

Architecture d'un jeu vidéo 3D

par Pierre Schwartz 

Date de publication : 5 juin 2007

Dernière mise à jour : 5 juin 2007

Cet article présentera un modèle d'architecture pour la réalisation de jeux vidéos en 3D.
Connaissances requises : C++.

| | |
|--|----|
| I - Introduction..... | 3 |
| I-1 - Schéma général..... | 3 |
| I-2 - Les contraintes, les objectifs..... | 4 |
| II - Les différents modules..... | 5 |
| II-1 - Communication entre les modules..... | 7 |
| II-1-A - Les messages..... | 8 |
| II-2 - Ajout d'un module..... | 9 |
| II-3 - Utilisation des modules en mode passif..... | 9 |
| II-3-A - Implémentation du système de console..... | 10 |
| II-3-B - Un client pour la console..... | 11 |
| III - Le moteur graphique..... | 13 |
| III-1 - Vue d'ensemble..... | 13 |
| III-2 - Le graphe de scène..... | 13 |
| III-3 - Constructeur du moteur graphique..... | 14 |
| III-3-A - Un petit mot sur CEGUI..... | 14 |
| III-4 - A chaque frame du moteur graphique..... | 16 |
| III-5 - Les entrées clavier / souris..... | 16 |
| IV - Le moteur audio..... | 19 |
| V - Le moteur de réseau..... | 21 |
| V-1 - La sérialisation des messages..... | 22 |
| V-2 - Envoi / réception des messages..... | 22 |
| V-3 - Les sockets pour atteindre une machine réseau..... | 22 |
| V-4 - Spécification réseau local ou réseau Internet..... | 23 |
| V-5 - Réception des messages UDP..... | 25 |
| V-6 - Recherche de parties disponibles..... | 27 |
| V-7 - Propagation des messages..... | 28 |
| VI - Moteur de jeu..... | 29 |
| VI-1 - Intelligence artificielle..... | 29 |
| VII - Pour aller plus loin..... | 30 |
| VII-1 - First Person Shooting..... | 30 |
| VII-2 - Real Time Strategy..... | 30 |
| VII-3 - Jeu de simulation / course..... | 30 |
| VII-4 - Jeu massivement multijoueurs : MMO..... | 30 |
| VII-5 - Les points à ne pas négliger..... | 31 |
| VII-6 - Cohérence entre les joueurs..... | 31 |
| VII-7 - Téléchargement..... | 31 |
| VII-8 - Remerciements..... | 32 |



I - Introduction

Je vais présenter un modèle d'architecture pour la réalisation d'un jeu vidéo en 3D. Ce modèle devra pouvoir s'adapter simplement aussi bien à la réalisation d'un jeu à la première personne (FPS), d'un jeu de stratégie (RTS) ou même à d'autres types de jeux. L'architecture présentée pourra également gérer les jeux en mode multijoueurs en réseau local ou sur internet. De même, l'architecture pourra être simplement étendue pour la réalisation de jeux massivement multijoueurs: MMOFPS, MMORTS, MMORPG ...

Cette architecture sera entièrement codée avec des bibliothèques multi-plateformes.

L'objectif n'étant pas de fournir un jeu commercialisable, mais un modèle d'architecture sur lequel se baser pour réaliser un jeu.

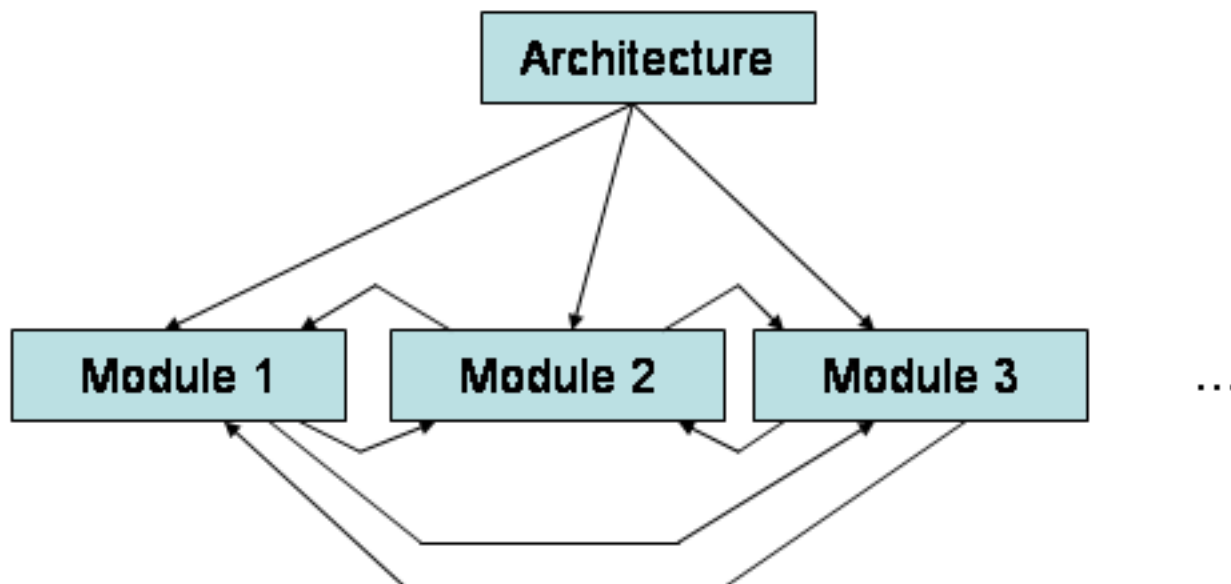
I-1 - Schéma général

Même sans savoir quel type de jeu on va écrire, il y aura toujours un socle commun, permettant de faire dialoguer des modules, permettant d'interfacer les différents moteurs mis en jeu.

Je vais utiliser les outils suivants :

- Irrlicht pour le rendu 3D la gestion des entrées utilisateur
- CEGUI pour la gestion des menus et des éléments graphiques non 3D
- asio pour la gestion du réseau et les threads
- Irrklang pour le son
- boost

Le jeu contiendra des modules plus ou moins indépendants, gérant chacun un aspect du jeu. L'objectif étant bien sûr de rendre possibles les développements des différents modules en parallèle, sans que ça n'entraîne d'effets de bords dans les autres modules.

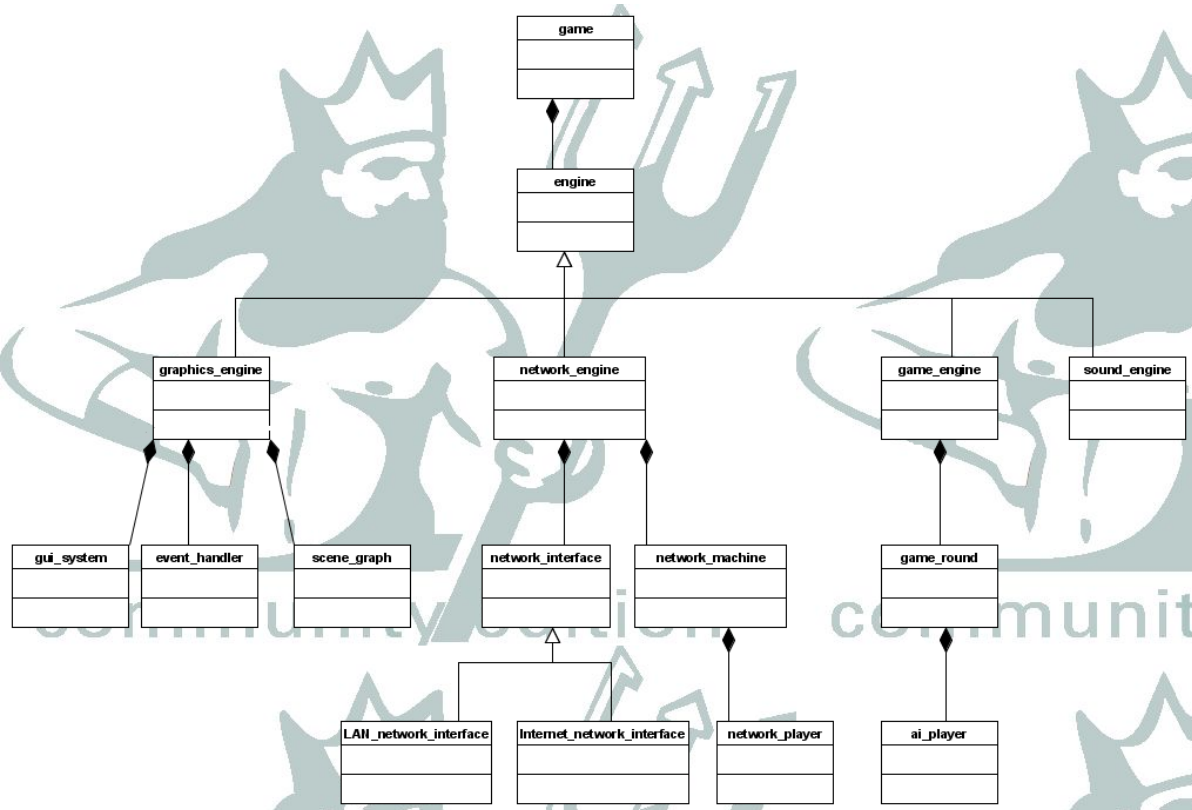


Un objet contiendra tous les modules. D'un point de vue purement conceptuel, l'architecture peut contenir un nombre non défini de modules, il faut donc les stocker dans un conteneur style `std::list`. Chaque module aurait un pointeur vers l'architecture, permettant ainsi d'accéder aux autres éléments de la liste des modules.

