous le contrôle des administrateurs de la ligue, donc pas de soucis. Pour un match entre deux personnes ne se faisant pas confiance, pas de miracle, il faut qu'ils se connectent sur un serveur en *zone franche*.

Des solutions à base de comparaison de valeur de hash sur les binaires/sources, ou de cryptographie asymétrique pour assurer l'intégrité des binaires ont été proposé, mais pas retenu, pour diverses raisons.

Il y a aussi les failles de code localisés (*buffer overflow*, ...), mais, à part savoir les détecter et les corriger, on n'y peut pas grand chose. Il est bon d'éviter ces failles en amont.

7.2 La gestion des clients

Gérer tous les clients, spectateurs... Comment ca, section inutile ?

7.3 Log/Replay

Pendant le déroulement d'une partie, un fichier est ouvert en écriture du coté du serveur. Ce fichier se comporte de la même manière qu'un client spectateur, c'est à dire qu'il récupère et

[1] C'est bien, tu as réussi à modifier un fichier xml.

stocke tous les messages `Serveur -> Client` au cours de la partie.

Pendant le replay d'une partie : le fichier de log est ouvert par le serveur, les spectateurs se connectent comme d'habitude. Le serveur ne reçoit pas de demande d'ordres des coach actifs (d'ailleurs, il n'y en a pas), ni ne contrôle par rapport aux règles. Il se contente d'envoyer les résultats contenus dans le fichier de log, tour à tour, aux clients. Il faut voir, plus tard[2], pour intégrer un contrôle de lecture (avance rapide, aller à un certain tour, ...).

On peut aussi inventer toute sorte de moulettes pour ce fichier de log, création de statistique sur le nombre de TouchDown, ...

[2] "Un jour".

CLIENTS

FIG. 8.1 – Schéma simplifié d'un client

Il y a une grosse partie, qui contiendra le point d'entrée (`main()`), qui varie selon le *visage* du client, est qui est encore totalement indéfinie. Dans tous les cas, elle devra contenir une instance de `GameClient`. `GameClient` est responsable de la connexion avec le serveur, et contiendra quelques méthodes pour la *maintenance* (comme connectToServer, disconnect, ...), mais rien en soi pour le jeu. Cette dernière à une instance de `CRules`, lui fournit les paquets reçu du serveur et envoie les paquets qu'il veut envoyer. `CRules` est prévu pour contenir tout l'état du jeu (tel que fourni par le serveur), plus quelques vérifications locale triviales (Inutile de demander au serveur si on peut bouger un joueur en dehors du terrain, par exemple).

8.1 Client IA

Exactement comme un client humain, excepté l'interfaçage épineux avec la visualisation. Il est lié à la bibliothèque *règles client* pour toute interaction avec le jeu et les transferts avec le serveur. Il peut aussi être écrit dans un autre langage plus propice à ce genre d'algo (OCaml, Haskell, Python, ...).

Dans ce cas de l'utilisation d'un autre langage, il est suggéré (mais pas impératif) de prévoir des bindings pour lier les 2 parties (l'IA dans le langage cible et les bibliothèques précédemment cités), en le faisant à la main, en utilisant swig, ...).

Cette partie ne présente aucune difficulté majeure pour l'intégration, peut être codé séparément, et à vrai dire, n'est pas nécessaire pour une première version jouable de TBT. Par contre, les

tripes et autres éléments interne de l'IA sont laissés à l'entière responsabilité de leur concepteur[1].

8.2 Visualisation graphique

En 2D, ok, en 3D, ca pourra être concevable ! Pour l'instant, arrêtons les rêves de grandeur[2], et restons sur la version 2D SDL. Bon, là, clairement, mes capacités de modélisations s'arrêtent là. On y va a l'arrach', et on dessine le graphique UML après. C'est aussi une partie où il y a beaucoup de code à récupérer de l'ancien TBT.

Concernant les graphismes, actuellement le consensus est de conserver ceux existant dans la version 0.5, ainsi que tout le look&feel. Néanmoins, on reste ouvert aux propositions.

Allez, rapide explication de la structure du code du client graphique en SDL. Comme le temps me manque, on va faire ca style SMS.

