# Projet Tour

## Choix d’architecture

L’architecture générale de notre application tourne autour d’une classe de gestion de jeu avec les entités et différents moteurs qui permettent de gérer la sortie et les entrées. Tout d’abord, les éléments du jeu, ce que l’on peut voir et réagir avec, sont représentés par des entités. Une entité contient des méthodes afin de réagir avec les autres entités, s’afficher correctement et se met à jour chaque tour de boucle de jeu.

La liste des entités est contenue dans la classe de gestion du niveau de jeu. Ainsi, le niveau peut gérer le chargement du niveau, la création et la suppression des entités tout au long de la partie. Les entités sont chargées dans une liste initiale, mais une autre liste contient les entités visibles à l’écran. Lorsqu’une entité entre dans la zone visible, elle est copiée dans la seconde liste. Seule la seconde liste est affichée et mise à jour. Finalement, lorsqu’une entité sort de l’écran elle est supprimée de toutes les listes.

L’objet de gestion du niveau est contenu dans la classe de gestion de jeu. Cette classe est la classe qui lie tous les éléments nécessaires au déroulement de l’application. Ainsi, la classe de gestion de jeu contient tous les moteurs (graphiques, ressource, entrée clavier, évènement, son) et la logique du jeu telle que la mort d’un joueur, l’ajout de score à un joueur, la boucle principale de jeu, et la gestion des états. Cette classe implémente le design patern du singleton étant donné qu’une seule instance est nécessaire à la fois que la plupart des autres objets doivent accéder à cette classe.

Les différents moteurs permettent de gérer les entrées et les sorties. Ainsi, le moteur graphique permet d’afficher la liste des entités et l’interface utilisateur. Le moteur de son permet de joueur la musique. L’approche par moteur permet à d’autres classes d’accéder à la gestion de certains éléments de manière séparée. Ainsi, si un bout de code a besoin d’afficher un élément, il n’a que besoin d’avoir le moteur graphique afin de remplir ses tâches. L’approche par moteur permet aussi au serveur de ne pas avoir à initialiser des moteurs dont il n’a pas besoin tel que le moteur graphique.

Les entrées tournent autour des évènements de jeu et la gestion de celle-ci selon le moteur d’évènement. Ainsi, un serveur va utiliser la classe de gestion d’évènement qui permet d’écouter les connexions et de renvoyer les événements reçus à tous les clients. Un client quant à lui ne fera qu’envoyer ses évènements au serveur et attendre les réponses de celui-ci.

Le moteur d’entrée utilisateur vérifie si la fenêtre a de nouveaux événements clavier. Si oui, elle notifie l’état courant du jeu afin qu’il traite l’événement.

Les ressources telles que les images ou les modèles sont gérées dans une classe dédiée. Ainsi, le chargement des ressources est transparent ailleurs dans le code et elles ne sont jamais chargées en double.

Finalement, le logging se fait grâce à une petite classe singleton afin de pouvoir l’utiliser facilement n’importe où. Le singleton de la classe de logging doit être initialisé à partir d’une classe dérivée qui implémente la sortie des logs. Ainsi, le serveur peut utiliser la console alors que le client écrit les informations dans la fenêtre graphique.

Il est important de noter que nous n’avons pas de projet pour notre serveur et un autre projet pour les clients. Le serveur utilise la même base de code que le client. L’avantage de cette approche est que nous n’avons pas à maintenir deux applications et qu’un changement dans la logique du client s’applique automatiquement au serveur. Par contre, cette approche amène une petite complexité de plus à savoir quand écrire du code spécialisé pour le serveur ou le client. Le but est d’avoir le moins de code spécifique.

## Abstraction de la couche technique de l’affichage

L’abstraction de l’affichage avec la logique du jeu est importante dans notre architecture étant donné que notre serveur utilise le même code, mais n’a pas besoin d’afficher quoi que ce soit. Ainsi, les seules classes s’occupant de l’affichage sont le moteur graphique et les modèles. Chaque entité contient un modèle qui va définir sa manière de rendre. Ainsi, le changement de notre API d’affichage demanderait seulement de modifier quelques fichiers plutôt que de réécrire plusieurs classes liées à la logique du jeu.