I-2 - Les contraintes, les objectifs

- L'architecture devra pouvoir convenir à un maximum de types de jeu, il ne faut donc faire aucune supposition sur l'utilisation qui en sera faite.
- Nous avons une très forte contrainte de performances : l'architecture proposée devra être aussi légère que possible tant en ressources processeur, qu'en mémoire ou en espace disque nécessaire.
- L'architecture devra aussi être légère en utilisation du réseau, de manière à favoriser des échanges rapides et supportés par les bandes passantes les plus modestes.
- Les modules étant les pièces de base de l'architecture, ils devront pouvoir communiquer simplement et rapidement entre eux.
- Dans le cas d'un jeu multijoueurs, il faudra pouvoir tirer profit des machines puissantes pour leur assigner des tâches telles que la gestion de personnages non joueurs.

Voici un diagramme de classes simplifié pour les principales classes mises en jeu :



II - Les différents modules

Voici la classe de base de notre architecture, celle qui sera instanciée pour démarrer le jeu.

classe de base pour notre architecture

```
class game{
public:
    game();
    ~game();
    // ...

private:
    std::list<engine*> l_modules;
};
```

Il va nous falloir une classe générique pour représenter un moteur du jeu :

classe générique pour un moteur

```
class engine{
public:
    engine(game*);
    virtual ~engine();

    // accepter les messages des autres moteurs
    void push_event(engine_event& e){
        events_queue.push(e);
    }

    // traite la file des messages
    void process_queue(){
        while (! events_queue.empty()){
            engine_event e = events_queue.front();
            events_queue.pop();


            process_event(e);
        }
    }

    // traitement propre à chaque moteur
    virtual void frame() = 0;
protected:
    // pointeur vers l'objet contenant
    game *parent;

    // file des messages à traiter
    std::queue<engine_event> events_queue;

    // traitement d'un message, propre à chaque moteur
    virtual void process_event(engine_event&) = 0;
};
```

Je détaillerai plus loin de principe des objets `engine_event`, messages envoyés d'un module à l'autre. La classe `engine` possède des fonctions virtuelles pures, elle ne peut donc pas être instanciée. Néanmoins, on remarque que pour pouvoir communiquer avec un autre module, il va falloir remonter à l'objet contenant pour récupérer un pointeur sur le module à atteindre, éventuellement downcaster le pointeur pour pouvoir accéder aux fonctions propres à ce module et ensuite spécifier les opérations à effectuer.

 *Les files de messages peuvent potentiellement être accédées par des traitements parallèles, chaque moteur doit donc posséder un mutex à verrouiller à chaque ajout de message et à chaque vidage de la liste.*

J'ai choisi de mettre les modules suivants :

- le moteur de jeu, pour gérer les différents joueurs et toutes les données propres au jeu

- le moteur graphique, chargé de représenter la vision du jeu pour un joueur (le joueur ayant physiquement démarré le jeu)
- le moteur audio, chargé de représenter l'aspect audio du jeu pour un joueur particulier
- le moteur de réseau, chargé d'échanger les données avec les autres joueurs

Le constructeur de la classe engine sert principalement à renseigner le conteneur parent pour chaque module. Ce constructeur devra être appelé à partir des constructeurs des classes filles.

L'exécution proprement dite du jeu consistera simplement en des appels continus à toutes les fonctions frame() des différents moteurs. La condition d'arrêt de cette boucle quasi infinie devra être modifiée par le moteur de jeu. L'architecture possède donc un booléen représentant l'arrêt du programme. Ce booléen devra être protégé par des verrous pour se garantir des accès concurrents éventuels.

```
class game{
// ...
boost::mutex still_running_mutex;
bool still_running;
// ...
};
```


```
void game::run(){
    bool current_still_running = still_running;

    // on crée un verrou pour accéder au mutex
    boost::mutex::scoped_lock l(still_running_mutex);
    l.unlock();

    while (current_still_running){
        n->frame();
        g->frame();
        gfx->frame();
        s->frame();
        l.lock();
        current_still_running = still_running;
        l.unlock();
    }
}
```

Ainsi, pour terminer le jeu, il suffira d'appeler un modificateur du booléen still_running.

```
void game::stoooooop(){
    boost::mutex::scoped_lock l(still_running_mutex);
    still_running = false;
}
```

 *La libération du mutex est faite automatiquement à la fin de la portée du verrou. Le destructeur de boost::mutex::scoped_lock appelle la fonction unlock(), il n'est donc pas nécessaire de le spécifier explicitement.*

Le destructeur de la classe game va appeler les destructeurs de tous les modules instanciés.

```
game::~game(){
    delete n;
    delete g;
    delete gfx;
    delete s;
}
```

II-1 - Communication entre les modules

La communication entre les modules étant vraiment à la base de notre architecture, de très nombreux appels seront effectués pour faire communiquer nos modules, cet aspect doit donc être le plus rapide possible. On pourrait mettre en place un système d'accesseur dans l'architecture pour renvoyer les pointeurs demandés, mais ça restera beaucoup plus lourd que des accès directs. C'est pourquoi j'ai choisi de donner à chaque moteur des pointeurs 'en dur' vers les autres modules. Cela nécessite de connaître à l'avance le nombre de modules à mettre en jeu.

Chaque moteur comporte ainsi 4 pointeurs vers les moteurs de jeu, graphique, audio et réseau. De même, chaque module possède une fonction permettant d'atteindre directement les files de messages des autres modules. Pour cela, il est nécessaire de lier tous les modules entre eux lors de leur création.

```
class game{
// ...
game_engine *g;
graphics_engine *gfx;
network_engine *n;
sound_engine *s;
};
```

```
class engine{
// ...
void send_message_to_graphics(engine_event&);
void send_message_to_network(engine_event&);
void send_message_to_game(engine_event&);
void send_message_to_sound(engine_event&);

game_engine *ge;
network_engine *ne;
graphics_engine *gfxe;
sound_engine *se;
// ...
};
```

Voici le code du constructeur de l'architecture, qui va lier tous les modules entre eux :

```
game::game() {
n = new network_engine(this);
g = new game_engine(this);
gfx = new graphics_engine(this);
s = new sound_engine(this);

// lier tous les modules ensemble
n->attach_game_engine(g);
n->attach_graphics_engine(gfx);
n->attach_sound_engine(s);

g->attach_graphics_engine(gfx);
g->attach_network_engine(n);
g->attach_sound_engine(s);

gfx->attach_game_engine(g);
gfx->attach_network_engine(n);
gfx->attach_sound_engine(s);

s->attach_game_engine(g);
s->attach_graphics_engine(gfx);
s->attach_network_engine(n);
}
```

Les différentes fonctions attach_XXX se contentent de recopier les adresses fournies dans les pointeurs vers les autres modules. Une fois l'architecture créée, tous les modules sont capables de communiquer entre eux, c'est déjà une bonne chose de faite.

Les modules ont donc deux moyens pour communiquer entre eux : s'envoyer un message au travers des files de messages ou bien appeler directement les fonctions des autres modules. L'envoi de messages a l'avantage qu'il renforce la séparation des modules, par contre, pour chaque message qu'on peut envoyer d'un module à l'autre, le module récepteur devra être capable de le lire et de l'interpréter. L'envoi de messages est légèrement plus coûteux puisqu'il va demander l'empilage du message sur la file du récepteur ainsi que l'analyse du message lors de sa réception. Il ne faudra donc pas trop abuser des envois de messages entre les modules pour privilégier les appels directs.

II-1-A - Les messages

Il nous faut une classe pouvant représenter tous les types de messages qui peuvent être échangés entre nos modules. Et pour faire encore plus générique, il faudrait que cette classe puisse aussi être utilisée pour communiquer entre les différentes machines du réseau. Les différents modules et les différentes machines peuvent s'envoyer des informations de diverses natures : des chaînes de caractères, des simples nombres, des vecteurs 3D représentant des positions ...

J'ai choisi d'utiliser une classe comprenant des `std::map` pour différents types de données :

```
class engine_event{
public:
    int type;
    std::map<std::string, std::string> s_data;
    std::map<std::string, int> i_data;
    std::map<std::string, float> f_data;
    std::map<std::string, serializable_vector3df> v_data;
    bool operator==(const engine_event& e);
    template<class Archive>
    void serialize(Archive& ar, const unsigned int);
};
```

`serializable_vector3df` représente simplement une classe fille de la classe de vecteur 3D de flottants d'Irrlicht. J'ai spécialisé cette classe pour la rendre sérialisable, donc pour pouvoir l'envoyer simplement sur le réseau en utilisant `boost::serialize` :

classe de vecteurs sérialisable

```
class serializable_vector3df : public irr::core::vector3df{
public:
    serializable_vector3df() {}
    serializable_vector3df(irr::core::vector3df& v):irr::core::vector3df(v) {}
    template<class Archive>
    void serialize(Archive& ar, const unsigned int){
        ar & X;
        ar & Y;
        ar & Z;
    }
};
```

L'attribut `engine_event::type` permettra de repérer le type de message, de manière à orienter la recherche des informations dans les `std::map`.



Il faudra penser à utiliser des clefs de `std::map` aussi courtes que possibles : en effet, ces clefs seront elles aussi sérialisées et envoyées. Quitte même à les troquer contre des `int` ou d'autres types plus légers.



Suivant les informations que vous choisirez d'envoyer par l'intermédiaire de ces `engine_events`, vous n'utiliserez pas nécessairement les 4 `std::maps` proposées, vous pourrez donc sans problème en supprimer l'une ou l'autre. Les messages s'en trouveront plus légers et plus rapides à échanger, notamment sur le réseau.

II-2 - Ajout d'un module

Dans le cas de l'utilisation d'une liste de modules `std::list<engine*>` il est très simple d'ajouter un module, il suffit d'ajouter un item à cette liste. Par contre dans notre cas, où les modules accèdent directement les uns aux autres, il va falloir rajouter un pointeur membre dans la classe game ainsi que dans la classe engine. De même, il va falloir attacher ce nouveau module à tous ceux déjà existants.

