|  |
| --- |
|  |
| **Projet C #3** |
| Algorithmique et Programmation en C |
|  |
| *Kaaptur : À la conquête du Graal*  *Inspiré par la série télévisée Kaamelott* |
| **Sujet adapté : Kapture – Jeu de Capture de Drapeau** |
| C:\Users\ArTiSTiX\Documents\EFREI\L1\Projets\Projet C #3 - Kapture\design\nouveau_logo_k.png |
| *Par Loïc Belmonte et Rémi Jarasson* |
| *22/11/2009* |

|  |
| --- |
| Rapport de projet sur la création en langage C d’un jeu multi-joueurs au tour par tour de capture de drapeau. |

**Table des Matières**

[Introduction 1](#_Toc257192564)

[Préparation du projet 2](#_Toc257192565)

[A. Le sujet et ses contraintes 2](#_Toc257192566)

[B. Nos idées 2](#_Toc257192567)

[1. Les outils 3](#_Toc257192568)

[2. L’architecture du jeu 3](#_Toc257192569)

[C. Optique et Éthique de Développement 4](#_Toc257192570)

[D. Organisation du développement 5](#_Toc257192571)

[E. Organisation du Rapport 5](#_Toc257192572)

[Le Game Design 6](#_Toc257192573)

[A. Les Graphismes 6](#_Toc257192574)

[B. L’univers du jeu : Kaamelott 7](#_Toc257192575)

[L’Éditeur de Maps 8](#_Toc257192576)

[A. L’interface de l’éditeur et du client 8](#_Toc257192577)

[1. Les Graphismes 8](#_Toc257192578)

[2. Le Code 9](#_Toc257192579)

[B. Structure de la Map 10](#_Toc257192580)

[C. Ecran de jeu 11](#_Toc257192581)

[1. Affichage de la map 11](#_Toc257192582)

[2. Scrolling de l’écran 11](#_Toc257192583)

[D. La Minimap 11](#_Toc257192584)

[E. L’Edition de la Map 12](#_Toc257192585)

[F. Sauvegarde de la Map 12](#_Toc257192586)

[G. Chargement de la map 13](#_Toc257192587)

[H. Générateur de map 13](#_Toc257192588)

[Le Client 16](#_Toc257192589)

[A. Lancer le jeu 16](#_Toc257192590)

[B. Stocker les pions 16](#_Toc257192591)

[C. Affichage 17](#_Toc257192592)

[1. Affichage de la map 18](#_Toc257192593)

[2. Affichage des pions 18](#_Toc257192594)

[D. Le Brouillard et l’Exploration 19](#_Toc257192595)

[E. L’interface 19](#_Toc257192596)

[F. Et le reste ? 20](#_Toc257192597)

[Le Serveur et le Réseau 21](#_Toc257192598)

[A. Premières difficultés 21](#_Toc257192599)

[1. Plus de console ! 21](#_Toc257192600)

[2. Incohérence des fonctions 21](#_Toc257192601)

[B. Déroulement d’une partie 22](#_Toc257192602)

[1. Initialisation de la partie 22](#_Toc257192603)

[2. Déplacements 22](#_Toc257192604)

[3. Résolution des batailles 22](#_Toc257192605)

[4. Test de victoire 23](#_Toc257192606)

[C. Sauvegarde et restauration 23](#_Toc257192607)

[D. Implémentation du Réseau dans le client 23](#_Toc257192608)

[Le Menu 24](#_Toc257192609)

[A. L’interface 24](#_Toc257192610)

[B. Lancer les programmes 24](#_Toc257192611)

[1. Arguments du serveur 24](#_Toc257192612)

[2. Arguments du Client 25](#_Toc257192613)

[3. Arguments de l’éditeur 25](#_Toc257192614)

[C. Retenir la configuration 25](#_Toc257192615)

[Améliorations Possibles 26](#_Toc257192616)

[A. L’optimisation 26](#_Toc257192617)

[B. Les Ressources 26](#_Toc257192618)

[C. Le Contenu 27](#_Toc257192619)

[Bilan du Projet 28](#_Toc257192620)

[D. Nos Acquis 28](#_Toc257192621)

[1. La SDL 28](#_Toc257192622)

[2. L’interface 28](#_Toc257192623)

[3. Le Réseau 28](#_Toc257192624)

[Conclusion 29](#_Toc257192625)

[Le Site du Projet 29](#_Toc257192626)

Introduction

Nous voilà au troisième projet de programmation de l’année, et là encore le ludique est au rendez-vous ! En effet, nous devions comme pour le deuxième projet créer un jeu en C, ce qui comme nous l’avions dit ajoutait une dimension plus motivante au travail.

Cette fois encore le jeu se joue sur un plateau, avec des pions organisés en deux camps opposés qui doivent s’affronter pour obtenir la victoire. Mais plus question de retourner les pions de l’autre, de les encadrer et de savoir qui aura le plus grand nombre de ces pions… non ! Ici, seul un pion fait l’objet de toutes les convoitises : le drapeau de l’adversaire.

Oui, car c’est lui et lui seul qui fait gagner la partie à l’un des deux camps, et on comprend facilement le nom de code du projet, « Kapture » : Si un camp capture le drapeau de l’adversaire, et le ramène dans son propre camp (à coté de son drapeau) alors c’en est fini de l’ennemi !

Ce projet s’annonçait donc prometteur, et était une nouvelle fois l’occasion pour nous de découvrir de nouvelles facettes du développement, mais comme toujours en nous laissant une marge de manœuvre suffisante pour nous permettre d’innover, de montrer nos capacités, mais surtout notre passion.

Nous avons de ce fait misé très gros sur ce projet, en nous investissant pleinement, voire peut-être un peu trop. Cependant il faut dire que le plus gros défi de ce projet était de créer un jeu dans un délai d’un mois et demi ! Et nous n’avions pas l’intention de nous limiter au strict minimum.

Préparation du projet

Cette fois, le projet semblait d’assez grande envergure. L’étape de préparation était donc d’autant plus importante, sachant qu’au précédent projet, nous avions eu d’assez grosses difficultés pour nous organiser, et travailler en parfait binôme.

La préparation s’est déroulée en plusieurs étapes : d’abord la lecture du sujet, pour nous donner une idée du travail demandé, suivie par une mise au point sur la manière dont nous allions créer le jeu, les outils que nous utiliserons ainsi que le plus grand nombre de détails. Enfin, la préparation conclura par la mise en place des règles que nous allions suivre au cours du développement et que nous pourrions appeler « éthique et optique de développement », et enfin, l’inévitable distribution du travail afin de ne pas commettre les mêmes erreurs qu’auparavant.

## Le sujet et ses contraintes

Tout comme pour le précédent projet, le sujet imposait des contraintes que nous devions impérativement respecter.

D’abord, l’interface graphique devait être au moins en console, avec des couleurs grâce à Conio. Ensuite, les règles du jeu devaient être scrupuleusement respectées dans la version de base, et nous avions le libre choix de faire une version alternative si nous laissions la possibilité de jouer aux règles prévues.

Aucune intelligence artificielle n’était imposée, cependant, nous devions fournir au moins une aide sur le déroulement du jeu au joueur. Enfin, la sauvegarde de partie était demandée, on devait permettre aux joueurs de reprendre une partie mise en suspens.

Dans les règles du jeu, que nous ne rappellerons pas sauf en cas de besoin au cours de ce rapport, les pions ont tous un champ de découverte du terrain, ainsi qu’un champ de visibilité tout autour. Nous avons considéré que ces distances étaient imposées dans la version de base, ainsi bien entendu que les règles concernant le déroulement des combats et les déplacements.

Néanmoins, nous avons décidé de ne pas nous restreindre aux simples éléments décrits dans le sujet. Nous avons donc réfléchi à tout ce que nous pourrions intégrer afin de répondre au mieux au sujet, mais également pour étoffer et personnaliser le jeu grâce aux idées que nous pourrions trouver.

## Nos idées

Le sujet présentait de nombreuses fonctionnalités qui auraient pu nous permettre de créer un jeu de bonne qualité, annonçant des parties longues et jouissives… Cependant, le sujet ne parlait que de console, de pions en forme de P, de T ou C.

Ainsi, nous nous sommes posé la question : Pourquoi passer du temps à développer un jeu qui se déroule dans un univers réel, dans une console extrêmement limitée ?

En effet, la console ne refléterait pas bien l’ambiance du jeu, d’autant plus que le grand public ne joue plus aux jeux en console depuis des années, alors pourquoi le développer sans graphismes ? Sachant qu’un tel jeu serait inintéressant, même pour nous !

C’est alors que nous nous sommes dit qu’il était enfin temps de passer à autre chose, de nous attaquer au développement d’interface graphique 2D. Et cela ne présentait que des avantages : ReversiX nous a permis d’exploiter au maximum la console, alors que ce troisième projet devait nous apporter de nouvelles compétences.

Nous avons donc décidé de faire ce jeu en Deux Dimensions.

### Les outils

La mise en place d’une interface graphique nécessite l’utilisation d’une librairie contenant des fonctions de base afin de créer une fenêtre, et d’afficher des images/pixels dessus.

