Bilan du projet Master 1 : BERREZKER

Les étudiants de M1IF (ENS Lyon)

27 janvier 2005

Table des matières

Chapitre 1

Graphismes

Membres : Alexis Ballier, Julien Robert, Pierre Clairambault et Benoit Boissinot.

1.1 Introduction.

Ce sous-module "graphismes" porte mal son nom, en effet, on aurait pu l'appeller "client" car il avait pour but de créer un client capable d'afficher un jeu, d'où son premier nom. Cependant, au départ, ce module ne visait pas à créer tout ce qui allait faire que le jeu serait "beau" et jouable...

Nous allons donc expliquer ici le fonctionnement du client tout en détaillant les problèmes rencontrés.

1.2 La librairie : ClanLib.

Le client est basé sur la librairie ClanLib (www.clanlib.org) qui fournit une API pour programmer des jeux (ou en tous cas, c'est le but voulu par les auteurs de cette librairie), ainsi, elle permet de gérer les graphismes, le réseau, la gestion de fichiers au format xml, etc. Cette librairie peut utiliser SDL ou OpenGL pour l'affichage, ce qui en fait donc des dépendances.

Pour BerRezKer, nous utilisons OpenGL, et il s'avère qu'à l'usage il vaut mieux avoir une carte accélératrice qui gère l'OpenGL en hardware pour que ça soit jouable.

Cette librairie est la pièce maitresse du client, nous reviendrons dessus à plusieurs reprises.

1.3 Architecture générale du client.

La première classe du client, celle qui contient la fonction "main" se trouve dans les fichiers client/berrezkerclient.{cc,h}. C'est dans ces fichiers que sont

faites toutes les initialisations et instanciations nécessaires pour que la librairie ClanLib fonctionne.

Après ceci, le programme crée une classe "game" (game/game.{cc,h}) et appelle en boucle sa fonction "mainloop", en récupérant la valeur renvoyée par cette fonction. Si cette valeur vaut 0, on détruit la classe "game" créée et on effectue les desinitialisations de ClanLib avant de terminer le programme.

Ce procédé permet d'avoir une classe qui ne contiendra que le code propre au client, qui ne devra pas se soucier des formalités dues à ClanLib, ni de boucler vu que sa fonction "mainloop" est appellée en boucle (d'où son nom...)

La classe "game" dont nous venons de parler crée à son instanciation une fenêtre dont les paramètres sont définis dans des ressources ClanLib (Nous reviendrons sur les ressources.) ; cette fenêtre servira à l'affichage, c'est ici que tout va être dessiné. "game" initialise aussi la connection réseau avec le serveur, les paramètres étant eux aussi contenus dans une ressource ClanLib (On peut noter que, s'il n'y a pas de reseau, c'est à dire que les paramètres sont mauvais ou que le serveur ne marche pas, le programme est quitté brutalement car ça ne sert à rien d'aller plus loin).

Cette classe instancie aussi ce que nous avons appellé le "maitre du temps", et une classe appellée "interface" dont nous allons parler bientot.

La fonction "mainloop" de la classe "game" sert à effectuer tout ce qu'il est nécessaire de faire avant de pouvoir afficher quelque chose. C'est à dire, effacer l'écran, appeller interface->mainloop pour qu'il se charge de l'affichage sprite par sprite, échanger les buffers.

On renvoie comme valeur de mainloop la valeur renvoyée par `mon_interface->mainloop`, donc en fait c'est interface qui décide quand le programme doit terminer.

1.3.1 Le double buffering.

On est ici en double buffering pour éviter tous les effets de scintillement désagréables. Ce procédé est classique en programmation de jeux pour éviter de changer une image pendant le balayage de l'écran et ainsi avoir, de temps en temps, une demi-image calculée au temps t et une demi-image calculée au temps $t+1$; s'il y a beaucoup d'animations, ce phénomène produit des scintillements désagréables. Ici, on change seulement le pointeur de la zone mémoire où la carte vidéo prend les données à envoyer à l'écran pendant le retour de balayage de l'écran, ainsi on s'assure d'avoir une unique image par balayage.

1.3.2 Le "maitre du temps".

Ce que nous avons appellé le "maitre du temps" est en fait un thread géré par ClanLib qui émet un certain signal à intervalles réguliers, réglable.

On a branché ce signal sur la plupart des fonctions "update" des objets créés, ainsi chaque objet a une notion de temps et peut, par exemple, s'afficher, s'animer, se déplacer indépendament de la vitesse de l'ordinateur sur lequel tourne le client.

Nous avons mis un pointeur ("extern") vers le signal généré par le maitre du temps dans partage/partage.h, inclus par tous les fichiers pour avoir acces aux variables qui doivent être communes à tous les objets. Chaque objet ayant acces à ce signal, on peut le "brancher" facilement sur chaque fonction dans le constructeur des objets grâce aux fonctions que ClanLib met à notre disposition.