8.2.1 Présentation du moteur

SDL est une bibliothèque simple, très simple, et finalement assez bas niveau. On s'en sert pour l'affichage (video), l'input (clavier et souris), et le son (pas encore codé).

Pour la video, SDL fonctionne avec des surfaces, des rectangles en réalité (SDL_Surface). L'ecran (ce qui est effectivement affiché à l'utilisateur) est aussi une surface (pour l'instant fixée a 800x600). Une image est également une surface. L'affichage de texte passe aussi par une surface. Pour finir, on peut aussi créer des surfaces 'vierges'. La seule opération permise sur les surfaces est la copie d'une surface vers une autre. On peut copier seulement une partie d'une image, appliquer de la transparence au passage, etc... Mais au final, ca revient toujours a des copies de rectangles.

Une fois toutes les surfaces chargées en mémoire, la première difficultée est de les copier sur l'écran dans le bon ordre, et au bon endroit. Copiez d'abord un joueur, puis le fond d'écran, et vous pouvez être sur que vous ne verrez jamais le joueur. De plus, avec le terrain qui peut être scrollable, gérer la différence entre les coordonnées du jeu et celle de l'écran est souvent un casse-tête. La deuxième difficultée est le temps de rendu. Si a chaque frame, vous essayez de recopier la totalité des surfaces sur l'écran, ca va ramer de manière inacceptable. Il faut s'arranger pour ne recopier *que* ce qui a besoin de l'être sur l'écran.

[1] Autrement dit : démerdez vous pour faire quelque chose de potable :).
[2] Dis, papa, comment est-ce qu'on conquiert le monde ?

La troisième difficulté, qui n'en est pas vraiment une, est que l'API SDL est en C. Les appels sont des fois un peu rébarbatifs, il faut gérer sa mémoire correctement, …. Vu qu'on code en C++, il est plus simple de tout wrapper dans des jolies classes, et ne plus a avoir à gérer le côté bas niveau. Tout ca pour introduire mon mini-moteur SDL, qui élimine au maximum ces 3 problèmes. En fait, il permet même au final d'afficher des choses à l'écran en moins de 10 lignes et sans appeler directement une seule fonction de la SDL.

Un petit exemple d'utilisation, qui permet d'afficher et d'animer la roue qui tourne (le sablier de tbt 0.5). les détails seront expliqués plus tard :

```
Sprite wheel("image/wheel"); // Cree une sprite a partir d'une image png
wheel.setZ(1);        // Position z de la sprite
wheel.setPos(113, 506); // Position x,y de la sprite
wheel.splitNbFrame(13, 1); // Splitte l'image en 13 miniatures
wheel.anim(100);      // Change de miniature toutes les 100 ms
screen_.addChild(&wheel_); // Ajoute la sprite sur l'écran.
```

Et c'est tout ! Tout le reste, affichage, rendu quand il faut, …, est géré automatiquement.

8.2.2 Détails du moteur

La visualisation générale de la partie graphique est décrite sur la figure 8.2.2. La partie graphique est divisée en deux parties principales, le moteur SDL (indépendant du reste du programme) et la partie qui s'occupe de plus précisément de TBT. Cette dernière partie est relié à la connection client, qui permet la communication avec le serveur, ainsi qu'avec l'API, les règles client, qui vont nous fournir toutes les informations utiles sur le jeu.

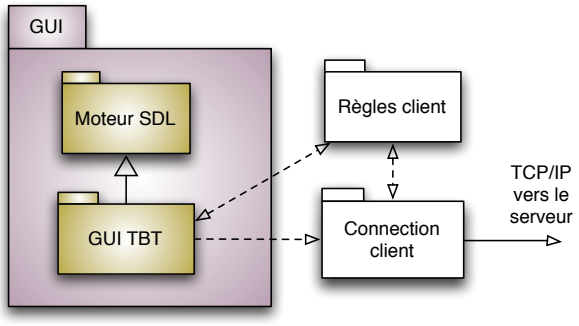
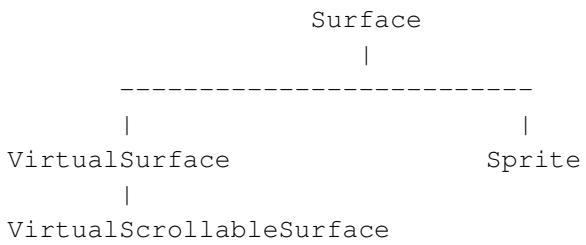


FIG. 8.2 – Relation par paquetage de la partie graphique.