C’est sur la SDL – Simple Directmedia Library, que notre choix s’est porté, car c’est une librairie simple d’accès, gratuite et open source, mais surtout parce qu’elle permet de faire bien plus que de l’affichage : elle permet également de gérer les événements clavier et souris, ainsi que de nombreuses autres fonctionnalités grâce à la présence de librairies additionnelles pour compléter l’API de base.

Par exemple, la librairie SDL\_mixer nous a permis d’inclure du son dans notre projet, et SDL\_image nous a permis d’utiliser d’autres formats d’image, comme le PNG que nous avons utilisé. Mais la SDL a également le gros avantage d’être multiplateforme, rendant compatible notre jeu avec Linux, Windows ou MacOs (non testé avec ce dernier). Avec cela, on était sûr d’avoir une expérience de jeu bien meilleure qu’en console.

### L’architecture du jeu

Pensant alors faire un jeu graphique, nous avons essayé de voir toutes les fonctionnalités à implémenter, et tout ce que nous voulions rajouter par rapport à celles contenues dans le sujet.

* Multi-joueurs, c’est bien !

La première idée, que nous avons eue, avant de décider de faire le jeu en 2D, était de faire un jeu multi-joueurs. Cependant, le premier problème qui se posait avec le sujet est qu’il sous-entendait que les différents joueurs devaient jouer sur une même fenêtre, se passant tour par tour le clavier… et supprimant donc l’utilité du brouillard de guerre et de l’exploration du terrain !

* En réseau, c’est mieux !

Et d’un accord commun et simultané, nous avons pris la décision de faire le jeu en réseau. Nous avions alors conscience qu’il fallait orienter tout notre développement sur ce point. Loïc a alors fait de brèves recherches concernant l’implémentation du réseau, confiant alors que son expérience passée avec l’API Windows le poussait à choisir une autre solution (pour une question de temps de développement). Et il s’est avéré que la SDL pouvait implémenter le réseau grâce à SDL\_Net, une librairie qui semblait très accessible et simple.

Ainsi, nous avons tout de suite pensé à une architecture Client-Serveur, car le Client-Client nous semblait un peu trop compliqué à mettre en place pour une partie avec beaucoup de joueurs. De ce fait, le projet imposait alors que nous fassions 2 programmes séparés. Cela peut sembler une difficulté supplémentaire, mais en réalité, cela nous faciliterait le développement !

En effet, le Client ne serait qu’une interface entre le joueur et le serveur, et ne ferait qu’appliquer les ordres de ce dernier, de manière complètement naïve. Et d’un autre coté, le serveur lui ne s’occuperait que de donner ces ordres, sans s’occuper de l’affichage et de l’apparence du jeu. Sans oublier qu’une telle architecture nous permettait de développer une partie de l’un et de l’autre de manière indépendante : on peut développer le client seulement graphiquement et le faire exécuter des ordres arbitraires, et créer le serveur à part en le faisant envoyer les ordres (en réseau) à un simple client console basique.

* Un menu c’est classe !

Mais, deux problèmes se posent encore : comment lancer les différents programmes, et ensuite, comment allons nous gérer les données d’une partie entre le client et le serveur. Le premier problème nous a simplement imposé de faire un 3ème programme, un simple menu, permettant de configurer la partie ainsi que de communiquer ces paramètres aux programmes tierces. Mais le second problème a complètement organisé le déroulement du développement.

* Un éditeur de maps, c’est encore plus cool !

En effet, il fallait absolument trouver le moyen de créer des cartes, appelées maps, et les enregistrer dans des fichiers, afin de pouvoir les réutiliser dans le serveur, ainsi qu’envoyer ces même données aux clients lors du début de partie. Il fallait donc créer également un éditeur de maps, afin de faciliter leur création, car il aurait été très difficile de faire des maps dans un éditeur de texte.

Ainsi, au cours de notre réflexion, nous sommes passés d’un programme en console, à 4 programmes graphiques séparés, dont 2 qui fonctionnent en réseau. Le projet était donc ambitieux et le travail s’annonçait long et difficile.

## Optique et Éthique de Développement

Le projet ReversiX nous ayant posés des problèmes concernant l’organisation du développement, et notamment au moment de la finalisation du code (qui rappelons-le nous a obligé à revoir une bonne partie du code et des fonctions), nous avons prévu le coup cette fois-ci pour éviter de nouvelles difficultés.

Notre principale erreur avait été d’avoir mal défini nos fonctions, et d’avoir mal découpé notre code. Cette fois-ci, le sujet imposait qu’on sépare le code en modules (code source .c et header .h) ainsi que d’utiliser les fonctions. Nous avons donc respecté scrupuleusement ces instructions, nécessaire au bon développement d’un jeu.

Mais la principale décision que nous avons pris a été d’utiliser pleinement nos connaissances, et travailler quasiment en POO – Programmation Orientée Objet, bien que le C ne soit pas le langage le plus adapté. Ainsi, nous avons fait nos objets grâce aux structures, et les méthodes associées grâce aux fonctions, auxquelles on a systématiquement passé les structures qu’elles concernent. Cette logique de développement nous a grandement facilité le codage, permettant à chaque fonction de faire de nombreuses choses avec peu de paramètres.

Enfin, nous avons jugé que pour la compréhension du code, il serait utile qu’utiliser un maximum d’énumérations, plutôt que des defines qui sont assez longues à taper et à actualiser. Enfin, suite à quelques difficultés rencontrées pour inclure les différents headers des modules dues à une interdépendance des types, nous avons rassemblé tous les types (structures, énumérations, etc.) dans un unique fichier typedef.h.

## Organisation du développement

Enfin, la dernière étape avant le lancement de la programmation était la répartition des tâches, c'est-à-dire, qui allait s’occuper de quelle partie du projet.

Et cette organisation s’est faite de manière quasi-automatique. En effet, Loïc avait déjà une expérience du réseau et est plus à l’aise avec les mathématiques et la création d’algorithmes optimisés. En revanche, Rémi est plus orienté graphismes, sachant manier les outils d’éditions graphique (The Gimp, Adobe Photoshop, Inkscape) et ayant une bonne expérience dans le WebDesign et donc des éléments graphiques. La répartition s’est donc faite ainsi :

|  |  |
| --- | --- |
| **Partie du projet** | **Développeur** |
| Menu | Rémi |
| Serveur | Loïc |
| Interface Client | Rémi |
| Implémentation Réseau (Client) | Loïc |
| Règles du Jeu | Loïc |
| Editeur de Maps | Rémi |
| Ressources / Médias | Rémi |

Notons que le travail en binôme s’est bien déroulé, et cela se ressent une fois dans le résultat.

## Organisation du Rapport

Enfin nous allons pouvoir vous expliquer le déroulement du développement du jeu. Pour ce qui est du plan suivi, contrairement à notre habitude, nous n’avons pas choisi l’ordre chronologique pour deux raisons. D’abord nous avons travaillé en parallèle ce qui rend cet ordre peu pertinent et ensuite certaines parties sont plus faciles à exposer et à comprendre si elles sont rédigées après certaines autres. Nous avons donc choisi de rédiger tout ce qui concerne l’interface, suivi de la partie plus technique avec le réseau et les règles du jeu.

Le Game Design

***Pourquoi faire un jeu vidéo sans graphismes ?***

Afin de rendre le jeu plus attractif, nous avons fait le choix de développer une interface graphique en 2D grâce à la SDL. Mais cela demande un travail supplémentaire assez conséquent. En effet, pour pouvoir créer un jeu graphique, il faut… des graphismes !

## Les Graphismes

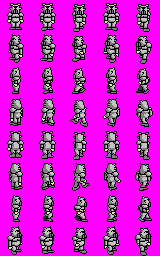
* Principe des graphismes 2D

La particularité des graphismes d’un jeu en 2D est que la résolution des images est très petite, afin de pouvoir boucler celles-ci pour créer un paysage plus grand. C’est ce qu’on appelle le « tile mapping ». Les différentes cases, que l’on appelle tiles, ont une taille de 16x16 ou 32x32 en général ; nous avons fait le choix des 32x32 (environ 1x1cm, appelée TILE\_TAILLE) plutôt adaptées à la taille des maps que l’on peut prévoir (celles-ci ne devront pas être trop grandes pour avoir des parties de bonne durée). Toutes ces tiles sont ensuite enregistrées dans une seule image, qui sera rognée au moment de leur affichage.

* Les personnages

La création des graphismes est donc la première chose à laquelle nous nous sommes attardés, et qui nous a pris énormément de temps. Dès le début, nous avons du commencer à faire les personnages du jeu : les troupes, les piétailles et les éclaireurs.

Le problème des personnages est que le sujet impose que les déplacements puissent se faire dans les 8 directions ! Étant attachés au travail bien fait, et voulant rendre le jeu le plus beau possible, nous avons donc fait les 3 personnages dans les 8 directions. Nous avons également fait les animations de marche, là encore dans les 8 directions. Avec 4 images par cycle de marche, cela revient à 40 positions différentes par personnage, soit 120 tiles !