1.3.3 Les ressources ClanLib.

Les ressources ClanLib nous ont beaucoup servi lors de la réalisation du client. Le principe en lui même est intéressant car il permet de charger des constantes dynamiquement via des fichiers xml, appellés fichiers de ressources.

Ainsi, en créant un "CL_ResourceManager" (que l'on nommera "main") à partir d'un fichier de ressources, on obtient par exemple l'entier "resolution_x" repéré, dans le fichier de ressources, par :

```
<integer name="resolution_x" value="512"/>
```

en appellant la fonction :

```
CL_Integer resolution_x("resolution_x", &main);
```

On peut donc charger des constantes via des fichiers de ressources facilement, ce qui permet d'éviter de devoir recompiler tout le programme à chaque fois que l'on veut changer une constante.

1.3.4 L'objet "interface".

Cet objet interface crée tout d'abord les menus, dont nous parlerons un peu plus tard.

Cet objet peut aussi créer l'objet "graphismes" à la demande des menus. L'objet "graphismes" et les menus sont tous deux des sous classes d'une classe abstraite "affichable".

C'est cet objet qui gère les évenements clavier / souris du jeu. Nous avons "branché" les signaux ClanLib émis lorsqu'un clic ou une touche est pressée / relachée sur des fonctions de cet objet. Cet objet se contente de faire passer les évennements à ses membres du type "affichable" par ordre de priorité, chaque membre répondant s'il a pris ou pas l'évennement. Si un membre a pris l'évennement, il n'est pas "propagé" aux autres membres de priorié inférieure. Ce procédé par exemple permet de désactiver le clavier et les clics souris pour le jeu lorsque le menu principal est affiché et ainsi éviter de voir son personnage se déplacer vers l'endroit cliqué alors que l'on voulait cliquer sur le menu.

Cet objet, dans sa fonction "mainloop" appelle les fonctions "show" de ses membres du type "affichable".

1.4 Les menus.

L'objectif des menus est de procurer une interface haut niveau entre les possibilités du jeu et l'utilisateur. Ils sont dans la version finale à un stade assez limité, c'est à dire qu'ils proposent un login à l'initialisation, puis un choix minimaliste d'options, comme démarrer ou quitter le jeu. C'est eux également qui prennent en charge l'entrée de l'irc accessible entre les joueurs.

Techniquement, ces menus sont réalisés par l'intermédiaire de ClanLib, qui fournit par ClanGUI des primitives (dont on regrette la documentation succinte) pour réaliser des GUIs, des interfaces graphiques utilisateur.

Initialement ces menus devaient être considérés par l'interface de façon abstraite comme des affichables, au même titre que les graphismes, et même sur un pied d'égalité avec eux. A l'initialisation, on créait tous ces objets et on ne les effacait qu'à la fin. Suivant les besoins, ils étaient à différents moments de l'exécution, soit actifs (auquel cas on les affiche, et on leur donne la possibilité d'intercepter les évènements clavier ou souris), soit inactifs (auquel cas ils étaient invisibles et inaccessibles). C'est ainsi que furent rencontrés quelques problèmes d'architecture, puisque par exemple le menu de login donne des informations indispensables à la création d'éléments bas niveau du jeu (comme la connexion irc et réseau), et doit donc être initialisé avant ceux là. Mais ces menus étaient sur un pied d'égalité avec des éléments du graphisme, qui devaient être créés avant et qui dépendaient du login...

Comme réponse à ce problème de dépendances, deux niveaux de menus ont été implémentés :

- les *menus d'initialisation* (en pratique l'écran de login), qui tournent sous leur propre GUIManager, et qui précèdent toute autre initialisation. Ils sont détruits dès qu'ils cessent d'être nécessaires.
- les *menus permanents*, qui sont gérés par le gestionnaire d'interface généraliste, et qui sont conservés et appellés au besoin pendant toute l'exécution du programme.

Ce système se montre efficace au point où en sont les menus et est facilement extensible, tel fut en fait le critère qu'on avait en vue lors de leur développement ; on avait pas l'ambition de faire un jeu avec un gameplay comparable aux jeux commerciaux mais pour être cohérents il fallait tenter de laisser l'ouverture nécessaire à un tel développement.

1.5 L'objet "graphismes".

L'objet graphismes est celui où l'on commence à avoir différents types d'objets à afficher : La carte, stockée dans un tableau, cf la section carte et surtout la partie "Level Design" et les objets graphiques, stockés dans une liste avec leurs coordonnées telles les matrices creuses, cf la section adéquate.

1.5.1 La profondeur d'affichage.

Pour pallier aux problèmes d'un personnage qui passe sous un pont, derrière un mat, évoqués lors des réunions, nous avons choisi d'avoir plusieurs niveaux d'affichage.

Ce qui veut dire que l'on a une profondeur d'affichage z, et que tout ce qui doit être affiché est dans un des niveaux d'affichage compris entre 0 et z. Lors de l'affichage on affiche les "sprites" de la couche 0 puis 1, etc, jusqu'à z.