La modélisation est la suivante :

```
                    1
SDLWindows ----------- Input
                |---- ResourceCenter
```



`SDLWindows` est la classe qui gère la création de la fenêtre SDL, de la surface écran, et des événements utilisateurs. Il faut lui appeler régulièrement (== boucler dessus) la méthode `processOneFrame`. C'est tout. Elle possède une instance de `Input` et `ResourceCenter`.

`Input` est une classe toute simple qui contient l'état du clavier et de la souris. Elle peut être récupérée par n'importe quelle classe qui veut faire des "choses interactives".

`ResourceCenter` est une classe essentiellement utilisé en interne. Elle s'occupe de charger et de mettre en cache les images et les fontes, histoire de ne pas charger 15 fois la même image. Vous n'auriez pas à priori à l'utiliser directement.

`Surface` est la classe de loin la plus intéressante. Elle contient en interne une (et une seule) structure SDL_Surface, et permet de stocker une image, ou être utilisée comme surface pour un affichage intermédiaire (en utilisant VirtualSurface derrière). Elle possède une plétore de fonction pour placer la surface (position), taille, position en Z (0 -> background, supérieur à 0 -> niveau de foreground), ... Utilisez `Surface` si vous ne voulez qu'afficher une image. Pour l'animation, le déplacement visuel, utiliser plutôt `Sprite`.

`Sprite` est un raffinement de `Surface`, qui permet de faire des choses un peu plus évoluée, comme splitter une image qui est en réalité une collection de miniature et n'afficher qu'une miniature ou l'animer, gère le déplacement... En fait, les fonctionnalitées sont rajoutés au fur et a mesure des besoins dans le reste de tbt. Il est toujours possible de créer d'autres classes dérivant de `Surface` ou de `Sprite` pour faire des trucs plus spécifiques et qui seront utilisé un peu partout[3], par exemple un menu contextuel (popup), une boite de dialogue, ...

`VirtualSurface` est la seconde classe la plus importante derrière `Surface`. Elle peut être vue comme une sorte de *container*, dans lequel des Surfaces (et d'autres VirtualSurface, aussi !) peuvent être ajouté. Elle s'occupe ensuite de mettre à jour les objets qu'elle contient, et effectue le rendu dans sa propre surface[4]. Elle dérive de `Surface` pour avoir sa SDL_Surface[5] et récupérer toutes les méthodes de positionnement de la surface. Finalement, la surface SDL principale qui est

[3] oui, c'est de la POO, quoi...
[4] Plus d'explications un peu plus bas (oui, dans le FIXME...)
[5] Oui, il est dit plus haut qu'il est possible de créer des surfaces vierge, c'est a dire ne provenant pas directement d'une image

affiché à l'ecran est de type `VirtualSurface`, sauf qu'à la création, on lui passe le SDL_Surface de l'écran plutot qu'il en crée un temporaire.

`VirtualScrollableSurface` est un raffinement de `VirtualSurface` pour permettre la création d'une surface *virtuelle* qui sera plus grande que ce qui sera affiché réellement à l'écran. Utilisé pour l'affichage du champ de bataille. D'ailleurs, ca pose pas mal de soucis, d'où la fonction `getRealSurface()` dans la classe `Surface`.

FIXME : expliquer en détail la relation père fils des VirtualSurface et des Surface, la procédure de rendu d'une frame (méthode update appelée recursivement depuis la surface parent, l'écran, puis render), et la gestion de la mémoire sur les Surface (le refcount).