Pour ce qui est du design des personnages, nous avons fait des croquis dont nous nous sommes inspirés. Mais il est évident que les graphismes en 2D se font pixel par pixel et qu’il n’y a pas d’autre moyen d’avoir une bonne qualité. Nous avons donc d’abord fait les personnages de face et ensuite toutes les autres positions.

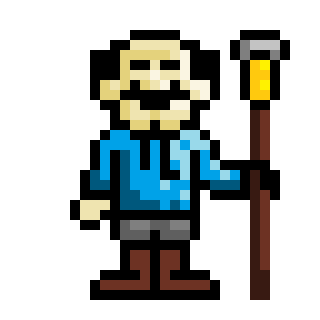
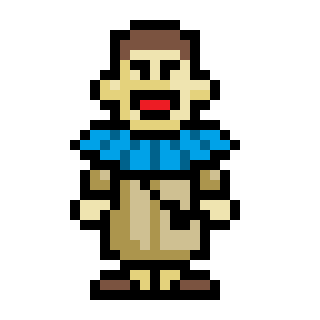
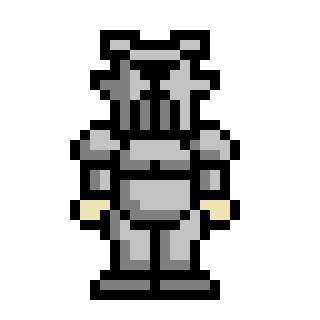


Image d’une Troupe, d’une Piétaille et d’un Éclaireur

* Le Tileset de la Map

C:\Users\ArTiSTiX\Documents\EFREI\L1\Projets\Projet C #3 - Kapture\design\arbre.pngPour la création des graphismes de la map, il fallait penser à tous les éléments que l’on pourrait poser dans l’éditeur. Le sujet parle de terrain normal, forêt et de rivière. Cependant, pour une question esthétique, une tile par type de terrain ne suffit pas : il faut penser à créer les bords de ces terrains, les enchainements entre les différents types. Ainsi, pour un certain type de terrain, il faut au moins 13 tiles !

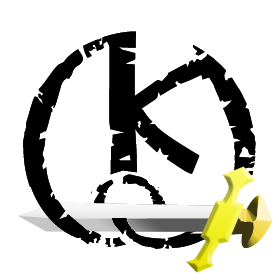
En revanche, pour la forêt (ainsi que les tiles décoratives), les délais nous ont forcé à récupérer un arbre créé par un tiers trouvé sur internet, que nous avons modifié. Cependant, celle-ci nécessitait 20 tiles pour que les enchainements soient corrects. (Arbre créé par *Saurav*)

Afin d’essayer d’ajouter notre petite touche personnelle au jeu, nous avons ajouté à la liste des types possibles des falaises qui seront infranchissables.

## L’univers du jeu : Kaamelott

Au fur et a mesure de la création des graphismes, nous nous sommes aperçus que nous étions très inspirés par la série télévisée Kaamelott. Nous avons donc gardé l’idée à de s’inspirer de cette série pour l’ambiance du jeu. C’est ainsi que nous avons décidé de renommer les pions :

* Troupe = Chevalier
* Piétaille = Pécore
* Éclaireur

La série Kaamelott s’inspire de la légende Arthurienne dont les principaux symboles sont l’épée Excalibur, les personnages légendaires (Arthur, Lancelot, Perceval, etc.) et bien évidemment, le Graal. C’est ainsi que nous nous sommes mis d’accord pour que le drapeau soit en réalité le Graal, ce qui correspond plus à l’ambiance de Kaamelott.

Enfin, pour le logo du jeu, nous avons préféré reprendre le logo de Kaamelott, auquel nous avons apporté une modification. Pour le nom, nous avons tenté de choisir un nom plus proche de la série. C’est pour cela que notre jeu s’appelle KAAPTUR.

Cet univers nous a permis notamment de rajouter des sons, répliques issues des épisodes de la série, et qui permettent de casser le rythme parfois lassant du tour par tour.

L’Éditeur de Maps

Une fois que les graphismes du jeu étaient à peu près finalisés, il a fallu trouver le moyen de faire des maps. En effet, il fallait anticiper le moment où Loïc aurait besoin des fichiers pour le serveur.

Nous avons donc commencé à faire l’éditeur de maps, car cela avait l’avantage précisé précédemment, ainsi que celui d’être profitable pour le client. En effet, tout comme le client, l’éditeur de maps doit afficher la map, ainsi qu’une interface étudiée, et les événements clavier et souris du joueur.

## C:\Users\ArTiSTiX\Documents\EFREI\L1\Projets\Projet C #3 - Kapture\design\idée.pngL’interface de l’éditeur et du client

L’interface de l’éditeur étant la même que celle du client, nous avons fait un croquis grossier de l’interface du jeu.

On voit de nombreux éléments dans ce croquis, qui nous ont permis d’orienter le développement du jeu dans les bonnes directions. On y voit différents éléments superposés dans la fenêtre : l’écran de jeu, la barre latérale, des champs de saisie, des boutons, l’aperçu du personnage sélectionné, des textes, ainsi qu’une minimap.

### Les Graphismes

Cependant, comme toujours, il fallait d’abord avoir les médias nécessaires pour pouvoir les utiliser dans l’interface.

Nous avons pensé au départ à une interface avec l’apparence du bois. Cependant cela demandant des ressources conséquentes (et pas forcément libres de droit), nous avons tenté d’imiter une texture de parchemin. Mais à force de travailler avec des effets d’image, nous avons fini par réussir à recréer une apparence de … pierre. Cela correspondait parfaitement à l’ambiance de Kaamelott (l’épée Excalibur étant plantée dans un rocher), nous avons donc créé de nombreux éléments en pierre : barre latérale, bouton, trous, barre de progression.

Pour la police de caractère, nous avons récupéré celle de la série Kaamelott (notons qu’elle n’est pas libre de droit, et sera probablement changée lors d’une éventuelle publication du jeu sur internet).

C:\Users\ArTiSTiX\Documents\EFREI\L1\Projets\Projet C #3 - Kapture\release\Binaries\progress.pngC:\Users\ArTiSTiX\Documents\EFREI\L1\Projets\Projet C #3 - Kapture\release\Binaries\boutonprecedent.pngC:\Users\ArTiSTiX\Documents\EFREI\L1\Projets\Projet C #3 - Kapture\release\Binaries\boutonsuivant.pngC:\Users\ArTiSTiX\Documents\EFREI\L1\Projets\Projet C #3 - Kapture\release\Binaries\ressources\switch2.png

### Le Code

Pour créer l’interface nous avons donc créé une GUI – Interface Utilisateur Graphique, qui fonctionne sur le principe de superposition d’éléments.

Pour faciliter le passage des données dans les fonctions, nous avons créé une structure, contenant toute l’interface, la surface de la fenêtre, la police de caractère, des informations sur les événements utilisateurs (clic, touche enfoncée, position du pointeur), mais également la liste des éléments : un tableau de gui\_taille éléments.

Un élément est caractérisé par des attributs, dont voici le descriptif :

* Type : Type de l'élément (BOUTON, IMAGE, TEXTE, SWITCH, SAISIE, MAP)
* Image : Surface de l'image de l'élément
* Valeur : Texte ou Valeur de l'élément
* Position : Position de l'élément dans la fenêtre
* Crop : Coordonnées de rognage
* Etat : Etat de l’élément - 0 -> Inactif, 1-> MOUSE\_OVER, 2 - ACTIF
* Visible : Booléen qui indique sa visibilité
* Flag : Action de l'élément
* Groupe : Groupe d'éléments auquel appartient l'élément
* Free : Détermine comment l'élément doit être supprimé

La plupart des attributs sont simples à comprendre. En revanche, le flag nécessite quelques explications quant à son fonctionnement.

* Le Flag

Le flag désigne l’action qui doit être effectuée lors de sa validation, ou lors de son actualisation. Ce flag est utilisé par 2 fonctions principales de l’interface :

effectuer\_gui();  
 actualiser\_gui();

Le flag est attribué grâce à une énumération pour mieux comprendre son utilité. Chacune des deux fonctions effectuent une action différente selon ce flag. A chaque actualisation du programme, actualiser\_gui() est appelée automatiquement, et lorsque un élément est validé (exemple, un bouton est enfoncé), effectuer\_gui() est appelée.

* L’element actif

L’élément actif, nommé gui\_actif dans la structure interface, désigne l’élément qui a été sélectionné, comme s’il avait obtenu le focus. Si cet élément est un bouton, il est actif pendant que le clic est enfoncé, et l’action est effectuée au relâchement. En revanche si cet élément est de type saisie, alors toutes les touches du clavier sont orientées vers la valeur du champ, et la touche Entrée effectue l’action correspondante.

## Structure de la Map

Enfin, abordons la chose essentielle de l’éditeur de map : la structure des données de la map. Nous avons beaucoup pensé cette structure en binôme, car tout le jeu tournerait autour de celle-ci. Plus que convenir à nous deux, elle devait surtout convenir parfaitement aux besoins du jeu. Les fonctionnalités que nous avions pensées nous ont orientées pour sa création :

* Chaque tile fait partie d’un grand tileset et est designée par un identifiant
* Il est possible d’animer les tiles
* Les tiles ont une couleur moyenne utilisable pour une minimap
* Chaque tile est caractérisée par un coût et le type de terrain (Sol, Obstacle, etc.)
* Chaque case de la map est caractérisée par sa tile
* On enregistre pour chaque case si elle a été explorée, et si elle est visible

Ainsi, là encore on a crée une structure pour la map, ou plutôt une hiérarchie de structures, contenant toutes les données nécessaires à son chargement (et sa sauvegarde, pensons-y !). On voit 3 structures différentes : map, tile, tileset.