II-3 - Utilisation des modules en mode passif

Toujours dans un souci de réutilisation, certains modules doivent pouvoir être démarrés en mode *passif*. Par exemple, il pourra être intéressant de ne pas faire d'affichage dans le moteur graphique dans le cas d'un serveur de jeu. On gagnera en performances. En effet, le moteur graphique et le moteur audio ne sont que la partie visible du jeu pour un joueur donné. L'idée est de pouvoir lancer le jeu en mode démon, pour qu'il puisse se concentrer sur la gestion du jeu et des autres joueurs.

Pour pouvoir lancer le jeu dans un tel mode, il va falloir paramétrer l'exécution : en passant des paramètres par la ligne de commandes.


J'ai donc créé une petite classe permettant d'analyser les paramètres de la ligne de commandes. Cette classe n'est pas primordiale, elle a plus un rôle d'accessoire qu'autre chose, mais elle est bien pratique tout de même.

```
class parameter_analyser{
public:
    parameter_analyser();
    parameter_analyser(int, char**); // argc, argv
    std::map<std::string, bool> data;
};
```

Le constructeur se contente d'analyser tous les paramètres de la ligne de commandes et de remplir la `std::map` à true lorsque le paramètre a été spécifié. Si les paramètres ne sont pas spécifiés dans la ligne de commandes, les valeurs de data sont mises à false. Par exemple, une exécution avec l'option `--daemon` ou `-d` mettra la valeur associée à la clef "daemon" à true.

Il suffit ensuite de passer une référence sur l'instance de `parameter_analyser` au constructeur `game()` qui analysera les données pour appeler un constructeur de module ou un autre. Ainsi, on peut créer un constructeur de moteur graphique qui accepte un booléen, suivant qu'il faut ou non créer un contexte graphique. Idem pour le moteur audio.

Le jeu démarré en mode passif doit être contrôlé différemment. En effet, vous ne pouvez pas interagir avec la souris ni le clavier, et c'est encore plus vrai si vous associez au mode passif la création d'un nouveau processus de manière à rendre la main sitôt après le démarrage du jeu. C'est la raison pour laquelle il va falloir mettre en place un système de console connaissant quelques commandes de base.

Prenons le cas d'un serveur de jeu, il faut pouvoir communiquer avec le serveur via le réseau. Notre console va donc presque naturellement écouter le réseau, accepter les connexions sur un port particulier, éventuellement faire une authentification par login / mot de passe puis va être en écoute des commandes. Pour sécuriser encore la transmission, on pourrait passer par une couche  **SSL**.

```
class console{
public:
    console();
    ~console();
    void process_command(std::string&);

    // fonction à lancer pour chaque client
    void server_thread_tcp_receive(asio::ip::tcp::socket *);
    // ...
private:
    // attributs propres à la gestion réseau : sockets, contexte réseau ...
```



```
};
```

La réalisation d'une telle console revient ni plus ni moins à implémenter un serveur multiclents. Et pour les mêmes raisons que dans mon précédent article sur une architecture de serveur multithreads, j'ai choisi d'utiliser un contexte multithreadé. Nous pouvons ainsi simplement gérer plusieurs connexions simultanées. Pour des raisons de simplicité et surtout de robustesse, j'ai choisi d'utiliser la bibliothèque asio pour la gestion du réseau.

II-3-A - Implémentation du système de console

Il va falloir rattacher la console quelque part dans notre architecture. J'ai choisi de la placer en attribut du moteur de jeu. Elle sera donc créée lors de son instanciation.

La déclaration complète de la console

```
class console{
public:
    console(game_engine*);
    ~console();
    void process_command(CEGUI::String&);
    void process_command(std::string&);

    void handle_accept_tcp(const asio::error_code&, asio::ip::tcp::socket*);
    void server_thread_tcp_receive(asio::ip::tcp::socket *);
private:
    game_engine *parent;
    asio::ip::tcp::socket *s;
    asio::ip::tcp::acceptor *tcp_acceptor;
    asio::io_service io;
};
```

Le constructeur

```
console::console(game_engine* g):parent(g) {
    // création de la socket d'écoute
    s = new asio::ip::tcp::socket(io);
    tcp_acceptor = new asio::ip::tcp::acceptor(io, asio::ip::tcp::endpoint(asio::ip::tcp::v4(), 12345));

    asio::error_code e;
    tcp_acceptor->async_accept(*s, boost::bind(&console::handle_accept_tcp, this, e, s));

    asio::thread t(boost::bind(&asio::io_service::run, &io));

    if (e.value() != 0){
        std::cerr << e.message() << std::endl;
    }
}
```

Le constructeur crée une socket qui va écouter les connexions TCP sur le port 12345. A chaque connexion entrante, il va appeler `console::handle_accept_tcp`. Cet appel est effectué en mode asynchrone. Chaque appel à `handle_accept_tcp` va démarrer un thread pour s'occuper de cette connexion :

```
void console::handle_accept_tcp(const asio::error_code& e, asio::ip::tcp::socket* socket) {
    if (e.value() != 0){
        std::cerr << e.message() << std::endl;
        return;
    }

    // on démarre le thread du client
    asio::thread t(boost::bind(&console::server_thread_tcp_receive, this, socket));

    // on réarme l'appel asynchrone avec une nouvelle socket
    asio::ip::tcp::socket *s = new asio::ip::tcp::socket(io);
    asio::error_code ec;
    tcp_acceptor->async_accept(*s, boost::bind(&console::handle_accept_tcp, this, ec, s));
}
```



Il ne nous reste plus qu'à écouter les instructions envoyées par le client :

```
void console::server_thread_tcp_receive(asio::ip::tcp::socket *s){
    // authentification éventuelle

    // on attend les instructions de manière bloquante
    for (;;) {
        boost::array<char, 1024> buf;
        asio::error_code error;

        size_t len = s->read_some(asio::buffer(buf), error);

        if (error == asio::error::eof)
            break; // la connexion a été interrompue
        else if (error)
            break;

        process_command(std::string(buf.data()));
    }
    s->close();
    delete s;
}
```

Le protocole est extrêmement simple : le client envoie des instructions et le serveur lui renvoie le résultat de la réception, tout ça sous forme de chaînes de caractères.

Et à chaque réception d'une instruction, on appelle le traitement de cette instruction. C'est ici qu'on doit implémenter au moins la commande de fermeture.

```
void console::process_command(std::string& c){
    if (c == "quit")
        parent->parent->stooooooooop();

    // autres instructions existantes
}
```

II-3-B - Un client pour la console

J'ai écrit un client basique pour pouvoir accéder à la console à distance sur un tel serveur de jeu. Il s'agit simplement d'une fenêtre de saisie de commande :



Le code source de ce client est également disponible dans l'archive en téléchargement à la fin de cet article.

III - Le moteur graphique

III-1 - Vue d'ensemble

Le moteur graphique va gérer tout l'affichage, qu'il s'agisse des objets 3D ou plus simplement des menus. Il devra implémenter toutes les fonctions virtuelles pures de la classe mère engine. C'est également le moteur graphique qui va gérer les entrées clavier et souris.

```
class graphics_engine : public engine,
    irr::IEventReceiver
{
public:
    graphics_engine(game*, bool); // objet contenant et mode passif
    ~graphics_engine();

    bool OnEvent(irr::SEvent);

    void frame();

    inline gui_system *get_gui(){return gui;}
private:
    gui_system *gui;

    irr::video::IVideoDriver *driver;
    CEGUI::System *sys;
    irr::IrrlichtDevice *idevice;
    void process_event(engine_event&);

    scene_graph *sg;
    bool console_visible;
    // ...
};
```

Les positions de tous les objets du jeu sont stockées dans le graphe de scène d'Irrlicht.

III-2 - Le graphe de scène

J'ai encapsulé le graphe de scène fourni par Irrlicht dans une classe plus spécialisée, permettant de gérer les accès concurrents, et surtout qui va propager les modifications aux autres joueurs.

```
class scene_graph{
public:
    scene_graph(graphics_engine*);
    ~scene_graph();
    irr::scene::ISceneManager *smgr;
    void add_node(int, irr::core::vector3df&, irr::core::vector3df&);
    void move_node(int, irr::core::vector3df&, irr::core::vector3df&);
    void remove_node(int);
    void render();

    void set_observer(network_interface*);
    void remove_observer();

    boost::mutex smgr_mutex;
private:
    graphics_engine *parent;
    network_interface *o;
};
```

Il conviendra d'attacher un observateur réseau à cet objet.

Les fonctions `add_node`, `move_node` et `remove_node` doivent être précisées pour spécifier un identifiant de machine réseau à laquelle il n'est pas nécessaire de transmettre la modification. Il suffit de rajouter un paramètre dans ces fonctions. J'aborderai ce point dans la partie relative au moteur de réseau.

Il faudra aussi créer au démarrage du jeu une collection de meshes et leur associer un identifiant, qui servira à appeler la fonction `add_node`.