8.2.3 Et tbt, dans tout ca ?

On a un moteur, il faut bien en faire quelque chose. On va donc contruire un truc par dessus ca. `Game` est la classe principale, qui contient le `SDLWindows` et toutes les autres instances de classe. Les autres classes devraient être assez explicite.

```
   Sprite      VirtualSurface    VirtualScrollableSurface
     |            |      |                |
 VisuPlayer     Panel    |            VisuField
                     ActionPopup
       Game
```

Ces classes font régulièrement appel à l'API des régles de tbt, pour récupérer toutes les infos nécessaires sur le jeu, la position des joueurs, etc... Les classes et les méthodes sont pas encore fixées, ca peut bouger en fonction des évolutions. Pour l'instant, la question de "quelle classe fait quoi" est pas franchement résolue, ca se fera au fur et à mesure en fonction de la cohérence/facilité.

8.2.4 Considérations diverses

La vitesse : Pas mal de choses, dans la hiérarchie de classe, fonctionne avec des méthodes virtuelles (il n'y a quasiment aucune template). On peut facilement dire que `virtual` est lent, mais ca dérange pas trop. En fait, je suis parti du principe que l'opération prenant tout le temps CPU est le blit. Donc, une fois les blits sont a peu près optimisé (cad, en faire le moins possible), ca suffit. Ca veut dire que je ne cherche même pas a optimiser le reste du code, on verra plus tard a coup de callgrind si il y a des parties de code trop consommatrice.

Les événements : Il y a un sytème d'événement pas trop explicité (bb5/client/Event.hh, dont hérite la class `Game`) qui permet de détourner les messages arrivant dans les règles client pour appeler directement des méthodes virtuelles dans l'interface graphique (via la classe `Event` et `Game`). C'est basé sur un vieux kludge, et finalement, je ne sais pas trop si ca va être utile ou même si c'est désirable. Tout ce qui arrive par ce système devrait aussi être récupérable en appelant explicitement les fonctions de l'API des règles. Simplement, c'était déjà en place quand j'ai commencé la gui, ca faisait moins de lignes de code a taper, ...Voir si c'est utile.

DIVERS

9.1 Son/Musique

Il serait agréable d'avoir une musique assez prenante pendant le menu, peut-être en fonction des races. Pendant le jeu, par contre, le consensus semble s'orienter vers une partie sans musique.

Pour les animations sonores durant le jeu, plus y'en aura des biens et des différentes, mieux ce sera :)

9.2 Manuel/Aide au joueur

Une aide en ligne, c'est à dire ajout d'un menu **Aide**, et affichage de petites bulles d'information dans le jeu serait très appréciable, mais pourra être fait plus tard est n'est pas une priorité.

Pour l'instant, un bête manuel off-line, sur quelques pages html avec des captures d'écrans et quelques notes explicatives pourront être données[1]. En plus, un non développeur pourra s'occuper de cela. L'avoir en français et en anglais serait un plus.

9.3 Editeur d'équipe

Pour sélectionner une équipe et la placer sur le terrain une fois pour toute, et ne plus à avoir à le faire à chaque début de partie. La composition de l'équipe est ensuite enregistrée dans un fichier, dont le format exact est encore à définir. Il sera en XML, et devrait se rapprocher le plus possible de ce qu'utilise actuellement javaBB.

[1] Bon, une fois qu'il y aura des captures d'écran à faire

Cette partie est optionnelle, et peut être réalisé en parallèle avec le reste. Ça serait une autre application complètement à part, faite en GTK, QT, Ruby/TK, . . .le premier qui s'y colle choisit !

Il en existe déjà une, en php/html, accessible et fonctionnelle sur le site. Rien n'empêche d'en avoir une deuxième livrée avec les autres binaires de TBT et fonctionnant sans serveur web.

9.4 Documentation et internationalisation

L'anglais a été choisi comme principale langue de communication (code, commentaire, documentation, rapports de bugs) pour améliorer la visibilité du projet. Le francais restera d'usage dans les communications courantes (mailing, wiki, forum, . . .), jusqu'à ce qu'il y ait un plus grand nombre d'anglophones[2].

Également, un mécanisme de traduction du texte (utilisation de `gettext`) devra être prévu dès le début dans le code. Pour l'instant, retenons 2 langages dans lequel le jeu sera traduit : le français et l'anglais. Plus si affinité.