Rajoutons à cela quelques précision : les tiles peuvent avoir différents types (SOL, OBSTACLE, MORT, CAMP). Le type SOL précise un terrain franchissable, un type OBSCTABLE bloque le personnage qui ne peut pas y aller, un type MORT devait faire mourir le personnage (mais n’est pas implémenté) et enfin le plus important, le type CAMP défini l’emplacement de départ d’un camp, à savoir l’emplacement du Graal. Si ce dernier type n’est pas présent sur la map, la partie ne pourra se dérouler. Il faut donc impérativement qu’il y en ait 2, 3 ou 4.

À partir de la structure de la map, chaque fonction peut accéder aux données de la map. Ainsi, on peut aisément afficher la map, actualiser la minimap, mais surtout, charger une map en initialisant la structure, et sauvegarder cette map en enregistrant cette structure dans un fichier en définissant notre propre format.

## Ecran de jeu

### Affichage de la map

L’écran de jeu est le premier élément de l’interface (n°0), et sa position dans la map est contenue dans les coordonnées de rognage (on a donc un traitement spécial pour celui-ci). À chaque rafraichissement de la fenêtre, on affiche l’interface, et cette étape débute par une actualisation de chaque élément, lors de laquelle la map est « Blittée » sur l’écran.

Afin d’optimiser l’affichage de la map sur l’écran, on ne blitte que les tiles qui se trouvent sur l’écran. Néanmoins, pour un écran de 600x600 (la taille par défaut), cela représente quand même 361 blits rien que pour la map. Et nous verrons à la fin de ce rapport que cela pose quelques problèmes.

Cet affichage se fait très facilement, car il suffit de récupérer les cases grâce à une double boucle POUR, qui se situent entre la position de l’écran (en pixels, donc on divise par la taille d’une tile, TILE\_TAILLE) et la même position + la largeur/hauteur de l’écran. Et rien de plus simple pour bien afficher les tiles comme une grille.

### Scrolling de l’écran

Enfin, une chose intéressante concernant l’écran, c’est le scroll, c'est-à-dire le déroulement de l’écran pour voir le reste de la map.

Nous avons décidé de faire un scroll de 3 manières : Avec les flèches du clavier, sur les bords de l’écran, et via la minimap. Pour le scroll au clavier, il n’y avait aucun souci, cependant, le scroll impliquait une petite subtilité.

En effet, nous voulions éviter qu’un déplacement de la souris sur la barre latérale déplace l’écran. Ainsi, nous avons ajoutés un petit délai de 200ms permettant d’éviter ce désagrément. Pour ce qui est du scroll via la Minimap, il suffira de faire correspondre la position sur la minimap avec la position dans la map, une fois la minimap créée.

## La Minimap

L’algorithme de calcul de la minimap se révèle être, au final, un algorithme de redimensionnement d’image avec un filtre de pondération des pixels.

En effet on part de l’image de la map réelle dont on réduit la taille à une image d’autant de pixels que de cases : les couleurs de ces pixels sont calculées en effectuant une moyenne pondérée des pixels de chaque case/tile. Cette première étape est effectuée au chargement du tileset pour optimiser les temps de calcul lors des mises à jour.

La deuxième étape consiste à redimensionner (tout en gardant les proportions) l’image obtenue pour qu’elle rentre parfaitement dans le cadre qui lui est réservé et dont la taille est 160 \* 160 pixels. On cherche donc le ratio (rapport de la taille de départ et la taille obtenue) optimal c’est-à-dire celui qui permet d’avoir la minimap la plus grande possible dans le carré de 160 \* 160. Ensuite, pour chaque pixel en entrée, on calcul le coefficient qu’on utilisera pour faire la moyenne à la fin, ce calcul prenant en compte la surface du pixel d’entrée qui recouvre le pixel de sortie. Un exemple pour clarifier ce calcul : supposons qu’un pixel en entrée se situe théoriquement après redimensionnement entre deux pixels de sortie. Ce pixel saura alors affecté d’un coefficient ½ et sera appliqué aux deux pixels de sortie concernés.

On finit par faire la moyenne pour trouver chaque pixel de sortie et il ne reste plus qu’à afficher le (magnifique) résultat. Cet algorithme est suffisant pour l’éditeur mais pas pour le client graphique qui doit ajouter les pions des joueurs par-dessus cette minimap. Il faut alors distinguer deux cas : l’agrandissement et le rétrécissement. Dans le premier cas il suffit de remplacer la couleur de la case/tile par celle du pion. Dans le second cas il faut afficher un pixel entier (sans coefficient) au dessus de la sortie pour obtenir au moins un pixel qui sera visible. Si on n’opère pas ainsi le pixel correspondant au pion sera mélangé avec la couleur du terrain alentour à cause de la moyenne pondérée et sera à peine visible.

## L’Edition de la Map

Pour éditer la map, il suffit de récupérer la position du curseur dans la map, à partir de la position du pointeur dans la fenêtre. Cela se fait simplement en additionnant la position du pointeur avec la position de l’écran dans la map (en pixels) que l’on divise par la taille d’une tile.

Ainsi, lorsque l’on a la position du curseur, lorsque l’utilisateur clic, on change la valeur de map.cases[curseur.y][curseur.x].tile par une nouvelle valeur, que l’on enregistre dans l’interface. L’utilisateur a donc la possibilité ensuite de changer cette tile courante, grâce à la roulette de la souris, ou aux boutons précédent / suivant dans la barre latérale, autour de l’aperçu de la tile courante.

Pour pouvoir éditer toutes les données des tiles, la barre latérale contient des champs de saisie, et même un menu déroulant, permettant de modifier le type de terrain, ainsi que le coût en points de mouvement de la tile courante.

D’autres champs dans la barre latérale permettent d’éditer le nom du tileset utilisé (si l’on veut faire plusieurs thèmes), ainsi que le nom de la map, pour pouvoir la sauvegarder.

## Sauvegarde de la Map

Nous avons fait la sauvegarde avant le chargement, ce qui est logique étant donné que le chargement nécessite d’avoir une map déjà sauvegardée.

Pour la sauvegarde, l’idée était simple : on enregistre toutes les données de la structure de manière ordonnée, en binaire (sous forme d’octets, et non pas sous forme textuelle, en ASCII) dans un fichier spécifique à notre jeu, en définissant un format de fichier où les données sont ordonnées pour pouvoir les récupérer.

* Le Format d’une map

Voici le descriptif du format d’une map, que nous avons appelée KMAP (pour Kaaptur MAP), d’où son extension .kmap :

* "KMAP" - 4 octets : Prouve que le fichier est une KMAP
* "map.png"+ EOT – variable : Nom du tileset
* Tilemax – 1 octet : Nombre de tiles (doit correspondre au tileset)
* Liste des Tiles (5 \* Tilemax Octets):
  + Type – 1 octet : type du terrain de la tile n
  + Cout\_pts – 1 octet : Couts en points de mouvement
  + Couleur.R – 1 octet : Composante Rouge de la couleur d’aperçu
  + Couleur.G – 1 octet : Composante Vert de la couleur d’aperçu
  + Couleur.B – 1 octet : Composante Bleue de la couleur d’aperçu
* Largeur – 1 octet : Largeur de la map
* Hauteur – 1 octet : Hauteur de la map
* Liste des Cases (Largeur \* Hauteur Octets):
  + Tile – 1 octet : Tile correspondante à la case

Grâce à ce format, nous pouvions donc récupérer l’exacte structure qui avait engendré le fichier .kmap.

## Chargement de la map

Le format d’un fichier étant assez simple à réutilisé, les explications seront très rapide. Pour charger une map, on vérifie qu’elle commance bien par KMAP. Ensuite, on récupere le tileset (la chaine de caractère jusqu’au caractère nul / EOT), puis tilemax qui nous servira dans une boucle POUR. Cette boucle POUR récupère alors l’un après l’autre les 5 données concernant la tile n. Ensuite, on récupère la largeur et la hauteur de la map qui nous servirons là encore pour une double boucle POUR, selon y et x. On associe ensuite les tiles dans les cases correspondantes.

Afin d’être sûr qu’il n’y a aucun problème, à la fin du processus, on est sensé tomber sur la fin du fichier. Si ce n’est pas le cas, alors on peut soupçonner une erreur, donc on quitte le programme qui n’a pas pu charger de map.

## Générateur de map

Le générateur de map n’était pas une fonctionnalité que nous avions prévue au départ. Pensant faire un éditeur de map justement pour pouvoir laisser la possibilité de créer des map soi-même, nous nous sommes rendu compte que créer une petite map de 32x32 était une tâche assez longue et pénible, vu la complexité du tileset.