III-3 - Constructeur du moteur graphique

Le constructeur va créer le contexte d'affichage, en tenant compte d'un éventuel mode passif, il va aussi créer le contexte CEGUI pour pouvoir afficher des menus.

```
graphics_engine::graphics_engine(game *g, bool passive):engine(g)
{
    // on crée le contexte OpenGL
    if (!passive)

        idevice = irr::createDevice(irr::video::EDT_OPENGL, irr::core::dimension2d<irr::s32>(800, 600), 32, false, false,
    else
        idevice = irr::createDevice(irr::video::EDT_NULL);

    driver = idevice->getVideoDriver();
    sg->smgr = idevice->getSceneManager();

    // on masque le curseur
    idevice->getCursorControl()->setVisible(false);

    // on crée le contexte CEGUI
    try{
        CEGUI::IrrlichtRenderer *myRenderer = new CEGUI::IrrlichtRenderer(idevice);
        new CEGUI::System(myRenderer);
    }catch(CEGUI::Exception &e){
        shared::get()->log.error(std::cerr << e.getMessage() << std::endl);
    }catch(...){
        shared::get()->log.error(std::cerr << "unknown exception" << std::endl);
    }

    // on définit le niveau de log de CEGUI
    CEGUI::Logger::getSingleton().setLoggingLevel((CEGUI::LoggingLevel)3);

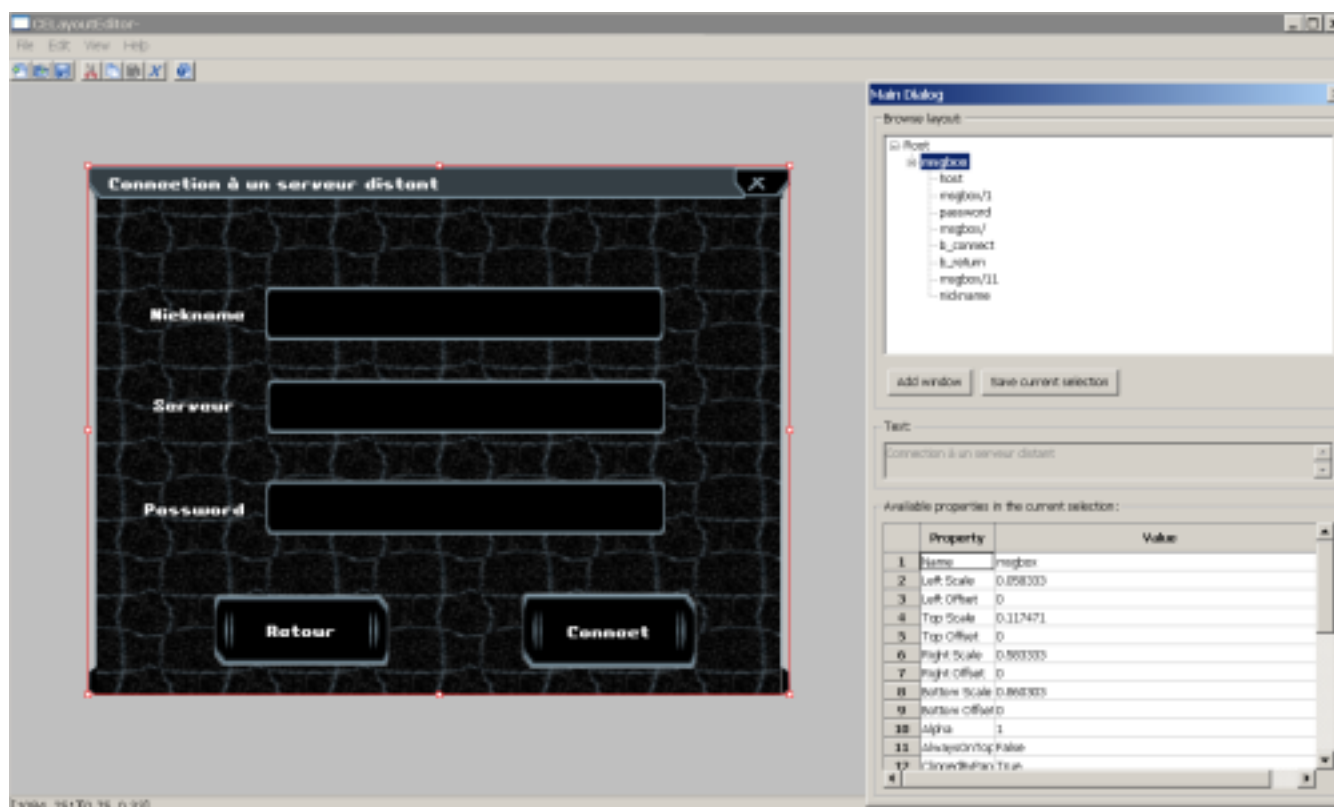
    // on charge le thème graphique des menus
    try{
        CEGUI::SchemeManager::getSingleton().loadScheme("../data/gui/schemes/TaharezLook.scheme");
    }catch(CEGUI::Exception &e){
        std::cout << e.getMessage() << std::endl;
    }

    // on charge une police d'écriture
    CEGUI::FontManager::getSingleton().createFont("../data/gui/fonts/Commonwealth-10.font");
    CEGUI::System::getSingleton().setDefaultMouseCursor("TaharezLook", "MouseArrow");

    gui = new gui_system();
    gui->set_parent(this);
}
```

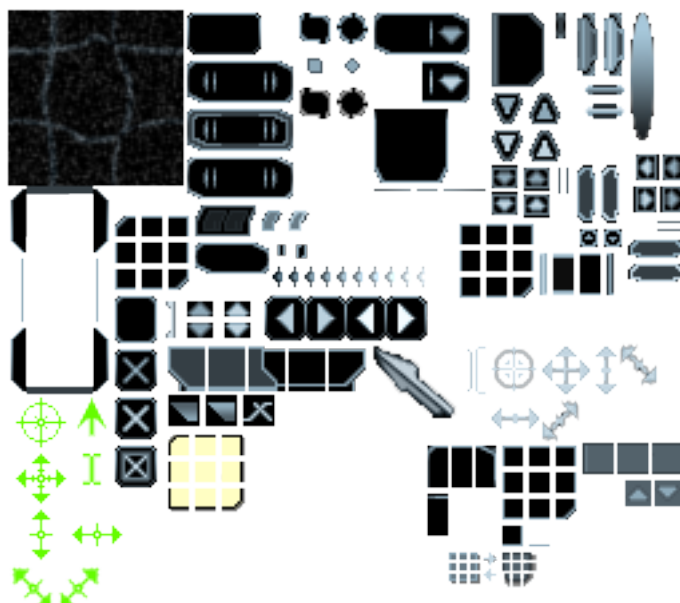
III-3-A - Un petit mot sur CEGUI

CEGUI est une bibliothèque permettant de gérer très simplement des interfaces graphiques dans des contextes OpenGL, DirectX, Irrlicht et Ogre3D. Chaque interface est décrite sous forme de fichier XML et il est possible de dessiner très simplement des écrans en utilisant des outils comme le **CEGUI Layout Editor**



Pour afficher une interface, il suffit de charger le fichier XML correspondant. Les différents événements peuvent être redirigés vers des fonctions LUA ou des fonctions C++.

Le thème graphique d'une interface est aisément modifiable : il suffit d'éditer le fichier image représentant tous les objets graphiques ainsi que les fichiers de configuration XML associés.



Le thème graphique 'Taharez'

L'affichage d'une interface se fait de la manière suivante :

```
void gui_system::display_interface(){
    CEGUI::WindowManager& winMgr = CEGUI::WindowManager::getSingleton ();
    winMgr.destroyAllWindows();


    // ajout éventuel d'options sur le rendu de l'interface

    // chargement du fichier XML
    background->addChildWindow (winMgr.loadWindowLayout (layout_path+"menu.layout", "root/"));

    // redirection des évènements
    CEGUI::WindowManager::getSingleton().getWindow("root/b_quit")->
        subscribeEvent(CEGUI::PushButton::EventClicked, CEGUI::Event::Subscriber(&gui_system::close, this));

    CEGUI::System::getSingleton ().setGUISheet (background);
}
```

Les redirections des évènements et les noms des objets graphiques sont différents d'une interface à l'autre, il y aura donc autant de fonctions que d'écrans à afficher.

 **Les noms des objets graphiques doivent être spécifiés dans le fichier XML. Ces noms sont repris dans le code C++. Dans le cas où un objet n'a plus le même nom, une exception CEGUI est levée. Il faudra donc attraper ces exceptions.**

III-4 - A chaque frame du moteur graphique

Le moteur graphique a un travail très simple : il se contente d'afficher le graphe de scène ainsi que l'interface CEGUI. Ces deux affichages doivent être protégés des éventuels accès concurrents :

```
void graphics_engine::frame(){
    // on vérifie si l'utilisateur n'a pas fermé la fenêtre
    if (!driver || !device->run())
        parent->stooooooooop();


    driver->beginScene(true, true, irr::video::SColor(0,100,100,100));

    // on dessine le graphe de scène
    sg->render();

    // on appelle l'affichage de l'interface
    gui->render();
    driver->endScene();
}
```

Le code d'affichage de l'interface est encore plus simple :

```
void gui_system::render(){
    boost::mutex::scoped_lock lock(shared::get()->mutex_gui);
    CEGUI::System::getSingleton().renderGUI();
}
```

 **L'interface est aussi protégée des accès concurrents par un mutex, stocké dans notre singleton.**

III-5 - Les entrées clavier / souris

Irrlicht se charge d'écouter tous les évènements clavier / souris. Pour les écouter, il suffit de créer un irr::IEventReceiver. Plusieurs possibilités pour ça. La première solution est de faire hériter le moteur graphique de irr::IEventReceiver. irr::IEventReceiver possède une méthode virtuelle pure, il nous faut donc l'implémenter. Irrlicht sera le seul point d'entrée des évènements, c'est donc lui qui va envoyer les évènements à CEGUI.