## Héritage et polymorphisme

Notre code repose très fortement sur le polymorphisme. La liste d’entités contient des objets de type IEntity, qui sont en réalité des classes dérivées qui redéfinissent la plupart des méthodes de comportement telles qu’update() ou onTouch() afin de spécialiser le comportement de l’entité. Le polymorphisme est aussi utilisé afin de créer des classes ayant un comportement spécifique pour le client ou le serveur. Ce cas d’utilisation est surtout utilisé avec les classes reliées à la gestion de la réseautique et des événements. Les modèles sont aussi polymorphiques afin de pouvoir laisser les entités travailler avec la classe virtuelle IModel ne contenant aucun code propre à l’affichage.

## Design patern

Notre code implémente plusieurs design patern afin de répondre facilement au besoin de notre architecture. Tout d’abord, notre gestion des modèles utilise le Prototype Design Patern. Lors du chargement de l’application, un fichier de configuration est chargé afin de créer une liste de modèles qui implémentent une méthode virtuelle clone(). Lorsqu’une nouvelle entité a besoin d’un modèle, elle ne fait que copier un modèle de la liste.

Notre application utilise aussi quelques classes implémentant le Singleton Design Patern. La classe de gestion de jeu l’implémente afin de pouvoir avoir accès à tous les éléments du jeu n’importe où dans le code sans avoir à toujours avoir à garder une référence vers la classe de gestion de jeu. Notre classe de logging est aussi un singleton pour les mêmes raisons.

La couche réseau utilise le design patern Command lorsqu’il utilise des callbacks avec les opérations d’écriture et de lecture. Notre couche réseau utilise des opérations asynchrones afin d’envoyer et lire des paquets. C’est pourquoi nous avons besoin de callbacks (Commands) afin de réagir à la réception de donnée. De plus, étant donné que la couche réseau est exécutée dans un autre thread, lorsque la couche de jeu veut envoyer des données elle doit appeler une méthode qui va demander au thread réseau d’exécuter une méthode passée en paramètre. Ce design patern est utilisé avec l’implémentation proposée par Boost dans la librairie Bind.

Notre utilisation de la STL et de Boost nous permet d’avoir accès à une implémentation solide des Iterators. Lorsque nous devons parcourir tous les éléments d’une collection, nous utilisons les itérateurs.

Un semblant de Null Object Design Patern est utilisé pour les modèles avec la classe CNoModel. Ce modèle ne fait simplement rien. Peu importe si setAnim est appelé ou setActive est à true, elle n’affiche toujours rien.

La classe de gestion de jeu utilise une architecture basée sur State Design Pattern afin de pouvoir gérer les différents états possibles du jeu facilement. Certaines opérations sont exécutées dans l’état alors que d'autres sont effectués hors l’état. Les moteurs sont tous mis à jour en dehors de l’état.

## Discussion des choix dans le design

Notre design a été fortement basé sur un projet antécédent afin de sauver beaucoup de temps. En gros, le design mise sur l’utilisation des évènements afin de changer l’état de jeu. La gestion de la logique de jeu se fait à l’aide d’entités et de moteurs afin de simplifier la boucle de jeu. Les moteurs proposent des fonctionnalités alors que les entités définissent le comportement du jeu lui-même. Nous avons choisi ce design afin de bien séparer la logique du jeu du reste. Ainsi, nous pouvions travailler chacun sur notre partie sans avoir à travailler sur le code de l’autre.

Nous avons aussi choisi de séparer l’affichage des entités au maximum pour les mêmes raisons citées plus haut. Un autre avantage à long terme aurait été de pouvoir changer le moteur graphique sans avoir à modifier de code dans la partie logique.

Nous avons pris soin de séparer la gestion des événements et la gestion du transport de ceux-ci afin d’avoir une bonne flexibilité lors de la programmation de la couche réseau. Le débogage est beaucoup plus simple aussi lorsque nous avons une limite claire de la partie événement et de la partie réseau.

Nous avons choisi d’utiliser un fichier de configuration pour la gestion des modèles afin de pouvoir modifier les animations sans avoir à tout recompiler et pour éventuellement ajouter de nouveaux modèles d’entité dans le futur.

Dans le cas des entités, l’ajout de nouveau type d’entités est très simple et l’ajout de nouveaux éléments de jeu demande très peu de modifications dans le code présent. Il serait par exemple possible d’ajouter des blocs en feu qui tue le joueur en n’ajoutant qu’une nouvelle classe dérivant d’IEntity et en ajoutant sa création dans la classe CLevelLoader.