En ce qui concerne la documentation, un extracteur de documentation (Doxygen) ne saurait être suffisant. Il est très bon pour la documentation d'un bout de code à plat, comme une bibliothèque, mais pas forcément pour refléter une modélisation comme celle proposée. Deux types de docs devront être gardés à jour le plus possible :

- Une documentation décrivant la modélisation générale, décrivant les diverses interfaces entre modules, et reflétant l'esprit du soft, genre ce document en un peu plus étoffé. Si quelqu'un sait comment obtenir la même chose plus automatiquement qu'en maintenant quelque chose de séparé, je suis preneur,
- Une autre collant plus au code, sous forme de commentaires dans le code, plus une description des interfaces sous format Doxygen.

9.5 Convention de code

Avoir un code homogène est agréable, mais il ne faut pas non plus que cela devienne une contrainte. Il a été choisi d'édicter nos propres règles, et de ne pas suivre de standard prédéfini (style Gnu). Ces règles sont disponibles sur le wiki[3].

[2] Ou pas.
[3] https://projects.nekeme.net/projects/tbt/wiki/ConventionDeCode

Il faut éviter de faire des erreurs de code qui peuvent devenir des failles. Elles sont en général assez difficile à détecter dès qu'il y en a un peu dans tous les sens, autant les éviter en amont. On fait du C++, pas du C, donc poubelles toutes les fonctions style sprintf, strcmp, …. A la place, il vaut mieux utiliser les iostreams du C++, les containers de la STL, qui sont un peu plus sûr.

9.6 Autoconfisquerie

Les *autotools* seront responsables de l'ensemble de la configuration du projet, de la génération des Makefile's et de la tarball distribuable. Ainsi, il pourra être porté facilement sur tout système unix (Linux, BSD, Darwin, …) sans trop de problème, et la création de rpms et de paquets debian en sera grandement facilité.

IDEAS FOR THE (MAYBE FAR) FUTUR

Quelques idées, en vrac, de fonctionnalités qu'il sera possible de rajouter, avec pour chacune des propositions d'implémentations en utilisant le système actuel (avec le moins de modifications possible, que des ajouts). Cette partie est surtout là pour éviter un jour de vouloir rajouter une nouvelle fonctionnalité, de s'apercevoir qu'il n'est pas facile/possible de le faire, et de devoir tout recoder.

10.1 Ajout de nouveaux jeu de règles

C'est une des fonctionalité qui semble la plus demandé, d'avoir un système qui permette à l'utilisateur de choisir de jouer avec n'importe quelle version du LRB, ou avec des règles dérivées (BeachBowl). Le code est plutôt bien séparé et cloisonné, mais le problème est que de telles modifications demandent de toucher à *quasiment* tout le code (parties arbitrage, règles clients, et la gui doit aussi s'adapter). Autant dire qu'en pratique, c'est la misère totale, et il serait étonnant qu'un jour TBT soit capable de telle prouesses. Néanmoins, quelques idées :

- Si les différences dans les règles sont minimes et que l'affichage ne diffère pas beaucoup, il est possible de modifier le code existant et d'ajouter des conditions, selon tel ou tel jeu de règles, faire ceci ou cela... Au risque de se retrouver avec un gros paté de code inmaintenable.
- Arriver a extraire une partie de code commune pour le serveur et le client, et en dériver pour chaque implémentation de règle. Pour l'interface graphique, prévoir le même système pour les parties qui changent. C'est probablement le système le plus viable, mais demandant un énorme effort de réflexion et de code.
- Faire un système générique pour gérer tout les situations possibles, créer un DSL pour décrire les règles et leur interactions, et avoir une interface graphique complètement configurable. Cette solution ressemble à la quête du graal, mais si personne ne l'a jamais fait, y'a bien une raison. Néanmoins, si vous êtes prêts à payer...

En résumé : revoir un peu toutes les parties de code.

10.2 Le jeu par courriel

Une version de jeu par email est faisable. Le principal changement serait que le serveur ne pourra pas être en route tout le temps (les parties pouvant durer des mois), il faut donc un moyen de le décharger/recharger en mémoire. C'est une fonctionnalité assez proche de la sauvegarde de partie en cours, pour le jeu normal.

La partie `arbitrage` pourra être conservée telle quelle, ainsi que la partie `règles client`. Par contre, il faudra refaire une interface coach, en s'inspirant de l'interface console existante et en lui faisant comprendre des ordres contenus dans un mail. La version *standalone* serait préférable, car il est plus facile de recharger une partie dans ce mode.

