

Architecture d'un jeu vidéo 3D, 2ème partie : réalisation d'un RTS en C++

par Pierre Schwartz 

Date de publication : 17 septembre 2007

Cet article précise l'architecture d'un jeu vidéo dans le cas d'un jeu de stratégie en temps réel. J'y explique ce qui change par rapport à une architecture classique. Il est nécessaire d'avoir lu le précédent tutoriel sur **[l'architecture de jeu vidéo en 3D](#)**. De plus, il est primordial d'avoir des connaissances objet, particulièrement en C++.

I - Introduction.....	3
II - Gestion des contextes.....	4
II-1 - Que faut-il changer ?.....	4
II-2 - Nouveau schéma de classes des contextes.....	4
II-3 - Gestion du jeu.....	5
II-4 - Gestion des IA.....	5
II-4-1 - IA pour chaque objet.....	5
II-4-2 - IA globale, par joueur.....	6
II-5 - Gestion des ressources.....	7
III - Gestion du réseau.....	8
III-1 - Que faut-il changer ?.....	8
III-2 - Traitement des messages réseau.....	8
III-3 - Le protocole réseau.....	10
III-4 - L'attribution des IA.....	11
IV - Une GUI personnalisée.....	12
IV-1 - Que faut-il changer ?.....	12
IV-2 - Architecture du système.....	12
IV-3 - Construction d'une interface de jeu.....	14
IV-4 - Acheminement des évènements jusqu'aux widgets.....	15
V - Gestion des données propres au jeu.....	17
V-1 - Que faut-il changer ?.....	17
V-2 - Acquisition des données.....	17
V-3 - Manipulation des données.....	17
V-3-1 - Méthode presque naïve.....	18
V-3-2 - Méthode du graphe triangulé.....	19
VI - Les chaînes de traitement.....	21
VI-1 - Que faut-il changer ?.....	21
VI-2 - Idée générale.....	21
VI-3 - Les actions à traiter.....	21
VI-3-1 - Le pathfinding.....	22
VI-3-2 - Les autres actions.....	23
VI-3-3 - Le comportement des objets.....	24
VI-4 - Les actions non unitaires.....	24
VII - Conclusion.....	25



I - Introduction

Cet article se place dans la continuité de mon précédent article sur l'élaboration d'une architecture de jeu vidéo en 3D. L'idée étant de la spécialiser pour un type de jeu particulier : le jeu de stratégie en temps réel. Rappelons brièvement les caractéristiques de ce type de jeu :

- Plusieurs joueurs s'affrontent sur une carte. Ils peuvent être en équipe.
- Chacun commande ses unités et ses ressources pour battre son ou ses adversaires.
- Les joueurs n'ont pas connaissance de la position des unités adverses, ils ne connaissent que le terrain et éventuellement les positions des unités alliées.
- La partie s'arrête quand une équipe est détruite ou quand un objectif précis est rempli.

La spécialisation de l'architecture a nécessité des modifications dans l'architecture de classes utilisées, notamment pour la gestion du réseau et des types de parties. Ces deux aspects seront traités dans les parties II et III. Chacune des parties suivantes est basée sur un aspect de l'architecture générique qui nécessite d'être revu ou précisé.

Les parties suivantes feront l'objet de points précis qui peuvent s'avérer techniques comme l'intégration de la gestion de pathfinding.

Rappelons dans les grandes lignes les idées développées dans l'article précédent :

- L'exécution du jeu donne tour à tour la main à chaque module (moteur graphique, moteur audio, données) pour les faire évoluer.
- Chaque module doit être au maximum indépendant des autres modules
- Quand un module a le focus d'exécution, il va en profiter pour faire évoluer ses sous modules et ainsi de suite.



II - Gestion des contextes

Entendons-nous bien, par 'contexte', j'entends un 'type de partie', qui peut être

- partie un joueur
- partie multijoueur en réseau LAN en tant que client
- partie multijoueur en réseau LAN en tant que serveur
- partie multijoueur en réseau Internet en tant que client
- partie multijoueur en réseau Internet en tant que serveur

II-1 - Que faut-il changer ?

Le contexte doit être géré comme étant un conteneur régulièrement instancié et détruit au fur et à mesure que le joueur enchaîne les parties, chaque partie devant vider toutes les données de la partie précédente.

Le jeu va instancier un contexte selon les choix de l'utilisateur. Et le contexte possèdera toujours la carte et les joueurs. J'y rajoute un objectif pour déterminer la fin de la partie. Une gestion des contextes via des pointeurs intelligents permet de réinstancier facilement de nouveaux contextes. En ayant toujours au plus un contexte instancié, chaque nouveau contexte créé va écraser le précédent.

II-2 - Nouveau schéma de classes des contextes

Voici un schéma simplifié des contextes :




Lors d'une partie en réseau, plusieurs contextes de types différents devront être instanciés par les différents joueurs, ce qui paraît normal.

II-3 - Gestion du jeu

Le contexte de jeu est en charge de la gestion des joueurs, c'est lui qui va autoriser de nouveaux joueurs à rejoindre la partie en cours, c'est aussi lui qui va déterminer si des joueurs ont perdu ou gagné. Bref, c'est lui qui va relier les différents acteurs du jeu.

Le contexte devient un élément clef qui nécessitera des traitements à chaque frame (ou presque) pour déterminer une conduite à tenir. Le traitement principal sera d'évaluer si le jeu est terminé ou non, en tenant compte de l'objectif de la partie en cours.

 *On pourra n'appeler des traitements qu'une fois tous les X frames ou une fois tous les X millisecondes, cela nous fera économiser des ressources.*

La gestion des objectifs peut se faire de différentes manières : on peut assigner un objectif à chaque joueur et lors de chaque frame, évaluer les objectifs de tous les joueurs. Mais on se rend vite compte que les objectifs des joueurs sont dépendants. C'est pourquoi j'ai choisi de n'évaluer que l'objectif du joueur humain qui fait tourner l'instance du jeu. Cette évaluation peut renvoyer 3 états : partie gagnée, partie perdue ou partie toujours en cours.

Je suis parti sur l'écriture de plusieurs objectifs :

- Détruire tous les ennemis (partie classique)
- Détruire une unité particulière (partie où il faut tenter le tout pour le tout, au risque de tout perdre)
- Tenir pendant une période de temps (typiquement subir un assaut important)


Chaque type d'objectif implémente une fonction d'évaluation, définie comme virtuelle pure dans la classe générale de gestion d'objectif.

De la même manière, on va pouvoir créer des scénarii de jeu en combinant des événements avec des dates de réveil, pour par exemple faire arriver des unités au bout de 10 minutes ou à la destruction d'une cible particulière.

Typiquement pour créer de nouvelles unités au bout de 10 minutes, il faut créer un objectif ad hoc, qui va vérifier le temps lors de son évaluation. L'idée étant de bien comprendre qu'une fois que l'objectif a le focus d'exécution, on peut le surcharger comme bon nous semble. Et bien sûr, on peut associer l'évaluation de l'objectif à un script. Ainsi, une partie deviendra dépendante d'un objectif, implémentant lui-même un éventuel scénario. De cette manière, c'est le script qui va 'implémenter la fonction virtuelle pure d'évaluation'.

II-4 - Gestion des IA

C'est également dans le traitement relatif au contexte qu'on va pouvoir intercaler une évaluation des différentes IA gérées par l'instance courante de l'application.

 *Rappelons que les IA peuvent être distribuées sur le réseau, chaque instance de l'application peut donc potentiellement être en charge de plusieurs IA.*

II-4-1 - IA pour chaque objet

Les objets pourront réagir à certains événements tels qu'une attaque, la découverte d'un objet ennemi ou autre. Chaque objet pourra décider individuellement du comportement à adopter. Ces comportements dépendent directement du moteur d'événements et ne sont pas très 'intelligents'. Ils correspondent aux actions / réactions directes. Les actions engagées sur chacun de ces événements seront dépendantes des types d'objets en jeu et donc seront déportables dans des scripts.

Les gestions propres à chaque objet regroupent également la gestion du pathfinding et des attaques, sujets traités dans la partie VI, chaînes de traitement.



II-4-2 - IA globale, par joueur

Si on veut qu'une IA puisse avoir un comportement réaliste, il faut que les actions des unités soient un minimum concertées. Il faut avoir une stratégie commune pour amener un groupe d'objets à réaliser une tâche précise et non uniquement donner des actions séparées à des objets indépendants.