Même si notre projet contient beaucoup de classes, nous avons tenté de séparer les fonctionnalités le plus possible. Ainsi, la modification ou l’ajout de nouvelle fonctionnalité sont assez simplifiés et demandent en général peu de modifications sur d’autres modules.

Parmi les choix différents que nous aurions faits, nous aurions changé notre gestion de jeu entre les clients et le serveur. Présentement, chaque instance a leur propre contexte de jeu et quelques événements de jeu sont propres au serveur tel que l’ajout de score au joueur, alors que d’autres sont propres au client. Par contre, cela ajoute une couche de complexité considérable étant donné que nous devons tenter de synchroniser tous les états de jeu entre les clients et le serveur. La latence avec internet rend la tâche plus ardue et permet d’avoir sur un écran un joueur mort alors que sur le jeu de l’autre joueur celui-ci ne l’est pas. À l’origine, nous avions l’intention de faire en sorte que chaque modification de l’état de jeu doit être traitée par le serveur. Avec la latence, cette technique s’est révélée donnée des résultats de jeu désagréables à cause des contrôles qui ne répondaient pas rapidement. Finalement, seuls certains événements sont validés par le serveur permettant d’avoir des états assez semblables sur chaque machine, mais laissant la possibilité d’avoir des joueurs avec un état différent des autres. Ces problèmes sont surtout le résultat d’une inexpérience de développement de jeu en ligne en temps réel.

Si nous avions à refaire la couche de jeu, nous aurions beaucoup plus utilisé les événements avec une gestion beaucoup plus poussée et fiable. Le principe des événements est de s’assurer que chaque modification de l’état de jeu soit bien communiquée. Or, notre résultat final communique seulement certains changements d’état laissant beaucoup de place à l’erreur. Pour contrer la latence, il serait possible d’appliquer un événement sur le client lorsque celui-ci la crée, et la revalider lorsque le serveur l’envoie aux autres clients avec la latence pour s’assurer qu’elle est bien valide. Présentement, nous envoyons nos événements au serveur et elle n’est pas appliquée tant que le serveur ne la renvoie pas à tous les clients. Cela donne le résultat de contrôle qui ne réponde pas instantanément. Un autre problème est la situation dans le temps d’un événement. C'est-à-dire que lorsque le client applique un événement, il change l’état de jeu au moment où il reçoit l’événement, ce qui laisse chaque client appliquer l’état à des moments différents. Même si cette différence se mesure la plupart du temps en fraction de seconde, c’est parfois assez long pour faire la différence entre la vie ou la mort d’un joueur. Une solution qui pourrait être apportée serait d’identifier le moment lorsque l’événement est créé et l’appliquer comme sur l’état de jeu à ce moment même si l’événement est reçu plus tard. Cela demande beaucoup plus de gestion et demande de pouvoir récupérer un état de jeu dans le passé.

Sinon, quelques changements seraient apportés dans des parties qui étaient initialement codées pour implémenter d’autre fonctionnalité qui a été abandonnée. La liste d’entité avec les CHandle en est un exemple. À la base les CHandle étaient prévus pour pouvoir identifier de manière unique toutes les entités autrement que leur pointeur afin de pouvoir communiquer sur le réseau. Par contre, plus tard nous avons utilisé l’ID du joueur afin d’appliquer des événements et nous avons décidé de ne pas communiquer les actions des entités non-joueur.

### Communication

Nos communications utilisent la bibliothèque Asio de Boost afin d’avoir une gestion de la réseautique poussée et multi plateforme. Un thread est dédié à lire les données et envoyer des données. Nous pouvons diviser notre gestion des communications.

La couche réseau, la couche de sérialisation et la couche événement. La couche réseau est celle qui s’occupe de créer la connexion avec le serveur et d’envoyer des paquets et de lire les données. La lecture de donnée avec Boost.Asio demande la taille du paquet à écouter. C’est pourquoi lors de l’envoi des données, elle rajoute un champ de 2 octets contenant la taille du paquet. Ainsi, l’application lit deux octets de donnés afin de savoir la taille du prochain message à écouter.