En résumé : supporter la sauvegarde/chargement de partie, et coder un nouveau client.

10.3 Serveur pouvant accueillir plusieurs jeux

Il serait possible de faire un serveur acceptant, pour un seul processus, plusieurs parties simultanément. Par rapport à lancer 150 processus serveur, ca aurait pour avantage un gain au niveau de la mémoire, du cpu (context switch), et des sockets, en évitant d'ouvrir 150 ports en écoute sur le serveur. En fait, cette fonctionnalité a déjà son FIXME : dans le code. Il suffit de modifier le binaire sur serveur pour accepter plusieurs instances de `GameMaster`, rajouter un identifiant à la partie pour que le serveur sache sur quelle partie rediriger, et modifier l'interface graphique avec un menu pour sélectionner quelle partie rejoindre, après une connection réussie à un serveur.

En résumé, améliorer une partie isolée du serveur, et revoir le protocole de communication avant (et uniquement *avant*) le début d'une partie.

CONCLUSION & FUTURE WORK

Vu qu'on repart de zéro, il reste pas mal de boulot à faire. Pour l'instant, l'objectif est une version 0.7 qui ne doit pas être trop inférieur à l'actuelle 0.5. On devrait retrouver un moteur de règles quasi complet, au moins jouable, une interface graphique similaire, et le mode réseau fonctionnel. Le gestionnaire de ligue, l'intelligence artificielle et l'éditeur d'équipe restent optionels pour la 0.7, mais si ils sont terminés d'ici là, tant mieux.

Au vu de tout ce qui a été dit, le travail peut être découpé en plusieurs bout, sans trop d'interaction (uniquement des interaction au niveau des formats de fichier ou protocole réseau) entre chaque. Voici un exemple de découpage, qui ne parait pas totalement aberrant :

- Module réseau,
- Moteur de règle,
- Interface graphique 2D,
- Client IA,
- Gestionnaire de ligue,
- Editeur d'équipe,
- Packaging (autotools, binaire windows, paquets debian, rpms, ...),
- Documentation générale (manuel de jeu, traduction, ...).

A partir de maintenant, si vous êtes interessé par une partie, vous pouvez soit compléter les spécifications sur ce présent document, soit directement commencer à coder. Faites vous connaître de notre GO[1], TuxRouge, qui se fera ensuite un plaisir de vous harceler par mail.

Happy Coding !

[1] Pas Trop Méchant Organisateur

GLOSSAIRE

Comme dans tout projet, nous avons notre propre terminologie sur certains termes, pour décrire rapidement telle ou telle partie. Il y déjà eu pas mal d'incompréhension entre nous, provoquant de petits *trolls*, chacun parlant de choses différentes. Ce glossaire tente de clarifier certaines expressions.

- Quand on parle des *règles*, au sens général, on se réfère aux modules règles client et arbitrage.
- Pour parler plus spécifiquement de la partie coté client, on utilise le terme *règles client*[1],
- Pour décrire les règles coté serveur, on utilise le mot *arbitrage*.

[1]Je n'ai pas encore trouvé de terme plus adéquat, plus *corporate*, si vous avez une meilleure idée...

Bibliographie

[projects, 2006] projects, A. (2006). Xerces c++ home page. http://xml.apache.org/xerces-c/.

[SkiJunkie, 2006] SkiJunkie (2006). 3dbowl home page. http://home.austin.rr.com/javabbowl/index.html.

[Soft, 2005] Soft, G. (2005). 3dbowl home page. http://gf-soft.net/index.php?dir=inicio&lang=en.

[Team, 2005] Team, R. (2005). Rudebowl home page. http://membres.lycos.fr/rudebowl/.

[Team, 2006] Team, T. (2006). Towbowltactics home page. http://www.towbowltactics.com/.

[Tokiros, 2005] Tokiros (2005). 3dbowl home page. http://www.tokiros.org/pyBloodBowl/index.php/2005/04/24/1-first-post.

[Wikipedia, 2006] Wikipedia (2006). The answer to life, the universe, and everything. http://en.wikipedia.org/wiki/The_Answer_to_Life,_the_Universe,_and_Everything.