Une IA peut se représenter comme une suite de valeurs, chacune correspondant à un trait de caractère, comme par exemple la témérité (qui déterminera si le joueur aura plus ou moins d'audace en essayant de s'attaquer à plus fort que soi), la préservation des unités (pour déterminer si le joueur doit continuer une attaque coûte que coûte ou bien ne pas hésiter à se replier en cas de surnombre), le niveau de rush (pour déterminer si un joueur va essayer d'attaquer son adversaire le plus vite possible), etc. Cette modélisation reste très statique et va nécessiter un grand nombre de traits de caractères pour créer des styles de jeu vraiment différents.

L'autre méthode de gestion des IA peut être la gestion par des scripts (LUA ou Python par exemple). L'idée étant d'avoir un moteur général d'IA qui va repérer certains événements dans le jeu pour appeler les fonctions associées pour chaque IA. On peut aussi voir ça comme la création de fonctions virtuelles pures dans le moteur d'IA, qui sont implémentées dans chaque IA différente. Au moteur d'IA ensuite de mettre en avant les événements comme :

- Une unité a été attaquée
- Un groupe ennemi a été repéré
- Un allié est attaqué
- Il n'y a plus d'argent
- ...

Lors de la création d'un contexte, les joueurs sont créés les uns après les autres, qu'il s'agisse de joueurs réseau, du joueur humain ou de joueurs IA. Chacun spécialise sa fonction de création et typiquement le joueur réseau est créé à partir (entre autres) du fichier de script contenant les fonctions virtuelles implémentées. Il conviendra de vérifier au préalable que toutes les fonctions nécessaires sont présentes dans le script.

Les événements seront repérés dans la chaîne de traitements, en partie VI. Une IA ne sera pas en charge de tous les comportements d'un joueur. On peut classer les comportements en plusieurs catégories :

- objectif à long terme : l'objectif de la partie
- objectif à moyen terme : une stratégie en cours
- objectif à court terme : une action d'une unité

Une IA ne sera en charge que de la gestion des objectifs à moyen terme, pour qu'elle puisse développer des stratégies d'attaques (ou de défense) haut niveau sans se préoccuper de la gestion des actions plus élémentaires telles que le pathfinding ou la gestion des attaques. Ainsi tous les joueurs auront les mêmes algorithmes de pathfinding ou de gestion d'attaques. Seuls seront dépendants des IA les choix d'investissements et les développements de stratégies particulières. On pourra aussi rajouter aux IA les réactions aux divers événements qui peuvent se passer pendant le jeu :

```
void ai_player::on_Attacked_By(drawable_object* attacked, drawable_object* attacking){
    // ...
}

void ai_player::on_Ally_Attacked(drawable_object* attacking){
    // ...
}

void ai_player::on_Seen_Enemy(drawable_object* enemy){
    // ...
}

void ai_player::on_No_More_Money() {
    // ...
}
```



...

C'est typiquement dans ces fonctions sur événements qu'on pourrait incorporer des appels à des fonctions scriptées.

II-5 - Gestion des ressources

Toujours dans un souci de gestion de la mémoire, nous devons nous assurer de ne charger qu'une fois chacune des ressources nécessaires, même si elles sont utilisées plusieurs fois. L'idée est de stocker les noms des ressources déjà chargées et lors d'un appel de ne charger que les ressources qui n'ont pas encore été chargées. Voici par exemple un gestionnaire de modèles 3D :

```
class mesh_manager{
public:
    mesh_manager(graphics_engine*);
    ~mesh_manager();
    irr::scene::IAnimatedMesh *get_mesh(std::string&);
private:
    graphics_engine *parent;
    std::map<std::string, irr::scene::IAnimatedMesh*> l_meshes;
};
```

Chaque fois qu'un objet aura besoin d'un modèle 3D, il devra invoquer la méthode get_mesh du gestionnaire de modèles qui procèdera au chargement des modèles non encore chargés.

```
irr::scene::IAnimatedMesh *mesh_manager::get_mesh(std::string& s){
    if (l_meshes.find(s) == l_meshes.end()){
        std::string filename = shared::get()->get_data()->get_objects_path()+s;
        irr::scene::IAnimatedMesh *m = parent->get_scene_graph()->smgr->getMesh( filename.c_str());
        if (m == NULL){
            shared::get()->get_logger()(logger::LL_ERROR) << "Cannot load mesh "<< filename << std::endl;
            return NULL;
        }

        l_meshes[s] = m;
        return m;
    }
    else
        return l_meshes[s];
}
```

J'ai mis en place des gestionnaires similaires pour les textures, les fontes et les sons. On pourrait continuer en en créant un pour gérer des fichiers de scripts.

Les gestionnaires de ressources pourront être indépendants ou non des contextes, libre à vous de décider si vous voulez pouvoir conserver des chargements d'une partie à l'autre ou bien si vous estimez que trop de mémoire sera encombrée.



III - Gestion du réseau

III-1 - Que faut-il changer ?

Voilà encore une partie qui a bien changé depuis l'architecture générale d'un jeu vidéo. J'ai choisi de n'instancier de gestion réseau que lors de la création d'un contexte réseau.

En créant des types de contextes réseaux différents selon le type de jeu, nous sommes en mesure de gérer les messages différemment selon l'instance qui les reçoit.

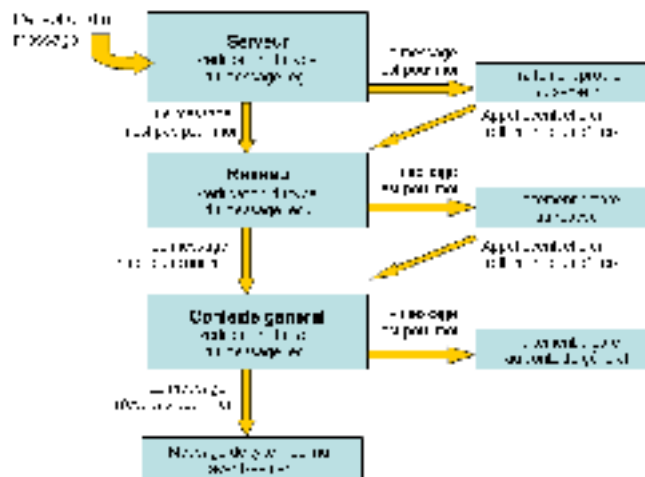
L'architecture du réseau a été revue pour la rendre plus orientée objet, notamment au travers des spécialisations des contextes réseau. L'idée générale reste la même :

- Chaque contexte réseau client est relié à un serveur
- Chaque contexte réseau serveur est relié à une liste de clients

III-2 - Traitement des messages réseau

Chaque réception de message va déclencher un traitement en cascades pour pouvoir cibler le traitement à appliquer : ainsi un message reçu par un serveur va d'abord chercher les traitements propres au serveur pour traiter le message (comme par exemple si un joueur a changé certaines de ses options ou encore si un client a envoyé la position d'une unité). Si aucun traitement spécifique au serveur n'existe pour le message reçu, le message est renvoyé au contexte plus général, à savoir le contexte réseau. Le contexte réseau pourra traiter les messages concernant la suppression d'un joueur. De la même manière, si le contexte réseau ne peut pas traiter le message reçu, le message est envoyé au contexte plus général, à savoir le game_context.

Pour réaliser cet enchaînement de traitements, chaque contexte doit surcharger une fonction de traitement de message composée principalement d'un switch pour repérer quels messages il peut traiter. Les messages non traités (ceux qui finissent dans la clause default du switch sont envoyés dans la fonction de traitement de message du contexte parent.



Une gestion en cascades permet aussi de gérer différemment des messages ayant pourtant le même type. Ainsi, un message indiquant la position d'une unité se contentera de mettre à jour le scene graph (et les autres structures de données associées) pour un client, mais transmettra l'information à tous les autres clients dans le cas d'un serveur. Il faut pour cela que le traitement propre au message "unité a bougé" soit pris en compte dans le traitement propre au serveur ET dans le traitement propre au client.

La réception des messages est effectuée au niveau de la socket. Les messages UDP arrivent par l'unique socket en écoute, au niveau du contexte tandis que les messages TCP arrivent au niveau de la machine réseau. Elle est en charge de désérialiser le flux reçu pour reconstituer le message. Dès qu'il est reconstitué, il est envoyé au parent de la machine réseau à savoir le contexte réseau. Les liaisons dynamiques se chargent d'appeler la bonne version de la fonction `process_event(engine_event &e)` : la version propre au serveur ou au client.