Ainsi, les "sprites" de profondeur inférieure ont plus de chances d'être recouverts par ceux de hauteur supérieure, par exemple, en mettant le joueur à la hauteur 1 et le mât d'un bateau à la hauteur 2, le joueur passera bien dessous le mât. Pour le pont, c'est légèrement plus complexe, mais il suffit de mettre le pont à la hauteur 2 par exemple, lorsque le joueur est dans la vallée sous le pont, il est à la hauteur 1 et passe dessous, s'il est sur une des montagnes reliées par le pont, il doit être à la hauteur 3.

Ce concept de profondeur d'affichage résoud bien les problèmes de hauteur (la 3ème dimension) pour un jeu en deux dimensions.

La profondeur d'affichage est demandée au serveur via un "NetStream" fourni par le module réseau.

1.6 La carte.

Pour ce qui est de la carte, nous avons fourni un prototype de cartes stockées dans un tableau à 3 dimensions (la 3ème servant à la profondeur d'affichage) où chaque case du tableau contenait un entier indentifiant un "sprite" carré chargé via les ressources ClanLib.

Ainsi, cela devenait simple pour l'affichage, il suffisait de connaitre l'intervalle de la carte à afficher et à afficher les sprites correspondant aux cases voulues.

Par la suite, le module "Level Design" a largement modifié et amélioré ce modèle basique tout en restant compatible avec le prototype, pour cela il faut se réferrer à la section appropriée.

1.7 Les objets "graphiques"

Ces objets ont pour but d'être les représentations de tout ce qui ne sera pas interactif ni mobile dans le jeu (personnages, projectiles, etc.) qui seront partagés par tous les clients et par le serveur. Dans la partie "graphismes" nous nous sommes interessés aux éléments qui peuvent avoir un rapport avec l'affichage des objets. En premier lieu, il y a donc leur affichage, puis la gestion des collisions entre différents objets (par exemple quand un monstre veut se déplacer, il ne faut pas qu'un autre objet soit sur son chemin).

1.7.1 L'affichage et l'animation.

La partie affichage des objets est faite grâce à ClanLib(en particulier, on ne s'interesse absolument pas de savoir comment est géré l'affichage, on envoie simplement l'ordre d'afficher un sprite à certaine coordonnées). Ainsi, on a créé des fichiers de ressources à partir desquels on crée des gestionnaires de ressources qui vont enntièrement gérer la mise en mémoire des images, et donc permettre d'accélérer considérablement la création d'objets graphiques (à chaque fois qu'on cré un objet graphique, on ne recharge pas l'image -ou les images- en mémoire). Le gestionnaire de ressources contient tous les sprites dont l'objet peut avoir besoin (plusieurs sprites par directions car l'objet est animé).

On a donc un tableau de gestionnaires de ressources dont chaque élément correspond au gestionnaire de ressources d'un type d'objet donné, les objets du même type partageant leur gestionnaire.

Lorsqu'un objet avance, il faut mettre à jours les sprites, ainsi, le fichier xml contient en plus un champ "speed" qui correspond, s'il est positif au nombre de pixel à parcourir pour que le sprite change, et s'il est négatif, le nombre de tops jeu à attendre avant de changer de sprite (en effet, lorsqu'une action de l'objet n'est pas une action de déplacement, il faut quand même mettre à jours les sprites).

Les tops jeu sont générés par le maitre du temps décrit plus haut.

La gestion des animations nous a amenés à créer un systèmes d'états : chaque objet est dans l'état "Pause","Marche" ou "Attaque", bien sûr on peut imaginer d'autres états, et pour cela il n'y aurait rien à changer dans le code, tout se fait dans des fichiers de configuration. Chaque état correspond à un ensemble de sprites différents (8 ensembles correspondants aux 8 directions).

Pour que les objets graphiques puissent être uilisés par les autres modules, on a fourni des méthodes "goto" (relatif ou absolu) et change state, ce qui nous permet de définir l'angle et la vitesse avec lesquels l'objet va se déplacer, et donc de choisir le sprite à afficher.

1.7.2 La synchronisation de ces objets via le réseau.

Chaque objet a un "ClientObject" associé, créé à partir du "NetObject" associé à l'objet sur le serveur. Les "ClientObject" émettent un signal lorsqu'ils ont été mis à jour par le serveur, et nous branchons ce signal sur la fonction "net_update" de l'objet graphique associé.

Lorsqu'un objet graphique reçoit ce signal, il peut ou pas "appartenir" au client (typiquement le joueur appartient au client et les monstres au serveur) ; s'il n'appartient pas au client, il synchronise toutes les données nécessaires avec la mise à jour reçue. S'il appartient au client il ignore la mise à jour puisque c'est à lui de faire les mises à jour.

Les "données nécessaires" sont ici l'état, décrit plus haut, la position de l'objet, et sa position visée (ie son "goto") et sa vitesse de déplacement.

Pour ce qui est de l'envoi des mises à jour, en théorie nous pouvons nous contenter de n'envoyer des mises à jour que lorsqu'il y a un changement autre que

les changements réguliers de position dus aux déplacements, mais en pratique cela ne marche pas.