Nous nous sommes donc attardés à faire un générateur de map, qui place automatiquement l’eau, la forêt, et quelques éléments décoratifs (pancartes, tonneaux, rochers, buissons, pierres), inspiré par la méthode d’Iso (Jean Dupouy) qui nous en avait parlé.

La génération d’une map aléatoire repose sur 3 étapes, qui sont répétées 1 fois par type de terrain ajouté :

* Mise en place de « graines »
* Agrandissement des graines
* Mise en forme des bordures
* Mise en place des graines

Les graines sont des petits pixels (des tiles posées aléatoirement sur la map). Cette pose se fait grâce à une simple fonction coef\_rand(), qui renvoie 1 ou 0 de manière coefficientée (le coefficient est un rapport sur mille sur le nombre de chance d’avoir 1).

Un petit rapport engendrera des petits points éparpillés, un grand rapport remplira la map de pleins de points resserrés.

* Agrandissement des graines

Une fois les graines « plantées », la génération de la map ré-explore le tableau, et pour chaque graine, va créer rond de terrain autour de celle-ci, avec un rayon aléatoire. Cela va créer des zones plus ou moins grandes, en fonction des coefficients appliqués, et des rayons de chaque terrain.

* Mise en forme des bordures

La dernière étape est la plus compliquée. En effet, c’est celle qui prenait le plus de temps dans l’élaboration d’une map manuellement car il faut placer les bordures en fonction des 8 cases qui entourent la bordure.

Pour mettre des bordures avec un terrain simple (13 tiles pour le type de terrain), il faut récupérer dans une variable mask la configuration autour de la case courante. Cette configuration se fait sous forme de masque avec l’opérateur binaire OU, chaque direction étant identifiée par un bit précis, que nous avons attribués un peu au hasard :

* + - 1 GAUCHE 00000001
    - 2 DROITE 00000010
    - 4 HAUT 00000100
    - 8 BAS 00001000
    - 16 GAUCHE-HAUT 00010000
    - 32 GAUCHE-BAS 00100000
    - 64 DROITE-HAUT 01000000
    - 128 DROITE-BAS 10000000

Ensuite, après avoir établit la configuration, il suffit pour chaque configuration reconnue, de placer la tile correspondante à cette configuration. Cependant, certaines configurations ne correspondent pas à une tile définie. Cela arrive peu souvent, car l’agrandissement des graines permet de rendre les bordures assez régulières, mais parfois, il est impossible de placer une tile.

Dans ce cas là, nous avons prévu le système de manière à ce que la bordure posant problème devienne du type de case 0 (le terrain sur lequel on pose la nouvelle couche). Ensuite, la boucle de mise en forme va revenir au début, et recommencer le travail jusqu’à ce qu’il n’y ait plus aucun problème.

* Génération d’une map complète

Pour générer une map complète il faut placer l’eau, le sol, la forêt et quelques décorations. Pour cela, on part d’un terrain tout en eau, sur lequel on place du sol. Ensuite, on place des blocs de forêt là ou il y a de l’herbe pure (pas de bordures donc). Les bordures se font à l’intérieur des blocs, pour ne pas détruire la couche de terrain inférieure.

La fonction de génération permet de créer des map relativement réaliste, et peut être ajustée en passant différents paramètres (à droite est indiquée une valeur raisonnable):

* BORDURE : Largeur de l’océan autour de la map8
* RAYON\_MAX : Rayon Maximum lors de l’élargissement des graines16
* EAU\_COEF (/1000) : quantité **de terrain** dans la map150
* FORET\_COEF (/1000) : quantité de forêt dans la map50
* DECO\_COEF (/1000) : quantité de décoration dans la map50



Capture d’écran de l’éditeur avec une map générée aléatoirement

Le Client

Une fois l’Éditeur de maps développé, du moins son interface et tout ce qui concernait l’affichage d’une map, nous avons pu commencer à développer le programme principal du jeu : le client graphique. Il ne nous a fallu que reprendre le code de l’éditeur, que nous avons modifié, notamment à cause du fait que la structure principale du programme n’était plus la map mais la partie.

## Lancer le jeu

Pour notre jeu, comme nous avions prévu de le faire en réseau, on a décidé de faire en sorte qu’un client n’ait pas forcément besoin d’avoir les ressources pour jouer sur une map, car le serveur enverrait ces fichiers au début de la partie. Cependant à ce stade, le serveur ne prenait pas encore en charge la partie, car il fallait d’abord créer les structures, ce que nous avons fait pour le client.

* La Structure d’une partie

Une partie se caractérise par certaines données, utilisables aussi bien par le client que le serveur, et qui doivent être synchronisées. Voilà donc le contenu d’une partie, pour laquelle on a crée là aussi une structure :

* La map : L’exacte structure utilisée dans l’éditeur
* Le numéro du joueur / Client actuel
* Le nombre de joueur
* Le joueur dont c’est le tour
* La liste des joueurs
* Le nombre de pions sur la map
* La liste des pions

Sachant qu’un joueur a plusieurs caractéristiques, son pseudonyme, sa couleur, sa position de départ, mais également les médias qu’il utilise (les charsets, qui sont des tilesets de personnages), nous avons créé là aussi une structure.

Pour la liste des pions, nous avons une structure bien étudiée, dont nous allons parler tout de suite.

## Stocker les pions

Les pions sont les éléments clés du jeu. Il leur fallait bien une structure qui puisse les rendre animés, beaux, et facilement maniable. Nous sommes partis au départ sur l’idée que l’on ne plaçait pas un pion dans une case, mais sur une case, que le pion ne devait donc pas faire partie de la structure map, mais dans une liste parallèle.

De ce fait, la position du pion devait être retenue. De même comme il peut se déplacer dans 8 directions, nous devions stocker celle-ci. Ainsi, nous avons créé l’énumération angle, qui contient l’angle de 0 à 7, en partant de la direction S (sud), et allant dans le sens des aiguilles d’une montre vers la position SE (sud-est).

Enfin, pour identifier un pion, il nous fallait :

* Le numéro du joueur à qui il appartient
* Le type de pion : GRAAL, ECLAIREUR, PECORE, CHEVALIER

À cela s’ajoute :

* Le nombre de points de mouvement maximum qu’il peut avoir
* Le nombre de points de mouvement qu’il lui reste
* Sa position actuelle
* Savoir s’il porte le graal et lequel

Ainsi, la structure semblait complète, le serveur n’aurait qu’à envoyer ces informations au début de partie pour créer ces pions et permettre de jouer. Néanmoins, un problème se posait : Comment animer les personnages ?

* L’Animation des pions

Pour animer les pions, nous avions pensé à de nombreux moyens : rajouter des données de destination, rajouter l’animation en cours, etc…

Au final nous nous sommes mis d’accord pour créer une liste chainée d’animations, car c’est ce qui nous semblait le plus simple, notamment pour faire des enchainements. Ainsi, lors d’une animation, le moteur d’affichage des pions afficherait d’abord chaque étape de la première animation, puis la supprimerait une fois finie, et donc passerait automatiquement à la seconde animation.

Nous avons donc créé une structure anim\_list, contenant tout ce qui était nécessaire pour créer une animation, peut importe laquelle :

* Type d’animation : DEPLACEMENT, TELEPORTATION, MORT, COMBAT, etc…
* Position X, Y : la position qui sert à actualiser le personnage
* Valeur : Donnée qui n’a pas d’utilité précise, elle peut être utile plus tard
* Frame : Avancement de l’animation. 0=Pas démarrée, 32=Finie
* \*suivant : Principe de la liste chainée. Une animation contient le pointeur vers l’animation suivante.

Remarque : Nous n’expliquerons pas le principe de la liste chainée (non vue en cours au moment du développement), on considère qu’il est compris.

Cette liste chainée était une excellente idée car pour créer une animation, il suffit que le serveur envoie celle-ci et l’affichage de pions s’occupera automatiquement d’actualiser les données.

## Affichage

Passons maintenant à la partie la plus importante du client : l’affichage des pions et de la map.

### Affichage de la map

L’affichage de la map a déjà été développé pour l’éditeur de maps, donc il n’a pas fallu faire beaucoup de changements. Cependant, le sujet implique que le joueur ne voit pas toute la map, et soit obligé de l’explorer. De même, il ne voit pas tous les pions, il voit seulement ceux situés à une certaines distance des siens.

Nous avons donc réutilisé l’état de la case stocké dans la structure map, pour savoir si elle doit être affichée entièrement, avec du brouillard, ou pas du tout.

Au départ, nous avions prévu d’afficher toute la map, avec le brouillard par-dessus. Cependant, pour des raisons d’optimisation, il était préférable de ne pas doubler le nombre de blits. Nous avons donc trouvé les solutions évidentes de ne pas blitter les cases non-explorées, et de blitter à 50% de transparence les cases masquées par le brouillard. Cette dernière solution nous a donné beaucoup de fil à retordre, car la SDL gère extrêmement mal le canal alpha, ce qui nous a obligés à ruser. Mais le gain n’est pas négligeable : On gagne beaucoup de blits, car la map est rarement affichée entièrement, les 361 blits annoncés deviennent donc un maximum pour la map.