Voici le code de la réception de message TCP dans une machine réseau :

```
void network_machine::TCP_async_receive(const asio::error_code& e, size_t bytes_received){
    if (e){
        // il y a eu une erreur réseau.

        // traiter l'erreur, par exemple en supprimant la machine d'où vient le message
        return;
    }

    // on désérialise le flux reçu
    std::string str_data(&network_buffer[0], network_buffer.size());
    std::istreamstream archive_stream(str_data);
    boost::archive::text_iarchive archive(archive_stream);

    engine_event ne;
    archive >> ne;

    // on ajoute une traçabilité du message
    ne.i_data["FROM"] = id;

    parent->process_event(ne);
}
```

La réception des messages UDP doit être plus robuste, nous ne devons accepter que les messages provenant d'une machine déjà enregistrée :

```
void network_context::UDP_async_read(const asio::error_code& e, std::size_t){
    if (e){
        // il y a eu une erreur réseau.

        // traiter l'erreur, par exemple en supprimant la machine d'où vient le message
        return;
    }

    // on vérifie l'origine du message
    int machine_id = is_address_registered(udp_remote_endpoint.address().to_string());
    if (machine_id == -1)
        return;

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

    engine_event ne;
    archive >> ne;

    // on ajoute une traçabilité
    ne.i_data["FROM"] = machine_id;

    // on traite le message reçu
    process_event(ne);
}
```

Le traitement d'un message se fait ensuite de la manière suivante :

```
void server_context::process_event(engine_event &e){
    switch(e.type){
        case engine_event::PLAYER_HAS_CHANGED_OPTIONS:
            process_client_has_changed_options(e); break;
    }
}
```

```
case engine_event::CHAT:    process_chat_message(e);break;
case engine_event::ADD_NODE: process_add(e);    break;
case engine_event::REM_NODE: process_rm(e);     break;
case engine_event::MOVE_NODE: process_move(e);  break;
case engine_event::CHOSEN_MAP: break; // erreur de protocole,
// les clients n'ont pas le droit d'envoyer ces messages
default:
// appel d'un traitement plus général
network_context::process_event(e);
}
}
```

III-3 - Le protocole réseau

Le protocole utilisé est basé sur plusieurs sécurités visant à garantir son intégrité. La sécurité la plus forte concerne le contrôle des adresses IP de provenance des paquets. Ceci permet purement et simplement d'ignorer les messages de machines non enregistrées dans la partie. Cette authentification se fait via le protocole IP. Les faiblesses du protocole IP seront donc aussi les faiblesses du protocole.

Les connexions TCP pourraient être améliorées en utilisant une couche SSL, (gérée par asio). Ceci garantirait une protection des données sensibles telles que l'envoi d'un mot de passe ou tous les messages importants du jeu.

Le talon d'Achille du protocole réside dans la socket UDP. De part la nature même de ce protocole, l'intégrité des données et des méta données n'est pas assurée aussi bien qu'elle pourrait l'être en TCP (encore plus en SSL). Un programmeur désireux de protéger son application pourrait donc remplacer cette socket UDP en diffusant tous les messages par TCP. Il se heurterait néanmoins à une plus grande lenteur due à TCP.

Pour qu'un joueur soit enregistré auprès du serveur de la partie, il doit s'y connecter en TCP, procéder à l'échange initial d'informations (nom du joueur, mot de passe éventuel pour joindre une partie ...). Le serveur jugera selon ces informations si la connexion avec ce joueur doit être maintenue ou pas. On pourrait tout à fait imaginer créer une liste noire de noms de joueurs ou d'adresses IP de joueurs bannis ou interdits.

Un joueur accepté sera rajouté à la liste des joueurs du contexte courant et sera ensuite informé des modifications du jeu au même titre que tous les autres joueurs. Les déconnexions des joueurs peuvent être détectées puisqu'elles génèrent la réception d'un paquet sur la socket TCP. Ce paquet contient justement un code d'erreur. Rien qu'avec ces quelques opérations, nous sommes en mesure de garantir qu'aucune machine tierce ne viendra troubler le bon déroulement de la partie (sauf attaque au niveau TCP/IP ou UDP/IP, ne dépendant plus de l'application).

Seront envoyés en UDP les messages envoyés souvent, tels que les changements de position des unités. Les messages de création ou de suppression d'unités, qui eux ne peuvent être perdus et sont nécessaires à l'intégrité du jeu, seront envoyés par TCP.

Voici une liste (non exhaustive) des messages réseau qui seront amenés à être échangés.

Type	Direction	Signification	Protocole
MSG_NICK	client vers serveur	Tentative de connexion à un serveur, en donnant login, mot de passe éventuel ...	TCP
REFUSED	serveur vers client	Refus de connexion	TCP
WELCOME	serveur vers client	Acceptation de connexion	TCP
GIVE_ME_YOUR_FPS	serveur vers client	Demande des FPS d'un client, dans le but de déterminer	TCP


		les performances de chaque joueur	
HERE_ARE_MY_FPS	client vers serveur	Réponse au message précédent	TCP
IM_READY_TO_START	client vers serveur	Avertir le serveur qu'on est prêt	TCP
EVERYBODY_IS_READY	serveur vers clients	On commence le jeu	TCP
ADD_NODE	indifférent	Un objet a été rajouté	TCP
REM_NODE	indifférent	Un objet a été supprimé	TCP
MOVE_NODE	indifférent	Un objet a été déplacé	UDP
...			

III-4 - L'attribution des IA

Si vous vous souvenez du précédent article, vous devez vous rappeler qu'il doit être possible de confier la gestion d'une IA à une machine du réseau, pour éventuellement décharger le serveur. Cette décharge est faite en recherchant la machine du réseau qui fournit les meilleurs performances, que ça soit en terme de vitesse de réponse (ping) ou en performances pures (fps). Chaque fois qu'une nouvelle machine s'enregistre auprès d'un serveur, elle va lui communiquer ses performances. Le serveur pourra donc ensuite déterminer la machine réseau la plus à même d'héberger une hypothétique IA.

Une machine distante est capable de déterminer ses FPS, mais il ne lui est pas possible de déterminer son PING. La solution développée est relativement simple : chaque fois que le serveur demandera les performances d'une machine, il va retenir qu'il attend une réponse. Quand la réponse arrivera avec les FPS de la machine distante, il suffira de calculer l'écart de temps entre la demande et la requête. Il ne s'agit pas exactement d'une requête PING réseau au vrai sens du terme, mais elle permet de calculer un temps de réponse, ce qui nous convient tout à fait.

Une fois qu'on connaît les FPS et le PING de chaque machine distante, on peut chercher quelle est la meilleure machine. Nous avons deux critères, il va nous falloir mettre en place une petite heuristique. Une heuristique qui fonctionne pourrait tout simplement être FPP - PING. La machine avec les meilleures FPS et le plus petit PING sera la meilleure.

 **Les performances seront calculées lors de la connexion et ne bougeront plus, un joueur ne souhaitant pas héberger d'IA pourrait donc tout à fait berner le serveur en lançant en parallèle des traitements lourds ou des téléchargements. Il passerait pour une machine aux piètres performances.**



IV - Une GUI personnalisée

IV-1 - Que faut-il changer ?

Nous avons vite remarqué les manques de CEGUI ou du moins l'utilisation pas toujours adaptée aux interfaces plus simples. Chaque interface de CEGUI va nécessiter la création de divers objets lourds et nous cantonnera aux widgets proposés par CEGUI. Dès lors que l'on souhaite avoir des widgets différents, nous avons deux possibilités : travailler le code de CEGUI pour créer nos widgets personnalisés ou bien développer un autre système de widgets basiques. C'est justement cette solution que j'ai retenue.

IV-2 - Architecture du système

Le système de GUI actuel est basé sur CEGUI, bien qu'on puisse assez facilement le remplacer par un autre. En effet, il suffit d'implémenter une liste de fonctions définies virtuelles pures dans le "système de GUI générique" pour se créer un nouveau système de GUI. Attention cependant, la création d'un système complet de gestion de GUI est une opération fastidieuse.