Réception des évènements

```

bool graphics_engine::OnEvent(irr::SEvent e) {
    if (!idevice)
        return false;

    switch (e.EventType) {
    case irr::EET_MOUSE_INPUT_EVENT:
        // envoyer la position de la souris à CEGUI
        CEGUI::System::getSingleton().injectMousePosition((float)e.MouseInput.X, (float)e.MouseInput.Y);
        // envoyer les clics à CEGUI
        switch (e.MouseInput.Event) {
            case irr::EMIE_LMOUSE_PRESSED_DOWN :
                CEGUI::System::getSingleton().injectMouseButtonDown(CEGUI::LeftButton);
                break;
            case irr::EMIE_LMOUSE_LEFT_UP :
                CEGUI::System::getSingleton().injectMouseButtonUp(CEGUI::LeftButton);
                break;
            case irr::EMIE_MOUSE_WHEEL :
                CEGUI::System::getSingleton().injectMouseWheelChange(e.MouseInput.Wheel);
                break;
            default:
                break;
        }
        return true;
    case irr::EET_KEY_INPUT_EVENT:
        if (e.KeyInput.PressedDown) {
            if (e.KeyInput.Key == irr::KEY_F2) {
                // afficher/masquer la console
                gui->toggle_console();
                break;
            }

            // injecter certaines touches
            switch(e.KeyInput.Key) {
                case irr::KEY_RETURN: CEGUI::System::getSingleton().injectKeyDown(CEGUI::Key::Return); break;
                case irr::KEY_BACK: CEGUI::System::getSingleton().injectKeyDown(CEGUI::Key::Backspace); break;
                case irr::KEY_TAB: CEGUI::System::getSingleton().injectKeyDown(CEGUI::Key::Tab); break;
                default:
                    CEGUI::System::getSingleton().injectChar(e.KeyInput.Char);
                    break;
            }
        }
        break;
    default:
        return false;
    }

    return false;
}

```

On remarquera l'appel à `gui_system::toggle_console()` pour afficher / masquer la console. En effet, la console a exactement le même rôle, qu'elle soit accessible par le réseau ou directement au clavier. Il suffit de rediriger l'évènement de validation de l'interface de la console vers la fonction de traitement d'instruction.

Au fur et à mesure que le jeu se complexifiera, la fonction `OnEvent` va très vite devenir très lourde et surtout une abominable imbrication de `switch` et de `if`. C'est pourquoi on va se tourner vers l'autre solution : nous allons créer une classe virtuelle d'écouteur d'évènement et nous allons instancier des instances de classes filles qui redéfinissent le traitement des évènements. Ainsi, à chaque changement d'état du jeu, nous n'aurons qu'à récupérer un pointeur vers l'écouteur d'évènement à utiliser. Il faudra donc instancier tous les écouteurs au démarrage du moteur graphique et les stocker dans un conteneur les associant au numéro de l'état du jeu : par exemple une

```
std::map< int, boost::shared_ptr < event_handler > >
```

Les changements à apporter sont mineurs :

- Il ne faut plus faire hériter le moteur graphique de `irr::IEventReceiver`

- Il faut créer une classe event_handler :

Classe mère des écouteurs d'évènements

```
class event_handler : public irr::IEventReceiver{
public:
    event_handler(graphics_engine* ge){parent = ge;}
    virtual ~event_handler(){}
    virtual bool OnEvent(irr::SEvent)=0;

    inline graphics_engine* get_parent(){return parent;}
protected:
    graphics_engine *parent;
};
```

- créer des classes filles :

La classe fille de gestion des menus

```
class menu_event_handler : public event_handler{
public:
    menu_event_handler(graphics_engine*);
    bool OnEvent(irr::SEvent){
        // tout le code de gestion d'évènements
    }
};
```


- instancier tous les écouteurs d'évènements et les stocker dans un conteneur dédié

```
l_ah[ON_MENUS] = boost::shared_ptr<event_handler>(new menu_event_handler(this));
l_ah[ON_GAME] = boost::shared_ptr<event_handler>(new game_event_handler(this));
```

- créer une petite fonction pour changer d'écouteur d'évènements :

```
void graphics_engine::set_state(int s){
    state = s;
    iddevice->setEventReceiver(l_ah[s].get());
}
```

Ainsi les gestions des évènements sont plus claires et aussi plus rapides puisqu'il y a moins de switch et de if à résoudre. Il n'a plus qu'à créer les différentes classes relatives aux différents états du jeu.

 On peut juste signaler que les écouteurs d'évènements doivent être contenus dans le moteur graphique, qu'ils ne doivent pas exister en mode passif.

IV - Le moteur audio

Le moteur audio est le plus simple de tous les modules. Je me suis appuyé sur IrrKlang, il ne reste donc plus qu'à implémenter une gestion basique des sons. IrrKlang est encore en plein développement, certaines des fonctionnalités finales ne sont donc pas encore disponibles, notamment la détection des fins des sons, je n'ai donc implémenté aucune fonctionnalité touchant à cet aspect.

Le moteur audio est le seul à posséder les chemins physiques vers les ressources audio (wav, ogg ...). Tous les modules qui souhaitent lancer la lecture d'un son vont fournir le nom logique du son à jouer. Le moteur audio possèdera une table de correspondance entre les noms logiques et les noms physiques.

```
class sound_engine : public engine{
public:
    sound_engine(game*, bool);
    ~sound_engine();
    void frame(){}

    inline void play_ambience(std::string& s){s_engine->play2D(sound_names[s].c_str());}
    inline void play_spatial(std::string& s, irr::core::vector3df
v){s_engine->play3D(sound_names[s].c_str(), v);}

    std::map<std::string, std::string> sound_names;

    void process_event(engine_event&);
protected:
    void get_config_data();
    irr::audio::ISoundEngine *s_engine;
    std::vector<std::string> playlist;
};
```

Le constructeur du moteur audio est similaire à celui du moteur graphique : les options de création de l'objet principal vont dépendre du mode passif ou actif.

Constructeur

```
sound_engine::sound_engine(game* g, bool passif):engine(g){
    passive_mode = passif;
    if (passive_mode)
        s_engine = irr::audio::createIrrKlangDevice(irr::audio::ESOD_NULL);
    else
        s_engine = irr::audio::createIrrKlangDevice();

    // configuration du moteur audio
    get_config_data();
}
```

Le moteur audio devra être capable d'associer un nom physique aux noms logiques, au démarrage il doit donc créer sa table de correspondance sound_names. Il peut la créer à partir d'un fichier XML, d'un fichier INI ...

De la même manière, le moteur audio doit repérer toutes les musiques qu'il a à sa disposition pour l'élaboration d'une playlist. Et à chaque fois qu'une musique est terminée, il enchaînera sur la musique suivante. La création de la playlist peut par exemple se faire en analysant tous les fichiers présents dans un répertoire donné :

```
boost::filesystem::path dir("./data/sound/playlist");
boost::filesystem::directory_iterator i(dir), end;





for (; i!=end; i++){
    if (boost::filesystem::is_directory("./data/sound/playlist/"+i->leaf()))
        continue;

    playlist.push_back("./data/sound/playlist/"+i->leaf());
}
```

Les sons 3D pourront également être mis directement dans le graphe de scène, de manière à regrouper en un seul endroit tous les sons en cours de lecture. Cette solution nous permet de largement simplifier la gestion des sons. Cela permet également de ne plus avoir à se préoccuper de la perception d'un son par divers joueurs du réseau. Tout sera géré en temps qu'objets du graphe de scène. Un bon réglage des distances maximales de perceptions permettra de limiter le nombre de sons à jouer simultanément, économisant ainsi des ressources CPU.

V - Le moteur de réseau

Le moteur de réseau est un gros morceau de l'architecture, il doit permettre à plusieurs machines de communiquer, que ça soit sur un réseau local ou bien sur internet. J'ai choisi d'utiliser l'organisation suivante :

- Sur un réseau  **LAN**, les paquets importants sont envoyés via  **TCP** entre le serveur et les clients. Les paquets de moindre importance sont envoyés en  **UDP**  **multicast**.
- Sur un réseau de plus grande ampleur comme internet, tous les messages sont envoyés entre le serveur et un client. J'utilise toujours deux connexions (TCP et UDP) selon que les messages à envoyer peuvent être égarés ou non.

Il va nous falloir distinguer dès le début la notion de machine réseau de la notion de joueur réseau. En effet, plusieurs joueurs réseau peuvent jouer sur une même machine réseau. C'est notamment le cas des personnages non joueurs qui peuvent être gérés par des clients. Cette distinction nous permettra de répartir la gestion des IA entre les différentes machines, pour utiliser la puissance des machines les plus performantes. J'ai choisi de déterminer la performance des machines réseau en regardant le nombre d'images qu'elles sont capables d'afficher par seconde.

Chaque machine possèdera une liste des autres machines auxquelles elle est reliée. Elle possèdera aussi les objets systèmes pour communiquer avec ces machines.

Lors d'un jeu en LAN, chaque joueur peut directement informer tout le réseau via la connexion multicast. Dans le cas d'un jeu sur internet, chaque machine souhaitant communiquer avec tous les joueurs devra envoyer le message au serveur, qui relayera ce message pour toutes les machines connectées.

Voici la classe utilisée pour représenter une machine distante :

```
class network_machine{
public:
    network_machine();
    network_machine(network_engine*, asio::ip::tcp::socket *);
    ~network_machine();
    void attach_player(network_player*);
    void remove_player(network_player*);

    void TCP_async_receive(const asio::error_code&, size_t);
    void TCP_send(engine_event&);
    void UDP_send(engine_event&);

    asio::ip::tcp::socket *s_tcp;
    asio::ip::udp::socket *s_udp;

    // ...

private:
    // liste des joueurs associés à la machine réseau
    std::list<network_player*> l_players;
    std::vector<char> network_buffer;

    // ...

    network_engine *parent;
};
```

Les machines réseau apparaissent comme des membres du moteur de réseau. Chaque machine réseau peut être atteinte via TCP ou UDP. Par contre, la réception des données venant de cette machine distante est légèrement différente : la réception TCP se fait simplement dans la machine réseau alors que la réception UDP se fait par la socket en écoute du moteur réseau.