### Affichage des pions

Grâce aux animations et aux données contenues dans la structure pion, on peut facilement afficher les pions dans l’écran de jeu.

Pour cela, on explore la liste des pions, et on vérifie si le pion courant doit être affiché, et c’est le cas si :

* Le pion est sur une case visible
* Le pion est dans la zone de l’écran de jeu

Si les 2 conditions sont réunies, on affiche le pion au bon endroit, avec l’angle correspondant. En revanche, avant l’affichage, on vérifie s’il n’y a pas une animation prévue, car si c’est le cas, celle-ci va modifier les données du pion.

Pour l’image du pion, celle-ci est sélectionnée automatiquement depuis les charsets des joueurs, en fonction des données du pion (propriétaire du pion et type de pion).

* Les animations

Chaque animation procède à ses propres modifications. Ainsi, un switch permet de choisir en fonction du type d’animation les actions nécessaires.

Pour un déplacement, nous avons défini que le cycle de marche se déroule 1 fois. L’image de marche est donc changée toutes les 8 images, pour parcourir en 32 frames, les 4 images de l’animation. Afin d’améliorer l’esthétique du jeu, nous avons également joué avec la transparence : un ennemi qui provient d’une zone masqué va apparaître progressivement, et inversement. Nous avons ensuite fait varier la position d’affichage entre la position de départ et la position d’arrivée en fonction de l’avancement de l’animation.

Pour une téléportation, nous nous sommes dit qu’il serait amusant de donner l’impression que le pion s’envole. Nous avons donc créé une animation, en modifiant l’angle du personnage, afin de le faire tourner sur lui-même. Et à la moitié de l’animation, comme ses coordonnées sont actualisées, il finit son mouvement après téléportation.

A chaque actualisation des pions, nous incrémentons frame, l’avancement de l’animation en cours, et lorsque celle-ci atteint 32, nous la supprimons, et nous remplaçons le pointeur vers la liste des animations par celui vers l’animation suivante.

## Le Brouillard et l’Exploration

Pour le brouillard, nous avons créé une fonction pour actualiser celui-ci. Le principe est simple : on réinitialise l’état des cases, si elle est inconnue, elle le reste, si elle est visible, on la rend masquée (par le brouillard).

Ensuite, il suffit d’actualiser l’exploration en rendant masquée les cases autour de chaque pion appartenant au joueur actuel. Enfin, on rend visible les cases autour des pions du joueur qui ont été explorées.

Pour savoir quelle case doit être modifiée autour d’un pion (exploration ou visibilité), nous avons simplement utilisé le théorème de Pythagore : si pour chaque case de la périphérie…  
*(position relative en x)² + (position relative en y)² ≤ (distance)²*  
… alors on actualise cette case.

Pour chaque type de voisinage, on a des distances différentes, qui peuvent être modifiées au lancement d’une partie.

## L’interface

* Selection de pion

Le client devait permettre au joueur de bouger ses pions. Pour cela, le meilleur moyen qu’il puisse avoir de choisir quel pion bouger est encore de le faire à la souris. Ainsi, comme pour l’éditeur de map, nous enregistrons dans l’interface le numéro du pion sélectionné, en récupérant celui-ci à partir des coordonnées du pointeur (de la même manière que pour les boutons de l’interface).

* Indication de la selection de pion

Afin d’informer le joueur sur le pion qui est sélectionné, nous avons changé l’aperçu de la tile de l’éditeur pour afficher celui du pion. Nous avons accompagné cet aperçu du nom du pion, et nous avons créé une barre de progression pour les points de mouvements. Celle-ci est Verte quand tous les points sont disponibles, et Rouge quand on en a plus du tout, la couleur intermédiaire étant le jaune.

* À qui le tour ?

Enfin, le dernier élément de l’interface qui semblait manquer dans l’interface était une indication du joueur dont c’est le tour. En effet, celui-ci est envoyé par message, mais celui-ci reste assez simple, et ne s’affiche pas longtemps. On a donc crée comme pour l’aperçu du personnage sélectionné, l’aperçu d’un joueur grâce à un avatar coloré (représentant le Roi Arthur), avec un texte qui indique en dessous le nom du joueur qui joue.

* Une fenêtre de messages

Notons également que l’éditeur possède une boite de message, qui lui sert lors de la sauvegarde de partie, ou tout autre message.

Cependant, cette boite était encore plus nécessaire pour le client, ainsi on l’a agrémenté d’une zone de saisie permettant au joueur de taper d’éventuelles commandes. La touche [TAB] permet de switcher la boite de saisie.

Cette boite a été crée tout simplement à partir du code du serveur que nous avions codé en parallèle, pour recréer une console. Cette boite permet d’afficher des messages colorés, et d’envoyer des messages au serveur. C’est ainsi qu’on a eu l’idée d’implémenter un chat, mais également des cheat code (codes de triche) afin de rendre le jeu plus cool :

* elleestoulapoulette : découvre tout le terrain (réplique de Kadoc)
* moonwalk : Tous les personnages dansent le moonwalk

## Les Commandes

Le client met à profit des événements qui lui sont bien spécifiques. En effet, en plus des événements de l’interface graphique, le joueur utilise des touches du clavier pour jouer. Cela va du scrolling de l’écran (comme pour l’éditeur) au déplacement des pions. Voici la liste des touches pour le client :

* A ou Pavé Numérique 7 : Déplacement ⭦ - NO
* Z ou Pavé Numérique 8 : Déplacement ⭡ - N
* E ou Pavé Numérique 9 : Déplacement ⭧ - NE
* Q ou Pavé Numérique 4 : Déplacement ⭠ - O
* S ou Pavé Numérique 5 : en Moonwalk, Danse
* D ou Pavé Numérique 6 : Déplacement ⭢ - E
* W ou Pavé Numérique 1 : Déplacement ⭩ - SO
* X ou Pavé Numérique 2 : Déplacement ⭣ - S
* C ou Pavé Numérique 3 : Déplacement ⭨ - SE
* ⭡ Scrolle vers le Haut
* ⭠ Scrolle vers la Gauche
* ⭢ Scrolle vers le Droite
* ⭣ Scrolle vers le Bas
* TAB Afficher/Masquer la Fenêtre de Chat
* Echap Quitter

Pour le déplacement des pions, cela se fait pas à pas, car il faut envoyer les données au serveur également pas à pas. Nous pourrions améliorer le système en implémentant le déplacement au clic, permettant d’aller à une certaine position. Cependant cela n’étant pas impératif, nous avons préféré garder un déplacement case par case, et garder un client passif.

## Et le reste ?

Le client est, comme nous l’avons dit en partie préparation du projet, un simple exécuteur d’ordre, il ne fait qu’afficher la partie. C’est le serveur qui s’occupe des règles et de faire bouger les pions. Ainsi, tant que l’implémentation réseau n’est pas faite, le client ne fait rien.

Bien sûr lors du développement, nous avons fait en sorte de pouvoir commander les pions au clavier directement, mais seulement pour tester.

Alors pour ce qui est du jeu réellement, il faut aller voir la partie suivante, concernant le serveur et le réseau.



Capture d’écran du client

Le Serveur et le Réseau

Nous abordons enfin la partie la plus technique du projet : le réseau ! En effet les exigences au niveau de la synchronisation des données entre les différents joueurs est tellement contraignante qu’elle a conditionné toute l’organisation du projet autour de l’architecture client-serveur. L’importance du serveur pour le jeu nous a obligé a commencé le développement par cette partie et pourtant c’est l’une des dernière a avoir été terminée.

Pourquoi ajouter une telle contrainte ? Comme nous l’avons dit précédemment, nous avons choisi cette implémentation réseau pour conserver l’intérêt du brouillard de guerre (inutile si tous les joueurs voient les mouvements de leurs adversaires) et augmenter l’intérêt de la découverte du terrain (chaque joueur ne peut connaître que le terrain qu’il a lui-même découvert).

## Premières difficultés

### Plus de console !

Le chargement de la SDL (pour ensuite charger la couche réseau SDL\_Net) redirige les entrées/sorties et il n’est donc plus possible d’utiliser les fonctions printf et scanf comme nous le faisions en console. Nous avons recherché un moyen d’annuler ces redirections, sans succès. La solution alors adoptée a été de refaire une pseudo-console avec les fonctions et la gestion d’évènements de la SDL. Il a donc fallu recoder toutes les fonctionnalités de bases comme pour la saisie clavier, etc. Bien que très instructif, nous nous en serions bien passés !

Au final, nous avons un résultat très satisfaisant : nous pouvons afficher de (belles) couleurs (contrairement à conio avec laquelle nous sommes très limités pour le choix), afficher et entrer des accents, etc. Nous avons même pu réutiliser ses fonctions de saisie pour les applications graphiques citées précédemment. Cette pseudo-console a été réutilisée pour un client de test (en attendant le client graphique) qui ne fait pas partie des fichiers rendus.

### Incohérence des fonctions

Ce qui nous a fait choisir SDL\_Net pour le réseau c’était d’abord sont apparente simplicité et ensuite la portabilité. Cependant cette librairie est étrangement codée : les parties envoi et réception ne présentent aucune cohérence. En effet, après une rapide documentation pour pouvoir effectuer un choix, nous pensions avoir des fonctions non bloquantes comme c’est le cas pour l’envoi de paquets. Malheureusement, la fonction de réception, elle, est bloquante ce qui est gênant pour le serveur comme pour le client : on ne peut pas se permettre d’attendre la réception de paquets sans effectuer des traitements comme les événements clavier ou l’affichage.