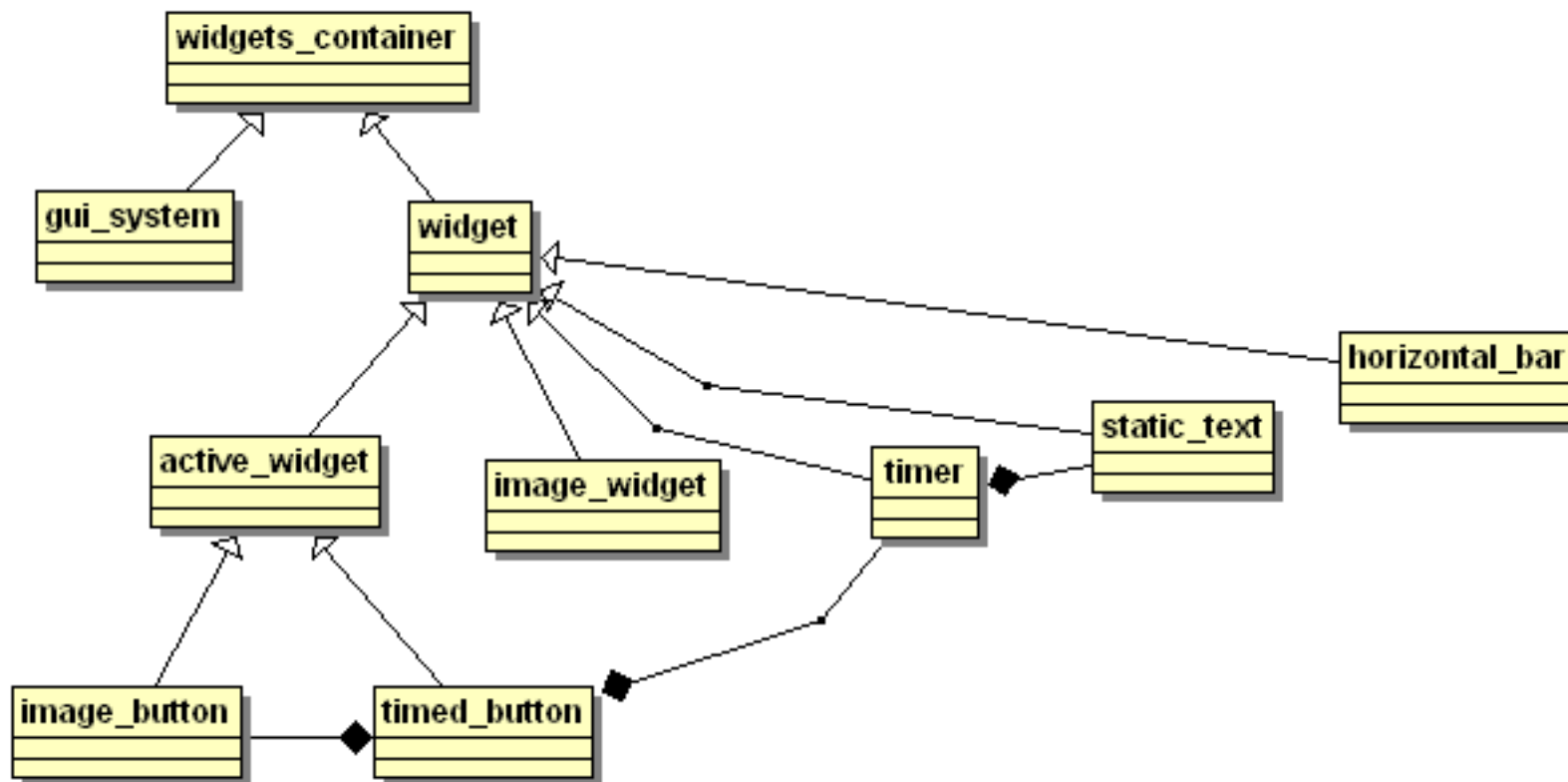
C'est la raison pour laquelle je n'ai développé que quelques widgets, parmi les plus utiles pour pouvoir simplement ajouter des éléments interactifs dans le jeu. Les éléments que j'ai rajoutés sont : le bouton, le timer, le bouton muni d'un timer, la barre de vie (ou barre d'autre chose), le texte simple et le déroulement de textes lors du jeu pour suivre les dernières modifications.

Ce nouveau gestionnaire vient simplement s'ajouter à la liste des éléments à afficher lors du rendu graphique. Il contient une liste de widgets, munis de liens de parenté. Chaque type de widget implémente sa propre fonction draw(). Comme chaque widget connaît l'emplacement qu'il occupe à l'écran, on peut aisément savoir si un événement souris est pour un widget ou pour un autre. Les widgets dits 'actifs' lanceront un traitement sur les différents événements souris. Pour qu'ils puissent analyser la position du curseur, il va falloir intercepter les événements souris dans l'écouteur d'événements relatif au moteur graphique pour simplement les envoyer dans la gestion d'événements du système de GUI nouvellement créé.



Toujours dans un souci de performances il serait très intéressant de classer les différents widgets dans un arbre de partitionnement planaire, de manière à savoir très rapidement si un événement souris est fait sur un de nos widgets ou pas.

Voici un schéma de classes simplifié des widgets mis en jeu.

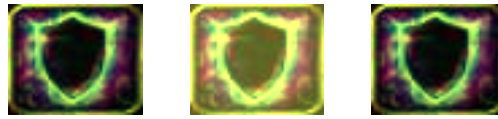


Les widgets ayant du texte sont créés avec une police donnée, les widgets comportant des images sont créés à partir de ces images. Ainsi, le composant 'static_text' est construit à partir d'une police référencée par une image des principaux caractères :



On remarquera la présence de points rouges et jaunes pour délimiter chaque caractère, c'est la méthode utilisée par Irrlicht.

Et voici les 3 images servant à définir un 'image_button' : il nous faut l'image du bouton dans son état normal, l'image lorsqu'il est survolé et l'image lorsqu'il est cliqué. Rien n'empêche d'utiliser une même image pour plusieurs états. Le gestionnaire de textures s'occupe de ne charger la texture qu'une seule fois.



Une fois qu'on a implémenté les widgets de base (images, texte simple), il devient très intéressant de les combiner pour créer par exemple le bouton basé sur un timer. Le timer n'est qu'un texte mis à jour lors de son rendu, on peut donc tout à fait dessiner un timer par dessus une image, à vous de faire attention que le texte ne sorte pas de l'image.



! L'ordre de création des widgets est important, l'affichage se fera dans le même ordre que la création.

IV-3 - Construction d'une interface de jeu

Une interface complète de jeu peut être réalisée en combinant ces widgets de base. Les dessins des zones de boutons sont des simples images sur lesquelles on vient ajouter des image_buttons.

La GUI doit être créée lors du passage à l'écran principal du jeu, donc lorsque le joueur a cliqué sur 'démarrer le jeu' et a renseigné toutes les options nécessaires.

Exemple de création d'interface :

```
irr::core::position2di r;
widget_container *k = get_gui_container();

// tout d'abord les arrières plans
std::pair<int, int> res = parent->get_resolution();
r = irr::core::position2di(res.first - 180, res.second - 100);
image_widget *bg_bottom_right = new image_widget(
    k, // élément parent
    r, // position du coin supérieur gauche, en pixels
    std::string("bg-bottom-right.png") // nom de la texture à afficher
);

r = irr::core::position2di(0, res.second - 70);
image_widget *bg_bottom_left = new image_widget(
    k, // élément parent
    r, // position du coin supérieur gauche, en pixels
    std::string("bg-bottom-left.png") // nom de la texture à afficher
);

// ensuite les autres widgets
r = irr::core::position2di(250, 100);
new image_button(
    k, // élément parent
    r, // position du coin supérieur gauche, en pixels
    std::string("defend.png"), // nom de la texture 'normale'
    std::string("defend-bright.png"), // nom de la texture 'on mouse over'
    std::string("defend.png") // nom de la texture 'on clic'
);

r = irr::core::position2di(250, 300);
new timed_button(
```

```
k,    // élément parent
r,    // position du coin supérieur gauche, en pixels
std::string("defend.png"), // nom de la texture 'normale'
std::string("defend-bright.png"), // nom de la texture 'on mouse over'
std::string("defend.png"), // nom de la texture 'on clic'
irr::core::position2di(8, 20), // décalage avec le coin sup gauche du texte
34    // temps du timer, en secondes
);
```

On peut aussi remarquer l'absence de chemins de répertoires pour les images, en effet, ça sera au gestionnaire de textures de reconstituer le chemin complet à partir du répertoire contenant les textures. L'autre avantage à utiliser des noms courts est tout simplement que l'appel à `std::string::operator<`, dans l'utilisation de la `std::map<std::string, ...>` du gestionnaire de textures, sera plus rapide (même si ça reste vraiment négligeable).