V-1 - La sérialisation des messages

J'ai choisi de m'appuyer sur le système de sérialisation de boost pour envoyer les messages des moteurs sur le réseau. Ce système permet de convertir n'importe quel objet *sérialisable* en une structure linéaire d'octets (entre autres) facile à envoyer sur une socket. De la même manière, le système est capable de reconstituer l'objet de départ.

Il nous faut donc déclarer notre classe de messages comme *sérialisable*, pour cela il nous suffit d'y implémenter une fonction `serialize()` :

```
template<class Archive>
void serialize(Archive& ar, const unsigned int) {
    ar & type;
    ar & s_data;
    ar & i_data;
    ar & f_data;
    ar & v_data;
}
```

La sérialisation d'un `engine_event` va automatiquement appeler la sérialisation des objets contenus, et donc des objets `serializable_vector3df` définis plus haut. Maintenant que nos messages sont sérialisables, voyons comment ils sont envoyés sur le réseau.

V-2 - Envoi / réception des messages

Pour envoyer un message sur une socket, il faut créer le buffer qui va stocker notre message sérialisé. La sérialisation proprement dite est réalisée par l'opérateur `<<`. Voici le code de la fonction d'envoi de message sur la socket TCP de la machine réseau.

```
void network_machine::TCP_send(engine_event& e) {
    std::ostringstream archive_stream;
    boost::archive::text_oarchive archive(archive_stream);
    archive << e;
    const std::string &outbound_data = archive_stream.str();

    s_tcp->send(asio::buffer(outbound_data));
}
```

La fonction de réception est exactement la duale. J'ai choisi d'utiliser une réception asynchrone, ainsi la fonction `TCP_async_receive` est appelée à chaque réception.

```
void network_machine::TCP_async_receive(const asio::error_code& e, size_t bytes_received) {
    // test sur la réception
    // ...

    // désérialisation
    std::string str_data(&network_buffer[0], network_buffer.size());
    std::istringstream archive_stream(str_data);
    boost::archive::text_iarchive archive(archive_stream);

    engine_event ne;
    archive >> ne;

    // on ajoute le message sur la pile des messages à traiter
    parent->push_received_event(ne);
}
```

V-3 - Les sockets pour atteindre une machine réseau

De manière à utiliser les appels asynchrones pour la réception en mode TCP, il nous faut le spécifier lors de la création de la socket :

```
network_machine::network_machine(network_engine* p, asio::ip::tcp::socket *so, int
i):s_tcp(so), parent(p), id(i){
// création de la socket UDP
udp_ep = new asio::ip::udp::endpoint( so->remote_endpoint().address(), parent->port_udp_reception);
s_udp = new asio::ip::udp::socket(io, *udp_ep);

address = so->remote_endpoint().address().to_string();

// mise en place de l'appel asynchrone
so->async_receive( asio::buffer(network_buffer),
boost::bind(&network_machine::TCP_async_receive, this,
asio::placeholders::error,
asio::placeholders::bytes_transferred)
);

asio::thread(boost::bind(&asio::io_service::run, &io));
}
```

Et de la même manière, le destructeur de la network_machine se contentera de détruire les sockets créées ainsi que tous les joueurs qui y sont éventuellement rattachés.

Une machine réseau doit être créée pour communiquer avec tous les acteurs du réseau, quel que soit le mode de connection (LAN, internet). Quand un client souhaite se connecter directement à un serveur, la création de la machine réseau est légèrement différente :

```
void network_engine::client_connects_to_server(std::string& host, std::string& nick, std::string&
pass){
network_machine *serveur = new network_machine(this, get_next_machine_id());
l_machines.push_back(serveur);

if (serveur->connect(host, nick, pass) == 1){
// la connexion a échoué, on supprime la machine réseau fraîchement créée
std::vector<network_machine*>::iterator i = l_machines.end();
i--;
l_machines.erase(i);
get_graphics_engine()->get_gui()->ShowErrorMessage(std::string("Impossible de se connecter à ") + host, gui_system);
} else
// on affiche l'écran suivant
gfxe->get_gui()->display_select_players();
}
```

La fonction connect(host, nick, pass) va essayer d'établir une connexion TCP sur l'adresse donnée, puis va commencer le protocole en envoyant le nom choisi et un mot de passe pour la partie. En cas d'échec (adresse injoignable ?), on affiche un message d'erreur.

V-4 - Spécification réseau local ou réseau Internet

Les traitements à effectuer pour propager une information sur le réseau en LAN et sur Internet ne seront pas tout à fait les mêmes. J'ai créé une classe network_interface qui se spécialise en LAN_network_interface et en Internet_network_interface selon les choix réalisés par le client dans l'interface graphique.

```
class network_interface{
public:
network_interface(network_engine* n):parent(n){};

// fonctions de l'observateur
void send_add(int, serializable_vector3df&, serializable_vector3df&);
void send_move(int, serializable_vector3df&, serializable_vector3df&);
void send_rem(int);

// utilisé en mode client et serveur
virtual void send_to_all_gameUDP(engine_event &)=0;

// utilisé seulement en mode client
void send_to_serverTCP(engine_event &);
}
```

```

// utilisé seulement en mode serveur
void send_to_all_gameTCP(engine_event &);

// pour repérer quel mode a été instancié
enum{
    LAN,
    INTERNET,
};

inline int get_type(){return type;}

protected:
    network_engine *parent;
    int type;
};

```

avec

```

class LAN_network_interface : public network_interface{
public:
    LAN_network_interface(network_engine*);
    ~LAN_network_interface();

    void send_to_all_gameUDP(engine_event &);
};

```

La définition de la classe Internet_network_interface est similaire. Seule la fonction send_to_all_gameUDP va changer : on utilisera le mode multicast en LAN et on bouclera sur toutes les machines réseaux en mode internet.

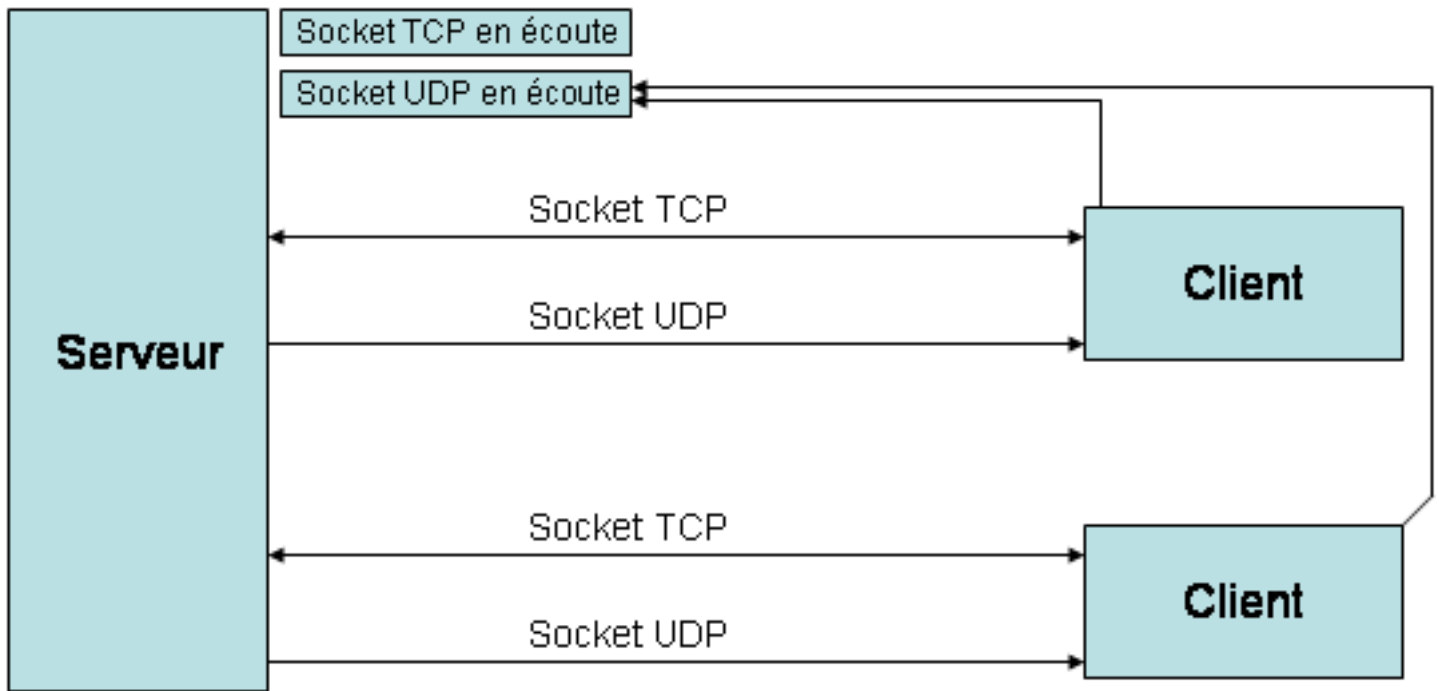
Un point important concerne l'aspect observateur développé dans la classe d'interface réseau. En effet, l'interface réseau doit retranscrire chaque modification du graphe de scène à toutes les machines connectées. Ces fonctions d'observateur sont appelées à chaque fois que le graphe de scène évolue. Chacune de ces fonctions va déclencher l'envoi d'un message en UDP ou en TCP selon l'importance du message. Ainsi la création et la destruction d'objets du graphe de scène sont envoyées en TCP. Ces messages seront beaucoup plus rares que les messages de déplacement, qui sont eux, envoyés par UDP.