Cette idée semble pourtant très intéressante et à creuser, puisque, les clients et le serveur ayant un "maitre du temps" séparé mais étant censé emettre des signaux aux mêmes intervalles réguliers, on peut déplacer les objets indépendament sur chaque client et sur le serveur, étant donné que les déplacements dépendent de la direction et de la vitesse et peuvent donc être faits sans avoir besoin de se synchroniser à chaque fois. Le but était de n'envoyer des mises à jour qu'à des intervalles de "taille humaine" (ie à chaque clic de souris) plutot qu'à chaque top émis par le "maitre du temps" (ie beaucoup plus souvent). En pratique, il s'est avéré que cela rendait le jeu trop saccadé.

1.7.3 Les collisions.

Le module graphisme s'est aussi occupé de gérer les collisions entre objets, ce qui est utilisé par exemple lorsqu'un monstre essaie de taper le joueur : il teste si le joueur est suffisamment proche, et ce test se fait au niveau "pixel" pour que le joueur ait l'impression que monstre ne frappe pas dans le vide.

ClanLibproposait une gestion des collisions, on l'a donc utilisée, pour cela on a créé pour chaque objet une image représentant son champ d'action. ClanLibcalcule alors un ensemble de segments délimitant l'image (collision outline) ainsi que le rayon du polygone alors créé.

Ainsi, on peut demander à ClanLibsi un objet intersecte un autre et ceci se fait assez rapidement (si un objet est loin d'un autre seul des tests sur les rayons sont effectués et donc pour la plupart des tests d'intersection, une seule opération est faite). On a éprouvé des difficultés avec la librairie, car au début, on souhaitait avoir un "collision outline" par sprite, mais la façon dont ClanLibgère le gestionnaire de ressources ne le permettait pas (après discussion sur la liste de diffusion de ClanLib, on a été convaincu que c'était pour le moment impossible).

Pour ce qui est des tests de collisions, nous lançons un thread qui testera en boucle quelles sont les collisions pour ne pas faire de boucle trop inutile, nous faisons un "sleep" après chaque test de tous les objets. Lorsqu'une collision est détectée, le client envoie les identifiants uniques des deux objets en collision au serveur via un "NetStream" fourni par le module réseau, le serveur se chargeant d'effectuer les opérations nécessaires lors d'une collision.

1.7.4 Le gestionnaire d'objets.

C'est en fait lui qui gèrera tous les objets graphiques via des vecteurs de vecteurs dans le BerRezKer, mais cette implémentation est très souple.

Ce gestionnaire se charge de lancer le thread de détection des collisions vu que c'est le seul à avoir un accès direct à tous les objets. Il se charge aussi de créer les objets lorsqu'il reçoit une update provenant d'un "NetObject" inconnu. C'est aussi ce gestionnaire qui "controle" les évenements clavier / souris destinés au joueur et fait les actions appropriées.

1.8 L'irc.

L'irc est un objet graphique à part, car il est à la fois un objet graphique affiché par le gestionnaire, son entrée est gérée par les menus, et l'envoi et la réception de messages se fait par le prototype d'irc fourni par le module réseau.

Ainsi, le menu chargé de l'irc est une "input box" où l'on peut écrire du texte et que l'on peut activer ou désactiver. Quand on entre une phrase destinée à l'irc, il l'envoie au serveur irc et à l'objet graphique chargé de l'irc.

L'objet graphique irc est chargé de l'affichage des phrases à l'écran, et est ainsi branché au signal de réception de message, traitant et affichant chaque message entrant.

1.9 Bilan du module.

Pour conclure, nous pensons avoir rempli les objectifs fixés par ce module aux vues des réalisations de l'équipe.

Le travail d'équipe a bien fonctionné en règle générale, nous pensons être restés cohérents tout au long du semestre et ne pas avoir eu à faire de grosses modifications à cause d'erreurs de conception, sauf peut-être pour intégrer une architecture client/serveur sur la fin. Les réunions de projets où nous exposions nos problèmes et proposions des solutions n'y sont surement pas pour rien.

Chapitre 2

Game Design

Codeurs : Sergueï Lenglet, Romain Demangeon, Yann Strozecki.

Le rôle de la partie game design a été de fournir les éléments qui permettent à Berrezker d'être ludique, agréable et intéressant. Il faut façonner un univers cohérent, proposer au joueur des défis et des personnalisations qui le plonge dans un autre univers. Le Game Design comporte entre autres, les aspects de gestion de l'évolution du personnage, de ses caractéristiques, des objets qu'il pourra utiliser. Il comprend aussi l'élaboration des monstres, de leur Intelligence Artificielle, de leur pathfinding. Enfin les aspects background, quêtes, scénarios sont aussi des tâches pour le Game Design.

2.1 L'intelligence artificielle des monstres

Une des principales tâches du Game Design était d'animer les personnages non joueur qui peuplaient le jeu. L'attention a été portée sur les personnages non joueur hostiles (les monstres), sans lesquels il n'y aurait pas de jeu ...