Un autre point intéressant est qu'on ne récupère pas les pointeurs retournés par les appels à `new`. Les widgets seront automatiquement détruits quand leur parent sera détruit. Dans cet exemple, tous les widgets sont fils du gestionnaire de GUI, noté `k`, mais nous aurions eu besoin de récupérer le résultat de `new` si un widget était fils d'un autre widget, auquel cas il aurait fallu passer le pointeur vers le parent au constructeur du widget enfant.

IV-4 - Acheminement des évènements jusqu'aux widgets

Les widgets sont capturés par le moteur graphique, en l'occurrence, Irrlicht. Si vous vous souvenez du précédent article, j'avais choisi de séparer le traitement des évènements selon l'état du jeu dans lequel on se trouvait. On utilisait un gestionnaire d'évènements pour tout ce qui concernait le paramétrage de la partie et on en utilisait un autre pour ce qui concernait le jeu proprement dit. Dans notre cas, il faut acheminer des évènements lors du jeu, nous avons juste à modifier légèrement le gestionnaire d'évènements relatif au jeu proprement dit : la fonction `game_event_handler::OnEvent` :

```
bool game_event_handler::OnEvent(irr::SEvent e) {
    if (!parent->get_device())
        return false;

    if (!parent->get_device()->isWindowActive())
        return true;

    // on crée une simple déviation des messages pour les amener dans la GUI
    widget *w = parent->get_gui()->get_kakou_gui_system()->is_event_for_me(e);
    if (w != NULL) {
        w->process_event(e);
    }

    // reprise du cheminement normal
    switch (e.EventType) {
        [...]
    }
}
```

La fonction `is_event_for_me` renvoie un éventuel widget ayant manifesté le désir de réagir à l'évènement traité.

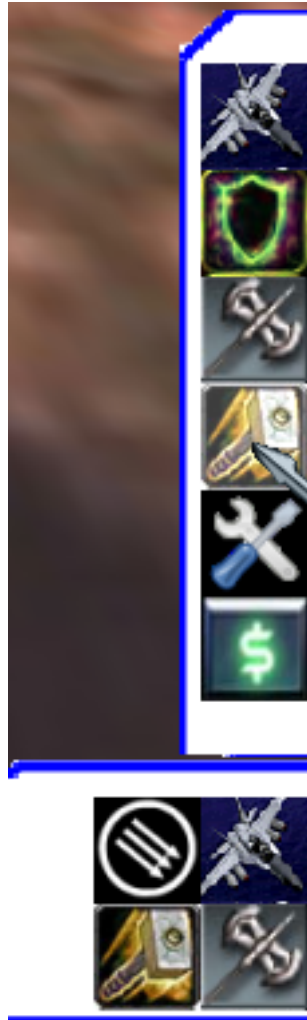
Et de la même manière qu'il nous a fallu intercepter le cheminement des messages pour alimenter nos widgets, il va falloir ajouter le rendu des widgets dans le rendu graphique. J'ai choisi de le greffer sur le rendu du système de GUI déjà existant :

```
void cegui_gui_system::render() {
    if (current_screen == gui_system::MAIN_GAME_SCREEN) {
        k->draw();
    }

    boost::mutex::scoped_lock lock(mutex_gui);
    CEGUI::System::getSingleton().renderGUI();
}
```


où il sera possible d'éviter le test `current_screen == gui_system::MAIN_GAME_SCREEN` en créant deux classes filles, l'une faisant le `k->draw()`; et l'autre non. Le passage de l'instance d'une classe fille à l'autre serait fait lors du passage à l'écran principal du jeu, à l'instar du passage d'un écouteur d'évènements à un autre.

Et voilà un exemple de ce qu'on peut facilement obtenir, en quelques lignes de code :



V - Gestion des données propres au jeu

V-1 - Que faut-il changer ?

La gestion des données propres au jeu couvre deux domaines : d'une part l'acquisition des données au démarrage du jeu, mais aussi la manipulation des structures de données dans lesquelles nous allons gérer le jeu.

Il est plus qu'évident que la gestion des objets sera modifiée entre une architecture générale et une architecture orientée RTS. Nous aurons d'autres objets à gérer et nous aurons des contraintes différentes.

Parmi les contraintes majeures amenées par le RTS, nous aurons la gestion globale des objets, de manière à pouvoir rapidement accéder à des objets. Dans le cas général, les objets sont présents dans les joueurs. Ainsi, si nous cherchons un objet particulier en utilisant des coordonnées géographiques, nous allons devoir parcourir toutes les listes de pointeurs, ce qui bien entendu sera très (trop) long. L'idée est de créer une structure globale pour gérer et classer tous les objets. Nous devons avoir une notion de classement géographique en deux dimensions, ce qui interdit toute utilisation de liste unidimensionnelle, je présente deux solutions permettant de gérer ces objets de manière plus efficace.

V-2 - Acquisition des données

Pour ne pas changer, j'ai utilisé des fichiers XML pour stocker tous les objets dont je pourrais avoir besoin. Chaque module ayant besoin d'acquérir des données XML va instancier un XML_extractor avec le nom du fichier XML à lire, puis va demander un parsing spécifique selon les données qui l'intéressent. La classe XML_extractor possède ainsi autant de méthodes d'extractions qu'il y a de types de fichiers XML : on y retrouve une fonction get_lod() qui renvoie un conteneur de tous les objets 3D du jeu, une fonction get_weapons() qui renvoie un conteneur avec toutes les armes existantes dans le jeu ... Ces fonctions ne sont pas compliquées, elles utilisent simplement l'API XML que j'ai choisie : TinyXML. Pour en utiliser une autre, il suffira de réimplémenter toutes les méthodes getXXX de XML_extractor. Notez au passage qu'il est très important que l'API XML ne soit pas utilisée ailleurs que dans la classe XML_extractor.

L'acquisition va créer les structures de stockage des données. Ces données ne seront ensuite plus modifiées de tout le jeu. Elles serviront à alimenter les données de jeu, notamment au travers des constructeurs. Ainsi, chaque nouvel objet sera construit à partir d'un objet modèle : l'objet lu à partir des données XML.

V-3 - Manipulation des données

Le plus difficile est de trouver une structure de données permettant de stocker et surtout de manipuler simplement et rapidement tous les objets présents sur la carte.

La structure choisie devra pouvoir être stockée ou bien reconstruite, de manière à pouvoir la récupérer lors d'un chargement d'une partie sauvegardée. Et pour reprendre la méthode développée dans l'article précédent, elle devra être (dé)sérialisable.

Quelle que soit la méthode retenue, elle devra s'utiliser exactement de la même manière : elle devra fournir des éléments solutions ou des listes d'éléments. Les algorithmes utilisés doivent faire partie exclusivement de la partie privée de la classe. Enfin ... d'un point de vue objet oui, mais d'un point de vue pratique on pourra avoir besoin de ces structures de données dans d'autres endroits, comme par exemple dans la gestion des IA ou dans le calcul des zones visibles.

L'une des premières questions à se poser est si l'on veut un jeu discret ou continu, par là j'entends un jeu géré en cases ou pas. J'ai choisi de partir sur un jeu continu, les positions de chaque objet seront donc exprimées en coordonnées flottantes. La difficulté principale est de fournir une structure de données permettant d'obtenir rapidement une liste des objets situés à proximité d'un point donné, en effet c'est ce genre de recherche qui sera effectué le plus souvent, que ça soit pour déterminer si un objet ennemi est situé à portée, si une zone est dégagée, si un objet est dans le champ de vision ou pas ...