Une fois les fonctions de base du moteur de réseau écrites, il reste à organiser leurs appels : il va falloir écrire un système permettant d'instancier les bons objets en fonction des choix de l'utilisateur, notamment en suivant les clics qu'il fait dans l'interface graphique. Ces choix nous permettront d'instancier la bonne classe.

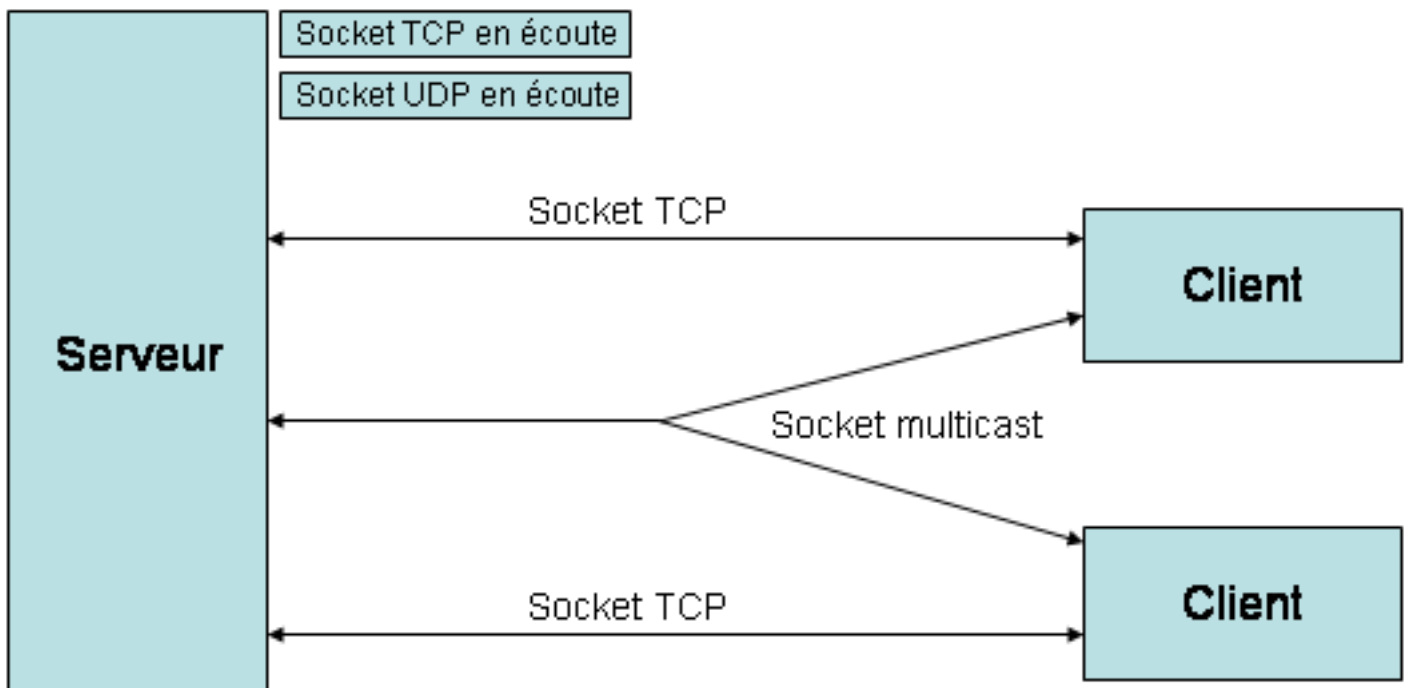
Un serveur aura toujours une socket TCP en écoute, pour repérer les éventuelles machines qui souhaitent se connecter et participer au jeu. A chaque nouvelle machine, le serveur va créer un thread et ajouter la machine dans sa liste.

Un client ne sera jamais relié qu'à une seule machine : le serveur, et tout se fera exactement comme si tous les autres joueurs jouaient physiquement sur le serveur puisque celui-ci relayera tous les messages.

Voici deux schémas qui valent mieux qu'un long discours :



Organisation du réseau en mode Internet



Organisation du réseau en mode LAN



Les sockets TCP sont à double emploi : elles nous permettent de transmettre des messages importants et aussi elles nous permettent de repérer les déconnexions éventuelles des joueurs.

V-5 - Réception des messages UDP

Tous les messages TCP arrivent dans leurs machines réseaux respectives, par contre, les messages UDP arrivent tous sur la même socket, qui est la socket générale d'écoute UDP du moteur réseau. En effet, en créant autant de

sockets UDP d'écoute que de machines réseau, il aurait fallu ouvrir autant de ports et donc potentiellement rerouter autant de ports sur des éventuels routeurs.

UDP étant un mode non connecté, nous pouvons économiser des sockets. Par contre, et c'est l'effet pervers du mode non connecté, tout le monde peut envoyer des messages sur cette socket. Il suffirait de connaître le numéro du port pour pouvoir accéder à notre application. C'est pourquoi, j'ai choisi de n'accepter aucun message UDP qui ne vienne pas d'une machine réseau déjà enregistrée. En effet, il n'y aura d'échanges UDP qu'une fois que la machine réseau sera enregistrée en TCP, nous pouvons donc créer une table des adresses IP connectées. Chaque machine réseau possèdera connaîtra son adresse IP, il suffira de parcourir les machines connectées et de comparer leur IP pour savoir si on peut accepter les messages d'une IP donnée.

Mise en place de l'écoute UDP générale

```
asio::ip::udp::endpoint listen_ep(asio::ip::udp::v4(), port_udp_reception);
s_udp_in = new asio::ip::udp::socket (io, listen_ep.protocol());

// mise en place de l'appel asynchrone
s_udp_in->async_receive_from( asio::buffer(network_buffer),
    udp_remote_endpoint,
    boost::bind(&network_engine::UDP_async_read, this,
        asio::placeholders::error,
        asio::placeholders::bytes_transferred)
    );
```

Chaque appel va stocker l'adresse et le port qui ont émis le message dans `udp_remote_endpoint`. Il nous suffira de le tester dans l'appel asynchrone :

```
void network_engine::UDP_async_read(const asio::error_code& e , size_t bytes_received){
    // on vérifie si l'adresse est enregistrée
    if (!is_address_registered(udp_remote_endpoint.address().to_string()))
        return;

    // on déséréalise le message
    std::string str_data(&network_buffer[0], network_buffer.size());
    std::istream archive_stream(str_data);
    boost::archive::text_iarchive archive(archive_stream);

    engine_event ne;
    archive >> ne;

    // on ajoute un identifiant pour repérer l'expéditeur
    ne.i_data["FROM"] = get_machine_from_address(udp_remote_endpoint.address().to_string())->id;

    // on ajoute le message sur la pile des messages à traiter
    push_received_event(ne);
}
```

La fonction `is_address_registered` se contente de parcourir les machines déjà enregistrées pour en chercher une qui possède l'adresse à tester.

On remarque l'ajout d'un identifiant pour l'expéditeur : cet identifiant est généré à chaque fois qu'une nouvelle machine va se connecter. Cet identifiant ne doit **jamais** circuler sur le réseau, il est propre à chaque exécution du jeu. En effet, un client ne sera relié qu'à un serveur alors que le serveur sera relié à plusieurs clients, l'identifiant "1" peut donc exister plusieurs fois mais avec des correspondances différentes.

Chaque machine physique possède une socket UDP en écoute sur un port connu à l'avance. Il est donc délicat de masquer plusieurs machines derrière un sous réseau. Leur adresse publique est identique. Pour pallier ce manque, on pourrait mettre en place un système où chaque machine physique cherche elle-même un port libre puis le communique au serveur. Mais ce port pourrait changer à chaque exécution et les routages NAT devraient être régulièrement vérifiés.

V-6 - Recherche de parties disponibles

Chaque client souhaitant participer à une partie doit d'abord contacter le serveur qui héberge la partie. Pour cela, il doit en connaître l'adresse. Sur Internet, nous avons deux possibilités : chaque serveur proposant une partie s'enregistre auprès d'une entité tierce, accessible par tout le monde et dont l'adresse est connue ; ou bien chaque client doit connaître l'adresse du serveur (adresse transmise par email, ou par tout autre moyen). En réseau local, nous pouvons nous appuyer sur les adresses multicast : si tout le monde écoute cette même adresse, les clients pourront être informés de toutes les parties créées sur le réseau local.