Pour remédier à cela nous avons utilisé une méthode avancée de programmation : le multithread. Heureusement, la SDL propose quelques fonctions pour implémenter le multithread. Nous ne nous étendrons pas sur le concept de multithread mais nous retiendrons que cela permet, grossièrement, de lancer un programme en parallèle du programme principal.

Dans notre cas, le programme principal sera chargé de traiter les évènements (clavier, etc.), du traitement des paquets reçu (pas de la réception !) et enfin de l’affichage. Le (ou les, pour le serveur) programme(s) lancé(s) en parallèle (un par client) sera (seront) chargé(s) de recevoir les paquets et de les stockés dans une liste (chainée) de paquet en attente de traitement accessible par le programme principal.

## Déroulement d’une partie

### Initialisation de la partie

Lors de la mise en place d’une nouvelle partie, il faut placer les premiers pions. Le graal est posé sur l’emplacement du joueur (définie par la carte). Pour les autres pions, le sujet précise qu’ils doivent être placés au tour du graal. La fonction place\_libre s’occupe donc de trouver une place libre (sur laquelle on peut poser un pion et qui n’est pas déjà occupée par un autre pion) autour d’une position passée en argument.

La recherche s’effectue avec une double boucle POUR (coordonnées) dans une boucle TANT QUE qui élargit le champ de recherche tant qu’aucune position n’a été trouvée. Toutes les positions trouvées pour une distance de recherche donnée sont stockées puis la fonction choisit aléatoirement l’une d’entre elles à retourner. Le brouillard est ensuite calculé. Toutes les données nécessaires sont envoyées aux clients.

### Déplacements

Avant tout déplacement, le serveur vérifie si les pions sont autour du graal : si c’est le cas un avertissement est envoyé au joueur pour qu’il les déplace lors de se tour sous peine de voir son graal téléporté aléatoirement sur la carte (système anti-campement). Il remet au maximum les points de mouvements. Le serveur attend alors que le joueur dont c’est le tour fasse une requête de déplacement.

Il analyse ensuite cette requête et vérifie que toutes les conditions sont respectées : nombre de points de déplacement restants suffisants, case de destination libre, etc. Si c’est le cas alors le serveur déplace le pion, recalcule le brouillard, etc. Enfin il envoie l’ordre de déplacement aux clients. Ceci est fait en boucle jusqu’à ce que le joueur demande de mettre fin à son tour. Le serveur vérifie alors de nouveau si des pions sont autour de graal : si c’est le cas, il met ses menaces à exécution !

### Résolution des batailles

Le serveur vérifie pour chaque pion (dans l’ordre de déplacement pour que les résultats soient prévisibles et intuitifs) son entourage. S’il y a un ou plusieurs ennemis, il classe ces ennemis dans l’ordre inverse de la menace qu’ils représentent et applique les règles de combats exigées par le sujet. Le classement des ennemis permet de donner priorité à l’attaquant ce qui parait plus logique. Si un pion perdant portait un graal alors il est lâché aléatoirement sur une case adjacente grâce à la fonction place\_libre citée précédemment. Les pions perdants sont renvoyés au départ (toujours avec la fonction place\_libre) sauf lorsque le combat était Chevalier contre Chevalier. Dans ce dernier cas, comme le sujet l’exige, les pions recul dans la direction de leur drapeau (direction calculée grâce à l’évaluation d’un cosinus). Si l’un des chevaliers ne peut pas reculer il est alors renvoyé au départ. Tous ces résultats sont alors envoyés aux clients.

### Test de victoire

Le serveur va ensuite attribuer les graals tombés aux pions alentours grâce à la fonction pion\_case qui détermine si un pion est sur une case passée en argument. Le serveur vérifie ensuite si un pion portant son graal se trouve à son départ, si c’est le cas il repose le graal à son emplacement. Le serveur peut alors enfin passer au test de victoire à proprement parler : si le graal est à sa place et qu’un pion porte un graal ennemi autour de son propre graal alors la partie est terminée et un message de victoire est envoyé à tous les clients. Si ce n’est pas le cas alors on change de tour et on recommence.

## Sauvegarde et restauration

Le fait d’avoir programmé le jeu en réseau à beaucoup simplifié cette étape : toutes les données nécessaires à l’initialisation de la partie pour le client sont enregistrées : les informations élémentaires (nombre de joueurs, tour actuel, etc.), la carte, le terrain exploré, les pions, etc.

Toutes ces données sont enregistrées en binaire donc nous n’avons pas eu besoin d’utiliser des délimiteurs : les données ont une taille spécifique (comme les types en C) et il suffit de recharger les données dans le même ordre qu’elles ont été sauvegardées. Pour les quelques données qui n’ont pas une taille fixe (ex. pseudo, etc.), soit on ajoute le caractère nul de fin de chaîne soit on écrit avant la taille de cette données pour la recharger correctement.

Une fois le fichier chargé, reprendre la partie n’est vraiment pas difficile : il suffit d’envoyer ces données à la place des données initialisées aux clients pour que la partie reprenne.

La seule difficulté repose sur l’identification des joueurs. En effet comme nous sommes en réseau on ne peut pas demander aux joueurs de jouer les pions qu’ils avaient respectivement, il faut trouver un moyen de lui envoyer les siens. Pour cela, nous avons ajouté une comparaison avec les pseudos enregistrés lors de la sauvegarde de la partie. Cela permet d’envoyé les bonnes données aux bons joueurs.

## Implémentation du Réseau dans le client

Enfin, une dernière étape importante après le développement du serveur a été celle de la mise en place du réseau pour le client. En effet, sans client on ne peut pas faire de jeu en réseau.

Pour implémenter le réseau, cela a été relativement rapide. En effet, toutes les fonctions réseau ont été codées pour le serveur, et il n’a fallu faire qu’un copier-coller. On a ensuite modifié la fonction traiter\_paquets() pour permettre d’effectuer les actions ordonnées par le serveur, simplement en regardant son flag dans un switch. On a même pu créer un mini-chat permettant aux joueurs de converser lors d’une partie.

Le Menu

Voici enfin le dernier programme du projet, le Menu. Celui-ci est la passerelle entre les différents programmes précédemment créés, et est le meilleur moyen de configurer les paramètres de chacun.

## L’interface

Pour créer l’interface du menu, nous avons réutilisés le système de l’éditeur et du client, qui était tout à fait approprié pour cela. Néanmoins, c’est grâce au menu que nous avons rajouté pour le système l’attribut groupe qui permettait de créer différents écrans que l’on peut afficher ou masquer grâce aux boutons.

Là encore, la création du menu s’est précédée d’une étape de création graphique, car celui-ci avait besoin d’un fond d’écran, un « Splash Screen » donnant au jeu une certaine personnalité. Pour cela, nous avons utilisés Inkscape, un éditeur vectoriel, sur lequel nous avons tenté de reproduire les arbres du tileset, et Excalibur plantée dans un rocher, afin de rappeler Kaamelott.

## Lancer les programmes

L’utilité du Menu est de lancer les programmes depuis un seul programme, afin de faciliter le lancement de parties. Pour cela, on a 3 rubriques du menu permettant de le faire :

* Rejoindre Partie : Lance un client qui va rejoindre une partie créée précédemment
* Nouvelle Partie : Lance une partie sur un serveur et un client qui s’y connecte
* Editeur de Maps : Lance l’éditeur de map, et charge ou crée une map

Afin de compléter les fonctions du menu , nous avons rajoutés certaines rubriques :

* Options : permet de configurer les programmes
* « Ze Table Ronde » : Crédits du Jeu
* Quitter : Permet de quitter le menu

Cependant, chaque programme nécessitait que l’on passe en arguments certains paramètres, pour configurer une partie, se connecter à un serveur, charger une map, etc… Ainsi, on a rajouté la prise en charge d’arguments sur chacun des programmes. Voici donc un descriptif de chacun des arguments utilisables pour lancer le jeu sans menu.

### Arguments du serveur

Le serveur avait besoin de recevoir certains paramètres afin de créer une partie. De ce fait, on a crée des arguments à fournir au programme lors du lancement, pour chacune de ces données :

* Nom de la map : -map=nomdelamap.map
* Nom de la sauvegarde : -charger=sauvegarde.kjeu
* Nombre de pions (par type) :  
  -chevalier=3  
  -eclaireur=5  
  -pecore=2
* Distance d’Exploration : -dist\_inc=1
* Distance de Visibilité : -dist\_vis=2
* Lancement automatique : -lancer
* Arrêt automatique : -autokill

### Arguments du Client

Le client ne fait que se connecter à un serveur. Pour cela, il n’a besoin que de son adresse et de son port. Mais il est également nécessaire de lui fournir un pseudonyme. Pour une question de simplicité, on lui fournit également les options directement.