2.1.1 Principe

A la manière des graphismes des objets, l'action effectuée par chaque monstre se fait top par top, et est donc mise à jour par le "maître du temps".

Chaque mouvement possible (déplacement d'une case à une autre, mouvement d'attaque ...) se fait donc en un nombre k prédéfini de tops, qui est une des caractéristiques du monstre.

A un top donné, le monstre est donc soit en train d'effectuer une action élémentaire ($\frac{1}{k}$ de mouvement), soit en attente, i.e. il est en train de calculer le prochain mouvement à effectuer. En état d'attente, le monstre détermine d'abord s'il a un joueur dans son champ de vision : s'il n'y en a pas, il reste en attente pour le prochain top. Sinon, on a deux possibilités :

- Soit le joueur vu est à portée d'attaque, et dans ce cas le monstre initie le mouvement d'attaque

- Sinon, il doit se rapprocher, il calcule un chemin via l'algorithme de pathfinding (présenté dans la section suivante), et initie le mouvement de déplacement vers la première case du chemin calculé.

On peut imaginer d'autres comportements, comme par exemple la fuite : si le monstre est dans un état critique, il peut chercher à s'éloigner du joueur.

A la fin d'un mouvement, le monstre se remet en état d'attente, sauf s'il est en train de se déplacer de cases en cases : en effet, pour des raisons de performances, on ne peut appeler l'algorithme de pathfinding après chaque déplacement d'une case. Cependant, dans la mesure où le personnage joueur bouge, il est nécessaire de recalculer le chemin à intervalles réguliers.

2.1.2 Exemple

Voyons un exemple de comportement de monstre en cours de jeu, sachant qu'un déplacement nécessite k tops et un mouvement d'attaque k' tops. On suppose que l'on recalcule le chemin toutes les 3 cases.

Au départ, le monstre est en attente.

- Top 0 : le monstre voit un personnage joueur en (x, y), il doit se rapprocher. Il calcule le chemin, et se prépare à aller vers la première case c_1 calculée.
- Top k : le monstre est en c_1, se pépare à aller en c_2.
- Top $2k$: le monstre est en c_2, se pépare à aller en c_3.
- Top $3k$: le monstre est en c_3, se met en attente.
- Top $3k + 1$: le joueur est toujours en vue et hors de portée. Un nouveau chemin est calculé.

⋮

- Top $6k + 2$: le monstre est en attente. Il voit toujours le joueur, et il est à portée. Il initie le mouvement d'attaque.
- Top $6k + 2 + k'$: le mouvement d'attaque est terminé et touche le joueur. Le monstre se met en attente.
- Top $6k + k' + 3$: le joueur est toujours dans le champs de vision, et à portée. Le monstre initie le mouvement d'attaque.

⋮

2.1.3 Quelques détails

Le champ de vision

Le champ de vision est dans la version actuelle du jeu très simple, dans la mesure où l'on teste juste si la distance entre le joueur et le monstre est inférieure à une distance $vision$, paramètre du monstre. Le champ de vision est donc un disque de centre le monstre de rayon $vision$, quels que soient les obstacles présents sur la carte.

Une amélioration a été proposée mais non implémentée faute de temps : on vérifiait en plus de la distance que le segment de droite défini par le monstre et le joueur ne passait que par des cases permettant de voir. Par exemple, il

est impossible de voir à travers un case d'arbres, en revanche il est possible de voir à travers une case d'eau. Cela nécessite que chaque case contienne une information en plus.

Enfin dans le cas où plusieurs joueurs sont dans le champ de vision du monstre, celui-ci choisit de s'attaquer au plus proche.

La portée d'attaque

Dans le cas des monstres qui attaquent à distance, les vérifications faites pour savoir si un joueur est à portée sont les mêmes que dans le cas du champ de vision : un simple test de distance. La même amélioration est possible (et également non implémentée ...).

En revanche, dans le cas des monstres qui attaquent au corps à corps, il est nécessaire de s'appuyer sur les collisions au niveau des graphismes pour obtenir quelque chose de cohérent à l'œil. Le monstre ne fait donc que se diriger en direction du joueur, et lorsqu'il y a collision, il initie son mouvement d'attaque.

La probabilité de toucher

Les monstres comme les joueurs ne voient pas leurs coups faire mouche à chaque fois. Cependant un joueur "puissant" devrait toucher facilement un monstre faible, alors qu'un joueur qui vient de commencer le jeu doit avoir plus de mal.

Lors d'un combat, il faut donc prendre en compte la défense et le niveau d'attaque (capacité à porter des coups) des deux protagonistes, mais également leur niveau respectif : le niveau d'un joueur est déterminé par l'expérience accumulée, celui d'un monstre est une caractéristique fixe de celui-ci. Une formule prenant en compte ces divers paramètres a été mise au point par tests successifs.