La couche de sérialisation permet de transformer le paquet reçu en objet événement ou vice-versa. Lors de l’envoi d’un événement, cette couche va écrire le numéro identifiant l’événement dans un champ de 2 octets et la source de cet événement dans un autre champ de 2 octets. Finalement, les données propres à l’événement sont écrites. Lorsqu’un paquet est reçu, il recrée l’événement du type identifié par le premier champ et l’initialise avec les données binaires reçues dans le paquet.

La couche événement est celle qui s’occupe d’appliquer l’événement reçu à l’état de jeu. Elle doit être adaptée au multithread puisque les événements peuvent être ajoutés à partir du thread de jeu ou bien à partir du thread de réseautique.

Nous n’avons pas mis de temps à implémenter une gestion poussée des erreurs de communication, donc la déconnexion d’un joueur entraîne la fin d’une partie. Par contre, un message d’erreur est affiché avant la fermeture de l’application.

Afin de pallier au problème de délai, nous avons implémenté un événement qui vérifie que les positions des joueurs sont la même sur tous les postes. Une meilleure solution serait d’avoir une gestion plus fiable des événements comme discuté précédemment. L’utilisation de paquet de données binaires plutôt que de donnée en XML permet aussi de réduire un peu le coût de la bande passante et du délai. La complexité de la sérialisation n’est pas si pire que le XML. Par contre, le débogage des communications au niveau des paquets est beaucoup plus dur à lire lorsque les données sont binaires qu’en XML.

### Données d’état de jeu

La quantité de données propre à l’état de jeu sont plutôt nombreuses et complexes et ont certainement contribué à la complication de la synchronisation de celle-ci entre tous les joueurs. L’état de jeu contient la liste d’entité, l’état du niveau, le temps écoulé et les données des joueurs. La liste d’entité est certainement la structure de donnée la plus compliquée à gérer étant donné que les entités peuvent changer à tout moment et que la liste elle-même peut avoir de nouvel élément et en perdre.

### Architecture Client/serveur

Nous avons initialement choisi d’utiliser une architecture client-serveur utilisant la même application afin de pouvoir avoir un état de jeu propre au serveur considéré comme l’état « officiel ». Cela nous permettait de ne pas avoir à réécrire de code pour le serveur. Par contre, avec le temps nous avons décidé que les clients n’allaient pas s’assurer qu’il était synchronisé au serveur ce qui rend inutile le fait que le serveur ait son état de jeu. Le rôle du serveur maintenant n’est que de transmettre les événements à tous les clients et gérer les connexions et le début d’une partie. Ce genre de serveur est beaucoup mieux d’être codé séparément afin d’éviter à traiter un état de jeu additionnel non nécessaire.

### Librairie externe

Nous avons utilisé Boost puisque cette librairie propose plusieurs librairies portables et génériques permettant de résoudre les différents besoins d’architecture rencontrés. Tout d’abord, nous utilisons la bibliothèque Shared\_Ptr afin d’avoir une gestion rapide et réduisant les fuites de mémoire grâce au pointeur intelligent. Ensuite, nous utilisons la bibliothèque Asio afin d’avoir une gestion des communications réseau avec une implémentation asynchrone et orientée objet. La bibliothèque Bind est utilisée afin de créer les méthodes de callback sur une instance de classe. Les fichiers de configuration sont lus et traités à l’aide de la bibliothèque Property\_Tree et son implémentation des fichiers INFO. Finalement, Boost nous permet de gérer les options données en ligne de commande avec Program\_Optioné.

L’affichage graphique, le fenêtrage, la lecture des entrées clavier et la lecture de la musique sont tous effectués grâce à la bibliothèque SFML. SFML propose une interface orientée objet utilisant openGL pour le rendu, win32 pour le fenêtrage et openAL pour le son.

### Gestion des collisions

Lors de chaque tour de boucle de jeu, la méthode virtuelle update des entités est appelée. Dans update, une entité peut se déplacer, elle doit ainsi utiliser les méthodes présentes dans IEntity afin de vérifier les autres entités qui pourraient la croiser. Une entité entre en collision si son rectangle après son mouvement croise le rectangle d’une autre entité. Les méthodes retournent une liste d’entité qui sera ensuite itérée sur toutes les entités en collision et leur méthode virtuelle onTouch est appelée. La méthode onTouch définit la réaction de l’entité.