* Adresse du Serveur : -adresse=localhost
* Port du Serveur : -port=4444
* Pseudonyme (1 mot): -pseudo=Arthur
* Mode Plein Ecran : -f
* Volume du Son : -volume\_son=100
* Volume de la Musique : -volume\_musique=100

### Arguments de l’éditeur

Enfin, l’éditeur prend certains paramètres, afin de créer la map voulue, ou bien de la charger.

* Nom de la Map : -map=sansnom.map  
  Si la map existe, elle sera chargée et les autres paramètres ignorés
* Nom du Tileset : -tileset=map.png
* Largeur de la map ; -largeur=32
* Hauteur de la map : -hauteur=32

## Retenir la configuration

Afin d’éviter des étapes longues et répétitives, nous avons trouvé intéressant le fait que l’on puisse retenir les paramètres que nous avions lors d’un lancement précédent. C’est donc l’utilité de la rubrique Options.

Pour cela, il suffit de sauvegarder dans un fichier kaaptur.conf, l’ensemble des options que nous voulons retenir. Il n’y en a que 4 de base (pseudo et volumes sonores et mode plein écran) mais nous pouvons imaginer que l’on puisse améliorer le programme pour en rajouter beaucoup d’autres.

Le format du fichier de configuration est très simple, et proche du langage c : on stocke le nom de la donnée, suivi du signe égal, puis de la valeur (les chaines de caractères ne sont pas encadrées par des guillemets), et enfin, terminé par un point-virgule, et un retour à la ligne optionnel, le tout en ASCII. Pour sauvegarder ces options, il faut impérativement cliquer sur le bouton valider.

Améliorations Possibles

Au termes de ces 4 parties, ce projet est terminé… dans sa version de base. Nous avions prévu de faire une version plus personnalisée, avec un mode de jeu avancé, mais nous avons préféré miser sur le perfectionnement de la version de base plutôt que l’ajout de fonctionnalités, de règles, etc…

Cependant, même dans sa version de base, on peut procéder à de nombreuses améliorations, qui ne sont pas forcément nécessaires, mais qui pourrait offrir un avenir au projet.

## L’optimisation

Le principal point à améliorer dans ce projet concerne l’optimisation. En effet, le jeu est totalement jouable, cependant il est sensé être obsolète graphiquement. Or, celui-ci ne tournerait pas correctement sur des ordinateurs moins puissants que ceux que nous avons aujourd’hui.

Et cela est dû principalement à la SDL : L’affichage (les blits) ne sont pas du tout optimisés, et prennent 90% du temps d’exécution du programme, dont la plus grande partie concerne la map.

Au moment où nous nous sommes rendus compte de ce problème, nous avons cherché toutes les solutions envisageables, tout en utilisant la SDL, cependant, il est quasiment impossible d’améliorer l’exécution tout en gardant la même quantité de ressources. La meilleure solution restait alors de ne plus utiliser les blits de la SDL, et de passer par OpenGL.

La solution OpenGL est semble-t-il la meilleure, car c’est un moteur graphique permettant beaucoup plus que de la 2D, mais également d’utiliser la carte graphique à son maximum, au lieu de la puissance processeur. Suite à nos tests, l’utilisation de OpenGL a fait passé l’usage processeur de 40% à moins de 5% !

Pourquoi n’a-t-on pas fait la modification ?

Simplement parce que cela demandait un plus grand travail, et que l’on a estimé préférable le fait d’avancer doucement pas à pas, plutôt que franchir les marches trop vite. Probablement que le prochain projet graphique profitera de cette expérience, et nous poussera à utiliser OpenGL à la place du moteur graphique de la SDL.

## Les Ressources

La seconde chose que nous pourrions améliorer concerne l’ambiance du jeu, les médias que nous utilisons, et que nous avons créés. On peut voir notamment que de nombreux graphismes ont été faits rapidement, avec une relative application. La qualité pourrait aisément être élevée, si le temps le permettait.

Ensuite, nous pouvons remarquer que l’inspiration du jeu se puise dans la série Kaamelott, dont les personnages, les répliques, les éléments iconographiques existent déjà, et dont nous n’avons rien inventé.

Afin de créer une certaines identité propre au jeu Kaaptur, nous pourrions recréer tout ce cadre, et donc recréer certains médias qui ont été « empruntés » : les répliques des personnages, certains graphismes, les sons…

Pour augmenter encore l’idée d’identité, nous pourrions aussi rajouter de nombreux médias, comme des aperçus détaillés des pions, de la musique, d’autres sons, etc…

## Le Contenu

Mais la première idée d’amélioration qui nous était venue était de rajouter du contenu et de nouvelles règles pour le jeu.

Parmi les idées principales on retrouve :

* Ajout de nouveaux types de pions (magiciens, cavaliers, archers, etc…)
* Création de Bâtiments (Caserne, Forteresse, Tour de Guet, etc…)
* Combats Meurtriers avec points de vie et points d’attaque en plus
* Possibilité de créer des pions
* Possibilité d’améliorer ses pions
* Ajout de nouveaux items (armes, or, trésors, objets magiques)
* Ajouter une dimension temps réel (*hypothétique*)
* Ajouts de Pouvoirs Magiques, et de Dieux
* Création d’une histoire, avec des quêtes
* Plusieurs Modes : Quête, Roi vs. Roi (Défense du Roi), Exploration, Survie

Evidemment, ces idées ne sont que des idées, et ne sont pas forcément bonnes à implémenter. En effet il serait peut-être plus simple de revoir totalement le système du jeu (tour par tour), et de retaper ainsi tout le code. Mais ces idées restent néanmoins intéressantes, et peuvent toujours être mises en application après la fin du projet, celui-ci n’étant pas forcé de rester figé, fini.

C’est d’ailleurs pour cette raison que nous distribuons les sources de manière libre, à condition de nous citer lors d’une distribution (non-commerciale) ou d’une modification. On optera probablement pour une licence GPL par la suite.

Bilan du Projet

Kaaptur était un projet qui semblait extrêmement ludique, mais notre interprétation du sujet l’a rendu très ambitieux. Ainsi, les leçons à tirer de ce projet sont nombreuses, et celui-ci a été un excellent moyen de nous exercer et de progresser.

## Nos Acquis

### La SDL

Le plus grand gain de ce projet se situe au niveau des acquis. En effet, c’est la première fois pour tous deux que nous faisons un jeu complet en 2D. Ceci a notamment été possible par l’apprentissage de la SDL.

La maitrise de la SDL est venue petit à petit au cours du projet, et celui-ci nous a permis de nous faire les armes dessus : on peut considérer que l’on connait beaucoup mieux les failles de la SDL, et les capacités de la SDL (cf. Améliorations -> L’Optimisation). On sait exactement de quoi elle est capable, et c’est un gros avantage pour de futurs projets : si un projet peut parfaitement se réaliser avec la SDL, alors nous saurons exactement comment le faire. Ou au contraire, on peut savoir à l’avance que la SDL n’est pas la plus adaptée.

### L’interface

Le second défi de ce projet était de savoir utiliser les fonctions 2D à bon escient, et les utiliser dans le but de créer une interface graphique (GUI). Et c’est ce que nous avons su faire de manière très correct.

Bien que le système d’interface ne soit pas tout à fait « standard », car de nombreuses fonctionnalités sont manquantes (notamment la possibilité de sélectionner du texte, de déplacer le curseur du texte, d’effectuer des glisser-déposer, de créer des véritables menus, et checkbox à choix unique), celui-ci permet de bénéficier des avantages graphiques, mais également d’un certain esthétisme.

### Le Réseau

Enfin, le plus gros atout de ce projet est qu’il nous a permis de nous initier au réseau, à l’échange de données entre différents programmes, mais également aux aléas des connexions.

Bien entendu, Kaaptur n’est que la première pierre pour bâtir nos compétences en réseau, mais c’est déjà une bonne étape.

Conclusion

Kaaptur est un jeu de capture de drapeau. On peut donc dire que c’est un jeu d’aventure, mais ça a été avant tout une aventure pour nous !

Ce projet nous a motivé, et on a voulu donner le meilleur de nous-mêmes, nous surpasser en créant un jeu intéressant, de bonne qualité, dans un temps très limité de un mois et 3 semaines. On s’est pleinement investis, et c’est ce qui ressort de notre travail qui est notre meilleure récompense.

Ce projet nous a apporté beaucoup, la 2D, les graphismes, le réseau, soit bien plus que les précédents projets, et il nous a fait vivre notre passion pour le jeu vidéo, et la programmation. Aujourd’hui le jeu est fini dans sa version de base, mais ne l’est pas tout à fait pour nous, c’est probablement pourquoi nous continuerons de l’améliorer, afin de nous l’approprier encore plus.

À présent ce nouveau projet terminé, nous avons hâte d’être au prochain, tant Kaaptur nous a apporté de plaisir, et peut-être même un peu de fierté, (et de fatigue hélas).

Enjoy the game !

*Loïc Belmonte et Rémi Jarasson*

Le Site du Projet

Afin de faire partager notre travail, nous avons publié le jeu et son code source sur un site internet, ouvert à tous. Les éventuelles mises à jour seront donc publiées à cette adresse, aussi longtemps que possible :



***http://projects.artistix.fr/kaaptur/***