Une méthode correcte devra implémenter une interface semblable à :

```
struct obj_dist{
    float distance;
    drawable_object* o;
};

class data_container{
public:
    data_container();
    ~data_container();

    // 2 fonctions pour repérer des objets autour d'un point (x,y) ou d'un objet précis
    virtual std::list<obj_dist> get_objects(float x, float y, float dist, obj_functor&)=0;
    virtual std::list<obj_dist> get_objects(drawable_object*, float dist, obj_functor&)=0;

    // 2 fonctions pour repérer l'objet le plus proche d'un point (x,y) ou d'un objet précis
    virtual drawable_object* get_first(float x, float y, obj_functor&)=0;
    virtual drawable_object* get_first(drawable_object*, obj_functor&)=0;
};
```

Le paramètre `obj_functor&` est très important, il faut le voir comme un critère de recherche, permettant de spécialiser les recherches. Ainsi, pour trouver le premier objet à portée, on passera en argument un objet du type `at_attack_range`, dérivé de `obj_functor`, implémentant simplement le test pour savoir si un objet cible est à portée de tir de l'objet attaquant. La diversité de ces critères de recherche est la clef de la variété des traitements que l'on peut effectuer sur la structure de données.

Plusieurs possibilités s'offrent à nous, deux principalement : la méthode presque naïve basée sur une forte consommation de mémoire et la méthode par triangularisation, plus délicate à mettre en oeuvre mais tellement plus élégante

V-3-1 - Méthode presque naïve

La structure choisie pour la méthode presque naïve est basée sur un tableau muni de grosses cases, permettant de réduire très rapidement la liste des objets qui pourraient nous intéresser. Ainsi, à chaque recherche d'objets, on va d'abord déterminer quelles cases du tableau peuvent contenir les éléments qui nous intéressent. Il ne nous restera ensuite plus qu'à parcourir les éléments contenus dans ces cellules pour savoir s'ils sont effectivement à portée, ou bien visibles ou bien tout simplement à une distance donnée. C'est pourquoi chaque case du tableau contient une liste d'objets contenus.



Il conviendra bien sûr de stopper prématurément les boucles de recherche dans le cas où on cherche juste à savoir s'il y a un objet situé à proximité d'un point et qu'un objet a déjà été trouvé.

Cette structure fournit des résultats assez intéressants en terme de vitesse, mais nécessite une grande rigueur dans son utilisation : il ne faut sous aucun prétexte qu'un objet se retrouve sur plusieurs cases à la fois. Chaque fois qu'un objet se déplacera, il faudra mettre à jour la structure en plus de la position stockée dans l'objet.

Le paramétrage principal de cette structure réside dans sa granularité. Le cas extrême d'une seule case revient à parcourir tous les objets à chaque recherche. Je vous laisse imaginer les performances. Le cas situé à l'opposé revient, au contraire, à prendre une grille très fine, on est ainsi presque sûr qu'une case ne peut contenir au mieux qu'un seul objet. Par contre l'inconvénient résidera dans son occupation mémoire qui peut vite exploser.


Voici la définition d'une case du tableau :

```
class data_item{
public:
    data_item(){is_empty = true;}
    float x, z;
```


```
// liste de tous les objets présents dans la case, classés par joueur
std::vector<std::list<drawable_object*>> obj;

bool is_empty;

drawable_object* get_first_enemy(drawable_object*, float);
drawable_object* get_first_unit(float, float, float);
drawable_object* get_first_unit(drawable_object*, float, obj_functor&);
drawable_object* get_first_unit(drawable_object*, obj_functor&);
};
```

 *D'un point de vue conceptuel, il est important d'avoir des fonctions de recherche les plus génériques possibles, de manière à pouvoir rajouter des critères de recherche à volonté, en fonction des besoins.*

Chaque recherche effectuée sur le data_container naïf commence par isoler les cases potentiellement intéressantes, via un appel à get_near_from. Il suffit ensuite de parcourir ces cases à la recherche d'objets validant le critère de recherche.

 *Mais, rappelons-le, cette méthode peut devenir monstrueusement gourmande en mémoire, et si l'on veut des résultats assez satisfaisants, il va falloir y mettre le prix en RAM.*

V-3-2 - Méthode du graphe triangulé

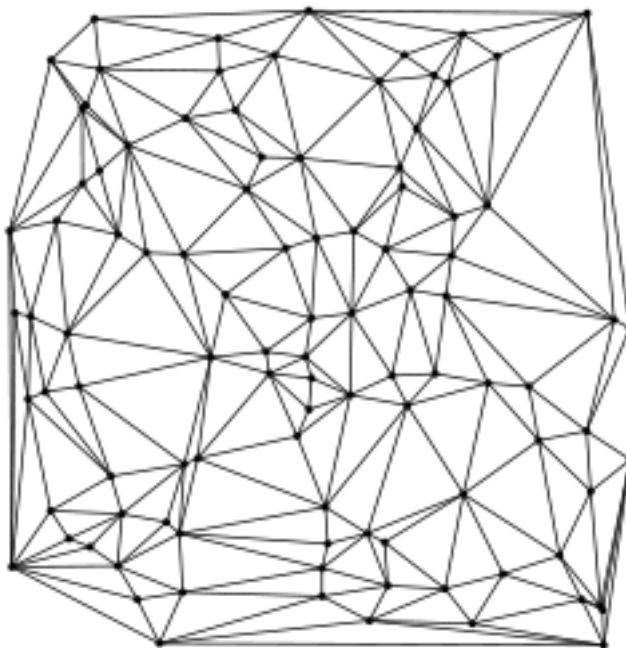
Pour trouver une structure de données alternative, il faut bien poser à plat tous les impératifs que nous devons remplir, on peut aussi essayer de définir comment un humain fait pour arranger des éléments dans un plan.

Les opérations que l'on sera amené à exécuter souvent seront :

- Repérer les objets situés à une distance donnée d'un objet (pour par exemple déterminer si un ennemi est à portée de tir)
- Repérer les objets situés à une distance donnée d'une position (pour par exemple déterminer quelles unités seront endommagées par une explosion)


En stockant naïvement les objets dans un tableau on les considère comme uniques à chaque fois, on oublie complètement les distances qui les séparent les uns des autres, on ne les considère que par rapport au terrain. L'humain, quant à lui, considère les relations entre objets, pour retenir que tel objet est situé de tel côté ou de tel autre côté d'un objet particulier. C'est là dessus qu'est basée cette méthode.

En implémentant un graphe où l'on fixe chaque objet à un noeud du graphe, on est déjà en mesure de retenir les distances entre les objets. La question est maintenant de savoir quel noeud relier à quel noeud. Et si on pouvait faire entrer en jeu une notion de proximité représentée par les arrêtes du graphe, ce serait bien pratique. Et c'est là que Delaunay vient à notre secours. En triangulant le graphe par la méthode de Delaunay, chaque objet sera relié à ses voisins les plus proches ainsi qu'à éventuellement d'autres noeuds.



Exemple de triangulation de Delaunay

Les recherches d'objets à proximité s'en trouvent grandement simplifiées : il suffira de parcourir le graphe en s'éloignant de l'objet référence pour rechercher les solutions. De même, pour trouver les objets situés à proximité d'un endroit, il faudra repérer le triangle contenant cet endroit puis de la même manière, s'écarter de ce triangle pour rechercher les solutions. Et nous pourrions même conserver la notion de critère de recherche développée dans la méthode presque naïve.

 *Il n'est plus nécessaire de donner une valeur aux arcs du graphe. Ça peut juste nous servir à accélérer les recherches de solutions quand il faudra déterminer la distance entre un objet et ses voisins. Les distances seront déjà calculées.*

La triangulation devra être revue à chaque ajout / déplacement / suppression d'un objet. Mais heureusement, il ne faudra corriger que localement le graphe. Et bien sûr, il sera efficace de reprendre une bibliothèque de calcul de diagrammes de Delaunay pour gérer rapidement cette structure.

De par leur légèreté, on pourra tout à fait se créer plusieurs graphes triangulés : par exemple un par joueur, mais on pourra aussi s'en servir pour équilibrer les positions de plusieurs groupes d'objets. Rien ne nous oblige à fixer une seule unité par noeud du graphe ... On peut aussi s'en servir pour déterminer les zones contrôlées par un joueur. L'enveloppe du graphe d'un joueur représente la zone qu'il contrôle, on peut ainsi détecter les intrusions d'objets ennemis ou bien détecter des points faibles quand une zone contrôlée est d'une largeur très faible.