2.2 Le pathfinding

Utile au monstre comme au joueur lors de déplacement à la souris, un algorithme de pathfinding a dû être utilisé. On cherche dans les deux cas à se rendre d'un point A à point un point B avec un chemin de coût minimum, le coût restant un paramêtre à notre convenance (poids sur les cases, ...). L'algorithme choisi est des plus utilisés dans le monde du jeu, l'algorithme A*.

2.2.1 Présentation de l'algorithme

A la manière de l'algorithme de Dijkstra, l'algorithme A* parcourt les chemins possibles progressivement, mais en prenant en compte en plus du poids de la case, une estimation de la distance pour atteindre le point de destination. Alors que dans l'algorithme de Dijkstra, toutes les directions sont équivalentes lorsqu'on construit les chemins, avec A*, une direction est privilégiée.

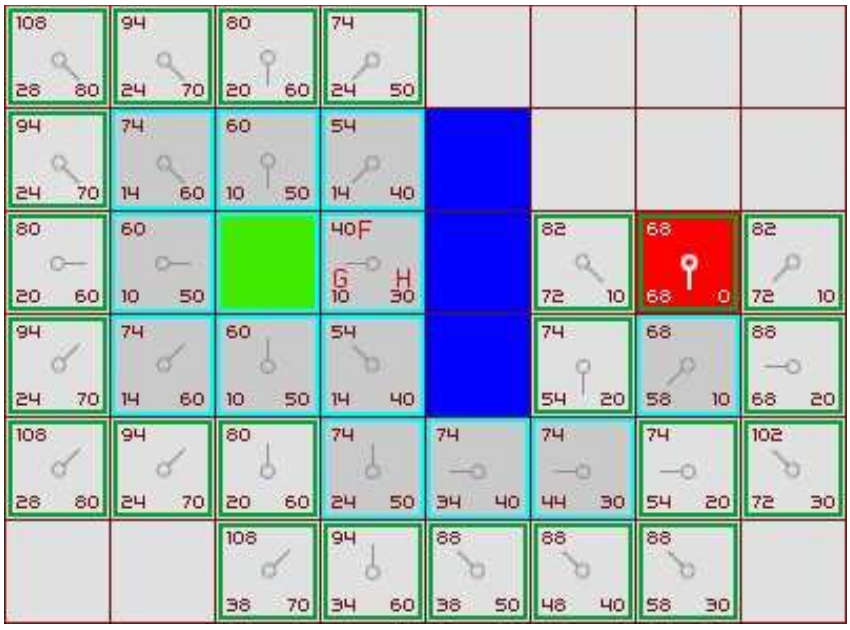

FIG. 2.1 – Exemple de mise en œuvre de l'algorithme A*

2.2.2 Principe de fonctionnement

On dispose de deux listes : la liste ouverte et la liste fermée. La première contient les dernières cases des chemins calculés et la seconde contient les cases déjà évaluées.

On commence par considérer toutes les cases adjacentes à la case de départ et accessibles pour leur attribuer un coût qui correspond au poids du chemin qu'il faut parcourir pour les atteindre plus le poids estimé du chemin qui reste à parcourir jusqu'à la destination.On rajoute ces cases à la liste ouverte avec comme attribut leur parent, c'est à dire leur case de départ, et leur coût. On ajoute la case de départ à la liste fermée et on l'enlève de la liste ouverte.

On réitère cette opération en prenant comme case de départ une de cases de coût minimal de la liste ouverte, qu'on retire de celle-ci pour l'ajouter à la liste fermée. On ajoute ensuite à la liste ouverte les cases adjacentes, si elles n'y sont pas, et on modifie le coût de celles qui étaient déjà présente si on trouve un coût inférieur. On s'arrête quand la case d'arrivée est ajoutée à la liste ouverte ou que la liste ouverte est vide (échec).

Pour reconstruire le chemin, il suffit de considérer les pères successifs.

2.2.3 Exemple

Sur la figure 2.1, la case verte est le point de départ, la case rouge est le point d'arrivée et les cases bleues représentes un obstacle. Les cases encadrées en vert ont été ajoutées à la liste fermée, les cases encadrées en bleu ciel à la liste ouverte. Sur chaque case sont incrits 3 nombres : en bas à gauche, le poids du chemin pour atteindre cette case, en bas à droite, le poids estimé pour atteindre la case d'arrivée, en haut à gauche, le poids total. L'orientation de la loupe permet de determiner le père de chaque case.

2.2.4 Implémentation

L'algorithme nécessite que les cases soit marquées accessibles ou non. Aucun poids autre que la distance n'a été utilisé vu que les cases n'induisent pas de pénalités de déplacement.

Une librairie C++ Gnu/GPL a été utilisée et légèrement modifiée pour s'adapter au jeu. Dans le cas où aucun chemin n'est trouvé, l'entité (monstre ou joueur) qui cherchait à se deplacer ne bouge pas.