Envoi de la proposition de partie, par le serveur

```
void network_engine::thread_send_multicast(engine_event gp){
    boost::mutex::scoped_lock l(state_mutex);
    int current_state = state;
    l.unlock();

    asio::deadline_timer timer(io, boost::posix_time::seconds(2));

    // on envoie la proposition tant que le jeu n'a pas démarré
    while (current_state == STATE_WAITING_PLAYERS){
        send_eventUDP(gp, s_multicast);

        // on envoie le message toutes les 2 secondes
        timer.expires_at(timer.expires_at() + boost::posix_time::seconds(2));
        timer.wait();

        l.lock();
        current_state = state;
        l.unlock();
    }
}
```

Chaque machine écoute l'adresse multicast et stocke les propositions de parties dans un conteneur dédié :

```
void network_engine::multicast_receive(const asio::error_code&, std::size_t){
    // désérialisation du message
    std::string str_data(&network_buffer[0], network_buffer.size());
    std::istringstream archive_stream(str_data);
    boost::archive::text_iarchive archive(archive_stream);

    engine_event e;
    archive >> e;

    switch(state){
        case engine::STATE_WAITING_PLAYERS:
            if (e.type == engine_event::GAME_PROP){
                // add the game proposition to the waiting list
                boost::mutex::scoped_lock l(propositions_mutex);

                // on regarde si le message a déjà été reçu dans les 3 dernières secondes
                std::vector<engine_event>::iterator i = std::find(
                    received_propositions.begin(),
                    received_propositions.end(),
                    e);
                if (i != received_propositions.end())
                    received_propositions.push_back(e);

                l.unlock();
                break;
            }
            break;

        default:
            // ne doit jamais arriver
            break;
    }
}
```

Chaque 3 secondes, on vide le conteneur dédié pour afficher toutes les parties repérées au client :

```
void network_engine::listen_multi_3seconds_udp() {
    boost::mutex::scoped_lock state_lock(state_mutex);
    state = STATE_WAITING_PLAYERS;
    state_lock.unlock();

    asio::deadline_timer timer(io, boost::posix_time::seconds(3));
    int current_state = state;

    while (current_state == STATE_WAITING_PLAYERS) {

        std::vector<engine_event> prop;

        timer.expires_at(timer.expires_at() + boost::posix_time::seconds(3));
        timer.wait();

        // on récupère toutes les parties repérées
        boost::mutex::scoped_lock l(propositions_mutex);
        prop = received_propositions;
        received_propositions.clear();
        l.unlock();

        boost::mutex::scoped_lock state_lock(state_mutex);
        current_state = STATE_WAITING_PLAYERS;
        state_lock.unlock();

        // affichage des parties dans l'interface graphique
        // ...
    }
}
```

V-7 - Propagation des messages

La propagation des messages ne se fera pas de la même manière en mode client et en mode serveur : un serveur doit transférer les messages qu'il reçoit aux autres machines connectées. Ça peut paraître bête à dire, mais le serveur ne doit pas retransmettre un message à la machine réseau qui vient de lui envoyer. On va donc devoir tenir compte des expéditeurs des messages en mode serveur.

Le mode client est moins problématique : on se contente de mettre à jour les données locales en fonction des messages reçus, rien n'est transmis.

Dans tous les cas, client ou serveur, LAN ou Internet, tous les graphes de scène doivent être identiques, sur toutes les machines du réseau. Sans ça, la cohérence entre les joueurs ne peut être assurée.

VI - Moteur de jeu

Le moteur de jeu sera en charge de gérer la partie en cours : l'instance de la classe `game_round`. Lors de la création de la partie, au fil des choix sur l'interface graphique, les différents modules vont enrichir une structure contenant toutes les options nécessaires pour créer une partie : nom de la partie, nombre de joueurs ... Cette structure sera passée à l'instance du `game_round` pour l'initialiser. Et c'est la partie qui va elle-même charger tous les objets graphiques nécessaires et accessoirement faire avancer la barre de progression dans l'interface graphique.

Les positions de tous les objets sont déjà stockées dans le graphe de scène. La liste des joueurs distants est déjà accessible via le moteur de réseau et les machines réseau, il reste à gérer les intelligences artificielles.

VI-1 - Intelligence artificielle

Chaque partie contient une liste des intelligences artificielles qu'elle doit gérer. Elles sont créées au chargement de la partie. La création d'une intelligence artificielle va directement dépendre du type de jeu, un personnage ne réagira pas aux mêmes stimuli et de la même manière dans un jeu de course et dans un FPS.

Chaque IA devra être capable de comprendre son environnement : repérer les objets qui l'entourent. En fonction des objectifs à long, moyen ou court terme, une heuristique devra déterminer les choix à effectuer. La complexité de cette heuristique caractérisera le niveau de l'IA : difficile ou non.

On pourrait aussi classer les méthodes de pathfinding dans la catégorie IA, même si les problématiques ne sont pas vraiment les mêmes.

VII - Pour aller plus loin

Cette architecture a pour but d'être facilement réutilisable pour la réalisation de tout type de jeu. Pour tout développement d'un type de jeu, il conviendra de spécifier un format de fichier (XML ?) pour représenter une partie, une carte, une intelligence artificielle... Une partie contiendra toutes les données nécessaires à la reprise d'un jeu sauvegardé : position des personnages, état de tous les objets... Une carte contiendra les noms de tous les modèles 3D nécessaires, les objectifs mais aussi des données nécessaires aux intelligences artificielles comme par exemple des points de passage, des zones de regroupements...

Le code que je propose se veut le plus générique possible, il va falloir poursuivre dans les directions suivantes :

- dégager une charte graphique ;
- développer un thème graphique CEGUI relatif à la charte graphique (détaillé dans les tutoriels CEGUI) ;
- interfacier un langage de scripts pour les intelligences artificielles et pour le suivi des parties (Boost.Python ? lua ?) ;
- modéliser les objets 3D, les terrains, les cartes ;
- développer un module de chargement/sauvegarde (sérialisation des objets map ?) ;
- réaliser les textures, les sons et les musiques ;
- réaliser des shaders pour l'aspect graphique (GLSL ?) ;
- vérifier l'intégrité des données (cohérence par checksum et grammaire des fichiers xml ?) ;
- intégrer ou non une gestion de la physique (wrapper Newton pour Irrlicht ?).

VII-1 - First Person Shooting

Pour réaliser un FPS, il va falloir une carte, des modèles 3D et faire déplacer ces modèles dans la carte. La caméra devra être spécialisée pour représenter un mouvement à la première personne. Il va falloir enrichir la gestion des événements dans le event_handler pour prendre en compte les coups de feu, il va falloir enrichir la classe player pour rajouter un niveau de vie, éventuellement des équipements. Vous pourrez même vous servir du moteur graphique pour déterminer les objets (et les polygones) situés derrière le viseur. La réalisation d'un FPS se révèle ainsi très conventionnelle.

VII-2 - Real Time Strategy

Ce type de jeu est légèrement plus conséquent. Comme pour un FPS, il faudra gérer les déplacements des unités et développer un modèle de pathfinding. La caméra devra être spécialisée. Il existe des modèles de caméras pour RTS développés par certains utilisateurs d'Irrlicht, mais vous pouvez tout à fait redévelopper la vôtre. Le gameplay et l'ergonomie sont directement dépendants du type de caméra Irrlicht.

VII-3 - Jeu de simulation / course

La réalisation d'un jeu de simulation / course est aussi très conventionnelle, il conviendra d'utiliser un modèle physique performant pour gérer correctement tous les déplacements des objets.

VII-4 - Jeu massivement multijoueurs : MMO...

L'aspect 3D lié à un jeu massivement multijoueur est la même que pour un jeu non massivement multijoueur. Ce qui va faire la différence sera l'utilisation d'une base de données pour stocker les positions de tous les joueurs. Ainsi, en mode serveur il faudra rajouter un observateur au graphe de scène : un module d'accès aux données, pour accéder par exemple à une base de données spatiale relationnelle, qui serait tout à fait pertinente pour ce genre d'utilisation.

Un autre aspect concernera les personnages non joueurs : il sera plus délicat de les faire gérer par les machines réseau, la cohérence du jeu sera beaucoup plus difficile à conserver. La solution la plus simple serait de les gérer au niveau du serveur.

La mise en place d'un jeu massivement multijoueur peut aussi avoir un autre impact sur la gestion réseau : il n'y aura pas forcément un seul serveur. Il faudra donc mettre en place un système de redirection des messages d'un serveur à l'autre selon que les joueurs sont connectés sur un serveur ou sur un autre. Un simple identifiant pour une machine réseau ne sera plus suffisant pour communiquer avec un joueur, il faudra que l'identifiant soit utilisé à la manière d'une adresse IP, passant de routeur en routeur. Chaque serveur devra savoir au premier coup d'oeil où transférer un message.

VII-5 - Les points à ne pas négliger

Il sera très important de réaliser proprement les interactions avec l'utilisateur : la gestion des événements dans le `event_handler` et la gestion de l'affichage dans le moteur graphique. Le game play sera le résultat presque exclusif de la bonne conception du `event_handler`. S'il n'est pas pratique, c'est tout le jeu qui ne sera pas pratique. Ensuite toute l'interface graphique gérée par CEGUI est basée sur des images, et des fichiers XML. On peut donc facilement rendre l'interface plus esthétique, plus colorée, plus pratique. Et surtout, une interface esthétique ne sera pas plus lente qu'une interface bâclée.

Les modèles 3D doivent posséder le moins de polygones possible. Les possibilités du moteur et de votre machine ne sont pas infinies. Encore une fois, un modèle réussi ne sera pas plus lent à afficher qu'un modèle bâclé et c'est pareil pour la texture.

VII-6 - Cohérence entre les joueurs

Il faut que le jeu réagisse de la même manière sur les différentes machines qui le lancent. Par exemple il faut que les modèles 3D soient partout les mêmes. De même, il faut que les caractéristiques des intelligences artificielles soient les mêmes. Pour garantir la cohérence du jeu, à chaque connexion d'un joueur, il va falloir vérifier qu'il possède les mêmes fichiers que le serveur. Pour cela, on peut mettre en place un système de comparaison de hash d'une archive contenant tous les fichiers nécessaires à une partie.

Il va aussi falloir éviter qu'on puisse modifier manuellement les fichiers de sauvegarde ou de caractéristiques de personnages. Et pour cela, même principe, on peut chercher à comparer les hashes pour vérifier que tout le monde joue bien avec les mêmes données.

VII-7 - Téléchargement

Voici l'archive contenant le code source et un projet Visual C++ 2005 : [./fichiers/archi.zip](#) 6 Mo

J'ai dû recompiler CEGUI et notamment le module IrrlichtRenderer pour le faire fonctionner avec la version d'Irrlicht que j'ai utilisée : la version 1.2.



VII-8 - Remerciements

Je tiens à remercier toute l'équipe de la rubrique 2D/3D/Jeux pour leurs remarques et leurs impressions ainsi que **gorgonite** , **trinityDev** et **ClaudeLELOUP** pour leur relecture attentive.