Autant la recherche d'objets voisins d'un objet donné est simple, autant la recherche d'objets voisins d'une position l'est moins. En effet, il va falloir déterminer le triangle contenant la position à étudier. Une méthode efficace est de dérouler un algorithme de sauts de puce pour déterminer par mouvements successifs comment se rapprocher au mieux de la position. On peut obtenir le triangle recherché en relativement peu d'itérations. Bien sûr, la longueur de l'algorithme dépend du nombre d'objets en présence. Dès qu'on ne peut plus améliorer la solution courante, c'est que le noeud courant fait partie du triangle optimal. En cherchant parmi ses voisins les deux plus proches de la position, nous obtenons le triangle solution. De plus, triangulation de Delaunay oblige, les 3 points les plus proches de la position recherchée forment bien un triangle de Delaunay et cet algorithme est toujours convergent.

Même si cette méthode est beaucoup plus élégante et propre que la méthode 'naïve', il va falloir l'optimiser au maximum. Il y aura beaucoup d'appels à ce graphe à chaque seconde, nous devons donc mettre toutes les chances de notre côté en termes de performances.



VI - Les chaînes de traitement

VI-1 - Que faut-il changer ?

La gestion des actions des différents objets est un élément primordial d'un RTS, nous aurons un grand nombre d'actions à gérer et ces actions pourront agir sur un grand nombre d'objets. De la même manière qu'il a fallu une structure dédiée à la gestion des objets, nous avons besoin de savoir comment organiser et traiter les actions à effectuer.

VI-2 - Idée générale

Voilà un très gros aspect d'un jeu de stratégie, il faut pouvoir organiser et traiter correctement tous les ordres donnés.

Le rendu graphique nécessite beaucoup de temps, c'est justement pendant ce laps de temps que nous avons l'opportunité de faire d'autres traitements s'ils sont situés dans un thread ou dans un processus différent. C'est la raison pour laquelle j'ai choisi de gérer tous les ordres et toutes les actions en cours dans un thread à part. Ce thread est démarré lors du démarrage de la partie. L'idée est de stocker toutes les actions à traiter dans une grande liste triée, où les premiers éléments à faire sont les premiers éléments de la liste. Chaque action à faire est associée à une date. Une boucle va s'appliquer à traiter toutes les actions dont la date de réveil est passée. En bouclant suffisamment vite, on peut réussir à respecter les dates de traitement des actions.

VI-3 - Les actions à traiter

Toutes les actions du jeu doivent avoir leur place dans cette grande chaîne de traitements, il faut y mettre les constructions en cours, les comportements des objets, les mouvements ...

Comme pour les critères de recherche, j'ai choisi d'utiliser des foncteurs pour spécialiser les actions stockées dans la liste de choses à faire. Pour chaque action à effectuer, le gestionnaire de traitements va simplement appeler la fonction `run()` à associée :

```
void calculator_mgr::run() {
    bool b;
    boost::mutex::scoped_lock l(still_running_mutex);
    b = still_running;
    l.unlock();
    while (b) {
        // on vérifie s'il y a des actions dans la liste
        if (!q.empty()) {
            boost::posix_time::ptime now(boost::posix_time::microsec_clock::local_time());

            boost::mutex::scoped_lock l2(queue_mutex);
            calculator_todo *c = *(q.begin());
            q.erase(q.begin());
            l2.unlock();

            // on calcule le temps jusqu'à la prochaine action à réaliser
            boost::posix_time::ptime::time_duration_type time_to_wait( c->n - now);

            // on attend la prochaine action
            asio::deadline_timer timer(io, time_to_wait);
            timer.wait();

            // on exécute l'action
            c->run();
            delete c;
        } else {
            // la liste est vide, on patiente un peu
            // .04 s <-> 25 FPS
            asio::deadline_timer timer(io, boost::posix_time::microsec(40000));
        }
    }
}
```

```
timer.wait();
}

l.lock();
b = still_running;
l.unlock();
}
}
```

Si une action est insérée pendant le `timer.wait()`, elle sera traitée au prochain tour de boucle.

Ce thread est arrêté lors de la destruction du contexte, en passant le booléen `still_running` à `false` :

```
boost::mutex::scoped_lock l(still_running_mutex);
still_running = false;
l.unlock();

// on attend la fin du thread
t->join();
```

Les différentes actions à traiter peuvent être de diverses natures : lorsqu'un objet n'a plus rien à faire, il demande à ce qu'on lui donne une tâche, ça fait l'objet d'une action à traiter. La demande est relative à l'objet demandeur et est planifiée pour être effectuée quelques millisecondes plus tard. La résolution de cette tâche détermine le comportement des objets quand ils sont inactifs. Le foncteur associé à ce type de demande va d'abord chercher s'il y a un ennemi à attaquer, puis s'il y a un allié à aller aider. S'il n'y a rien à faire, on replanifie une demande ultérieure. Le délai de planification est directement dépendant de la charge de la liste de choses à faire. En replanifiant immédiatement, on va augmenter les ressources processeur nécessaires au traitement de la liste, il faut donc replanifier les actions avec parcimonie.

On peut également ajouter dans cette liste de choses à faire des traitements non dépendants des unités, comme si par exemple vous souhaitez calculer les rentrées d'argent chaque 2 secondes, en fonction d'un critère qui vous est propre. Il suffira de créer une classe de foncteur qui spécialise la fonction `run()`.

VI-3-1 - Le pathfinding

Presque toutes les actions liées aux unités sont liées à une recherche de chemin :

- Un déplacement d'unité est typiquement une recherche de chemin
- Une attaque doit d'abord déplacer l'attaquant pour le rapprocher de la cible si elle n'est pas à portée.
- Une construction doit d'abord déplacer le constructeur pour le rapprocher du site de construction.

Nous avons une contrainte de taille concernant le pathfinding : les performances. L'idée de base sur laquelle je me suis basé est qu'il n'est pas nécessaire de calculer tout le chemin en une fois, il pourra encore changer d'ici à ce que l'objet arrive à destination. J'ai choisi d'utiliser un algorithme A* modifié, pour l'adapter au milieu continu.

Une recherche de chemin sera calculée au fur et à mesure que l'objet se déplace. A chaque fois qu'il aura avancé de quelques mètres, on calculera la prochaine petite destination et ainsi de suite, jusqu'à destination. Ceci nous garantit aussi que les déplacements ne seront pas omniscients. L'objet se déplacera au même rythme que le déroulement de l'algorithme. Le principe sera assez simple et calqué sur A* :

- L'objet va se déplacer dans la direction de la destination
- Quand il rencontre un obstacle (montagne, autre unité présente ...) il va étudier les deux manières de le contourner : par la gauche et par la droite. L'algorithme va déterminer quel contournement est le plus attrayant en essayant d'avancer de quelques mètres de chaque côté puis en déterminant la qualité de ces contournements (c'est l'heuristique qui peut être tout simplement une distance euclidienne ou une distance de Manhattan). La solution non retenue sera placée dans une liste ouverte, ordonnée suivant la qualité des solutions stockées.

- L'objet va s'avancer dans la direction retenue tout en longeant l'obstacle. Il va avancer d'une distance D, paramètre de l'algorithme (ou de l'objet) puis va réévaluer la situation. Il va calculer la qualité de la position courante et la comparer à la meilleure de la liste ouverte. Il se dirigera ensuite vers la meilleure solution. Chaque solution mise de côté devra être placée en liste ouverte.
- Le processus se continue tant qu'il reste des éléments en liste ouverte et tant que la destination n'est pas atteinte.

Nous n'avons pas de liste fermée car l'objet se déplace au fur et à mesure du déroulement de l'algorithme. Un point à ne pas négliger : quand l'algorithme décide de revenir à une solution mise de côté, il va aussi réaliser un déplacement et donc aussi un pathfinding. Par contre, on est sûr que la solution existe puisque l'objet en vient. Cet A* adapté a cependant des limitations : si une unité bloque un chemin quand l'algorithme l'étudie, puis libère le passage, on peut passer à côté d'une bonne solution.

Chaque fois que l'objet s'est déplacé de quelques pas, l'action s'arrête et se replanifie pour dans quelques millisecondes. Les déplacements ne sont pas omniscients, mais ce n'est pas une raison pour aller 20 fois dans la même impasse, on pourra donc développer un système de marqueurs indiquant des mauvaises solutions de contournement et en tenir compte dans l'évaluation de l'heuristique.