2.3 Background

La création d'un background cohérent n'est pas extrêmement difficile, mais elle mérite qu'on y attache de l'importance. En effet, elle a pour but de créer le monde-image dans lequel le joueur va être transposé, par le biais de son avatar. Sans background, le jeu n'est rein de plus pour le joueur qu'un simple programme, élément du monde réel, dénué de toute ouverture vers le rêve. Après une concertation parmi tous les membres du projets, un consensus autour de l'ambiance heroic-fantasy a été trouvé. Cette ambiance permet de placer l'avatar du joueur dans une situation de combat au corps a corps avec les monstres, mettant l'action au coeur même du jeu et de sa représentation, à l'opposé d'une ambiance futuriste, où les armes à distance et les échanges de tirs rendent les combats moins épiques.

Un premier background a été créé, assez classique, explicitant par la naissance d'un conflit global, l'ambiance hostile qui regne dans le jeu. Il est à noter que c'est le background qui s'est adapté au jeu et non l'inverse. Il a fallu, dans les premières versions, trouver une histoire et une justification aux moutons et au dinosaures qui se livraient une lutte sans fin. Il a fallu inventer plusieurs civilisation, correspondant aux ambiances variées que proposait le jeu, plaine, forête, cryptes.

2.4 Personnages, Caractéristiques et Trésors

La création de personnages, de leur caractéristiques évolutives et des trésors dont ils peuvent s'équiper sont un aspect important des jeu de roles électroniques modernes. Un système de caractéristiques simple a été créé, ainsi que

les divers attributs en résultant. Par exemple, un joueur décidant de faire progresser sa force augmentera les dégats effectués en mélée. Des formules et des tableaux ont du être mis au point et testés, pour ne pas déséquilibré le jeu. Malheureusement, à l'instar des trésors, ces fonctionnalités n'ont finalement pas pu être implémentées dans la version finale, faute de temps.

Chapitre 3

Etat de l'art

Membre : Romain Demangeon.

Le sous-projet Etat de l'art a eu comme tâche l'étude et la présentation de ce qui s'est fait dans les domaines respectifs du hack'n'slash et du MMORPG depuis les dernières années. Cette étude permet de mieux cibler ce à quoi Berrezker doit ressembler, d'être confronté aux erruers à ne pas commettre et de s'imprégner des grandes lignes qui dirigent les genres de jeu. Il a aboutit à une présentation publique lors d'une réunion d'un exposé relatant les caractéristiques des deux genre sus-cités, ainsi qu'à une revue non exhaustive des différents titres les plus joués.

Chapitre 4

Level Design

Codeurs : Laurent Jouhet, Aurélien Pardon, Laurent Braud, Rémi Brochenin.

Le but de l'équipe *Level Design* est de fournir un ensemble d'outils et de ressources permettant de créer l'environnement du jeu *BerRezKer*. Cela va de la création de cartes utilisées pour le terrain sur lequel vont évoluer les personnages du jeu, jusqu'à la création des sprites (images) utilisés, et à leur affichage. Les membres du module *Level Design* sont Aurélien Pardon, Laurent Braud, Rémi Brochenin, sous la responsabilité de Laurent Jouhet.

4.1 La génération de cartes : Aurélien Pardon

Dès le début du projet, l'utilisation d'un générateur de cartes aléatoires à été évoqué car il faut pouvoir fournir au jeu un nombre potentiellement illimité de cartes. On se déplace de cartes en cartes à travers des portails ; les cartes crées restent définitivement sur le serveur. Le fait de pouvoir choisir des tailles différentes pour les cartes est également intéressant.

Un des soucis des cartes aléatoires est peut-être leur manque de cohérence : il fallait trouver un algorithme qui permettent d'avoir des cartes réalistes avec des forets, des étangs, etc : c'est Aurélien Pardon qui a réalisé ce travail.

L'algorithme de génération de carte, réalisé par Aurélien Pardon, est plutôt simple. Il ressemble à une sorte de développement (bio)logique. Le terrain sera divisé en cases (de taille arbitraire 64 x 64) et chacune de ces cases aura un type (herbe, eau, sable, pavé, etc.). Chaque case va imposer des contraintes de développement sur les cases alentours : un principe d'actions sur l'environnement proche. La carte au début sera uniquement composée de cases vides et d'un nombre petit de cases pleines (deux ou trois), ces cases pleines ayant un type bien défini. Chaque case va proposer un type de cases pour ses cases voisines vides ; la proposition étant définie comme une probabilité d'être d'un certain type. Certaines cases vides se voient donc attribuer des probabilités de type de cases : on fait alors un tirage et on détermine définitivement le type de ces cases.

On recommence le processus, ces nouvelles cases vont influer sur la déterminisation de leurs cases voisines. Quand plus aucune case n'influence une autre, on arrête ; cela arrive forcément car les cases sont déterminisée définitivement. Après cette étape, on applique un lissage de la carte pour rajouter un peu de cohérence et d'esthétisme. Par exemple, si une case d'herbe et une case d'eau sont juxtaposées, la case d'herbe deviendra une interface herbe/eau lors de ce lissage. Enfin on associe à chaque case un sprite (une image) qui la représentera dans le jeu (cf 4.3).