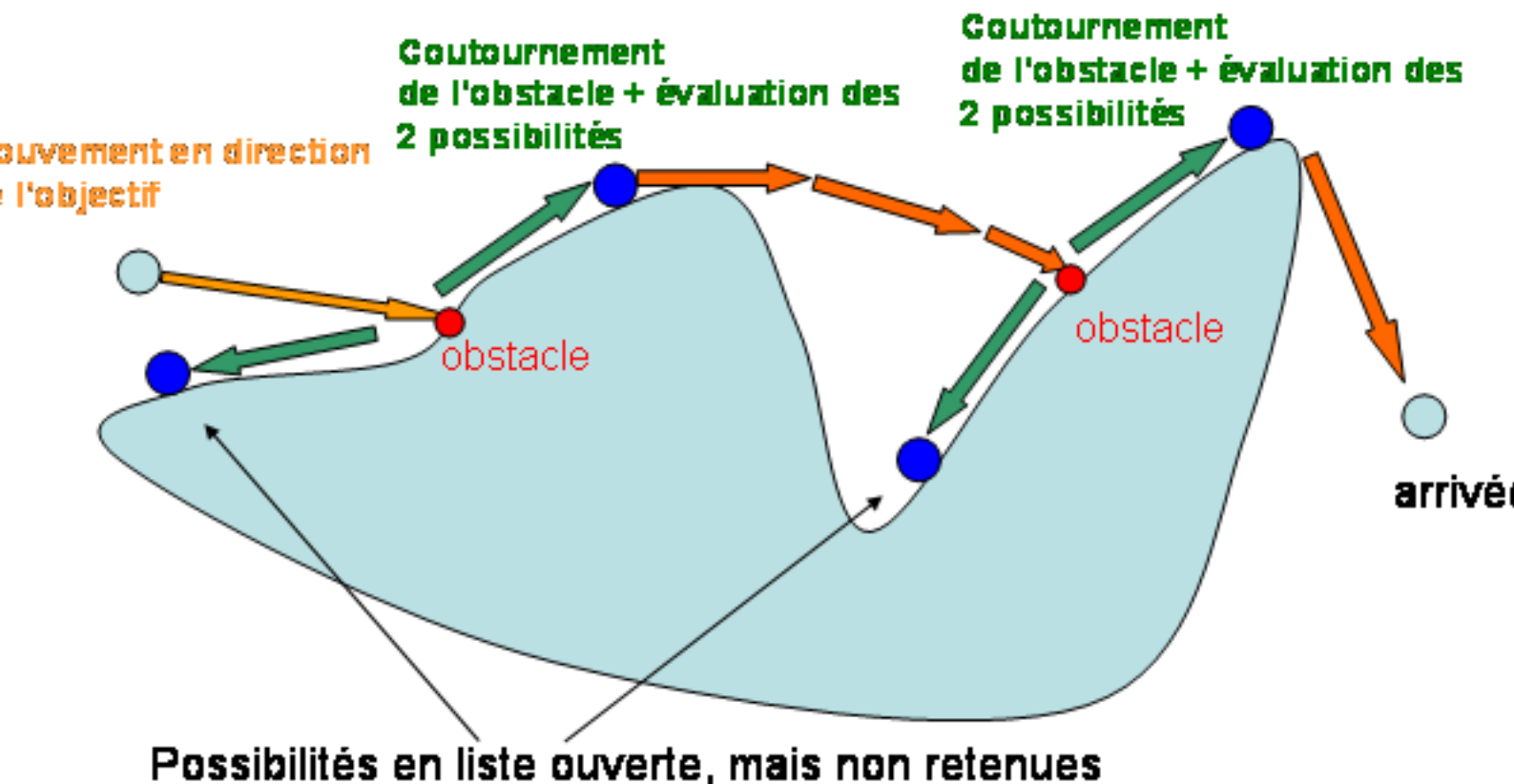



Schéma du pathfinding A* modifié

VI-3-2 - Les autres actions

Pour une attaque, le principe reste le même : à chaque appel de `run()`, le foncteur vérifie si la cible est à portée. Si elle ne l'est pas, on rapproche l'attaquant de la cible et on replanifie une action d'attaque dans quelques millisecondes. Si la cible est à portée, on crée une action relative à une arme, pour simuler un projectile (qui pourra aussi être affiché à l'écran). A ce stade là, nous avons deux actions planifiées pour l'attaque : le rechargement de l'attaquant et le trajet du projectile. L'attaquant aura une nouvelle action d'attaque dans un laps de temps correspondant à sa cadence de tir. De même, le projectile se déplacera en ligne droite ou en cloche (ou autre) et avancera de quelques mètres à chaque fois, pour se replanifier quelques millisecondes plus tard.

La replanification permet de gérer les cas où la cible est mouvante et passe hors de portée. Elle va nécessiter un déplacement pour que l'attaque se poursuive. De la même manière, une gestion séparée des projectiles permet de gérer des projectiles qui suivent la cible, même si elle bouge.


 *Si une cible passe hors de portée, l'attaquant peut ne pas la poursuivre, cela peut faire l'objet d'un paramètre propre au joueur ou à l'unité.*

Pour une construction faite par une unité, il faudra tout d'abord la déplacer jusqu'à la zone de construction. Si la zone est occupée, on a deux possibilités : on replanifie une vérification d'ici quelques dixièmes de secondes ou bien on détermine l'objet qui gêne et si c'est un allié on le fait bouger. Si la construction peut démarrer, on va planifier en cascades autant d'actions qu'il faudra, en fonction du constructeur et de l'objet à construire. On prendra bien soin de ne pas planifier immédiatement après l'instant courant. Les constructions peuvent d'avantage souffrir de cascades, on peut donc sans problème réévaluer la construction d'une seconde en une seconde. Il faut bien garder à l'esprit que plus la file de messages est chargée, plus les performances s'en feront ressentir.

Pour les objets qui n'ont plus rien à faire, une action peut être de leur trouver une occupation, que ça soit en cherchant une unité ennemie à attaquer ou bien un allié à aider.

VI-3-3 - Le comportement des objets

En découplant chaque ordre en une suite d'actions élémentaires, nous sommes en mesure de réévaluer l'action à faire en fonction des événements extérieurs, pour par exemple stopper une action en cours sur un stimulus particulier : "je suis attaqué, je me détourne donc de mon chemin pour riposter" ou bien "je suis attaqué et je vois que je ne pourrais pas gagner, je me replie vers mes alliés"

 *Tous ces comportements liés à des événements peuvent faire l'objet de scripts relatifs à des joueurs.*

VI-4 - Les actions non unitaires

Certaines actions (et c'est le cas de pas mal d'entre elles) ne peuvent pas être effectuées en un seul appel. C'est par exemple le cas d'une recherche de chemin qui va nécessiter plusieurs secondes de recherche, temps qui correspondra à la vitesse de déplacement de l'objet. Ainsi, les différents traitements relatifs à une même action doivent avoir des informations en commun. On peut retrouver dans ces informations communes des données comme la liste ouverte du déroulement de l'algorithme A* modifié, ou bien l'objet qu'on a pris pour cible.

Ces données en commun doivent être créées par le traitement qui démarre l'action. Un pointeur vers ces données devra être passé à l'action reprogrammée chaque fois qu'un traitement se terminera mais nécessitera des traitements supplémentaires. Le traitement qui terminera l'action sera en charge de détruire ces données qui ne seront plus d'aucune utilité. Ce qui reviendra à passer un bloc de données de traitement en traitement jusqu'à ce que l'action soit finie.

VII - Conclusion



Une fois l'ossature générale du jeu écrite (gestion des modules, des données et des traitements) il reste l'autre partie du jeu : l'aspect artistique, avec la création des modèles 3D, des textures, des sons, l'élaboration des caractéristiques de toutes les unités ... Toute cette partie artistique est entièrement indépendante du reste, il pourra donc suffire de changer le package "artistique" pour avoir un jeu dans une autre ambiance situé dans une autre époque ...

Après avoir dégrossi l'architecture d'un RTS, on peut lister une liste de modules qui nécessitent plus d'approfondissement :

- Améliorer les méthodes d'extraction XML pour par exemple utiliser un parseur validant
- Créer un thème graphique pour CEGUI, propre à l'ambiance que vous souhaitez donner au jeu
- Travailler l'implémentation du data_container utilisant la méthode de Delaunay.
- Travailler le pathfinding
- Travailler l'implémentation des IA, notamment en créant un chargement à partir de scripts.
- Créer des sons, des musiques.
- Créer des modèles 3D et des textures en différents niveaux de détail.

Je tiens à remercier **Dut** pour la relecture orthographique et toute l'équipe de la rubrique 2D/3D/Jeux pour leurs conseils avisés.