Pour un souci d'évolutivité et de clarté, cette algorithme a été implémenté en OCaml car c'est un langage très pratique pour faire du prototypage. Un système de module, similaire à une approche orientée objet, permet d'enrichir facilement ce programme. Il suffit de fournir au logiciel le type de case et les affinités des cases entre elles (influence sur les voisins) pour qu'il fabrique une carte du type demandé : ainsi deux exemples réellement différents ont été programmés. Le premier permet de créer des cartes de plaines avec de l'herbe, de l'eau et des arbres. Il y a très peu de types de case, le type eau est assez travaillé et permet d'obtenir des étendues d'eau de taille variable. Les sprites sont eux très nombreux car il faut penser à toutes les interfaces eau/herbe possibles. L'autre module tente de créer une crypte. Cette fois si les types de cases sont nombreux et complexes mais les sprites sont simples : sol et mur uniquement. Il est intéressant de remarquer que cet algorithme s'applique bien pour des paysages naturels, ce qui est plutôt normal car il en est inspiré de ça : les arbres laissent tomber les graines à leurs alentours, ce qui fait des forêts est transcrit par une case arbre va fortement influencer les cases voisines à devenir de type arbre. Mais cet algorithme marche très bien pour quelque chose de plus artificiel comme des couloirs ou des pièces dans une donjon : tout dépend du type de case que l'on met et c'est la principale raison pour avoir choisi OCaml : les types définissables sont très riches et permettent de définir beaucoup de choses. On peut noter que OCaml est aussi très performant.

L'algorithme s'occupe également de créer les fichiers .map et .par représentant respectivement la carte proprement dite (tableau de nombre) et un fichier permettant d'associer à chaque nombre un ou plusieurs sprites.

Le code Caml est richement commenté, on peut regarder un peu le code pour bien comprendre le tout.

(cf. BERREZKER/devel/map/terrain/*.ml)

Pour plus de détail, on pourra regarder le code de 'terraform.ml' qui le moteur de base, ainsi que 'plain.ml' et 'crypt.ml' pour les détails des différents types d'environnements.

4.2 Intégration dans *BerRezKer* : Laurent Jouhet

L'intégration de la partie *Level Design* dans *BerRezKer* (cf. BERREZKER/devel/map/map.cc) a été réalisée par Laurent Jouhet et consiste en :

- la mise au point du format des fichiers '.par' et '.map' pour une plus grande souplesse d'utilisation par la suite,

- l'utilisation de ces fichiers pour trouver et charger les bons sprites,
- l'optimisation, en utilisant des pointeurs vers des tableaux de sprites, de l'occupation mémoire (facilité par l'utilisation des fichiers '.par')
- l'affichage des cartes.

Lors du chargement d'une carte, on lit le fichier '.map', qui est une matrice représentant les différents types de cases. Chaque type de case va en fait correspondre à plusieurs sprites, et donc pourra combiner plusieurs caractéristiques (comme l'accessibilité).

Cette correspondance se fait grâce au fichier '.par' qui pour chaque type de carte, donne une liste de sprites.

Les fichiers '.map' et '.par' sont stockés dans 'BERREZKER/devel/cartes/', avec le fichier 'sprites.xml'. Ce fichier est un fichier de ressource *ClanLib* qui permet de charger facilement les sprites situés dans 'BERREZKER/sprites/map'. Tous les sprites utilisés pour l'affichage des map sont des '.png' car ils posent moins de problème à l'affichage que les '.tga', et ils pourraient de plus servir pour les collisions.

L'affichage lancé à chaque top par l'interface graphique s'effectue par couches et est systématiquement centré sur le joueur. Il est effectué grâce aux directives fournies par *ClanLib*.

Enfin, le *Level Design* utilisant des tableux assez gros (sprites), un soin particulier a été apporté à la suppression des objets lors du déchargement des cartes : il est fait 'à la main' (avec des boucles imbriquées) au lieu d'utiliser les mécanismes automatiques, afin d'éviter au maximum les fuites mémoires, provoquées par l'utilisation de pointeurs vers des tableaux à l'intérieur d'autres tableaux.

De plus on a rajouté sur chaque carte deux portails servant à passer d'une carte à une autre.

Remarques :

- Les modifications éffectuées après installation du réseau pour lire les fichiers '.map' et '.par' directement à partir du serveur, et pas seulement en local ont été éffectuées par Benoit Boissinot, équipe *Reseau*.
- Les collisions joueurs-environnement ne sont pas gérées grâce aux Outlines ClanLib comme c'est le cas pour les collisions joueurs-joueurs car les déplacements des joueurs sont directement gérés par la fonction de pathfinding (Sergueï Lenglet et Romain Demangeon, équipe *Game Design*) et évitent donc naturellement les obstacles de l'environnement.
- Le changement de carte fonctionne simplement (clic droit) : il suffit d'appeler 'load(fichier)' pour tout mettre à jour sauf le pathfinding des monstres, mais il a été désactivé dans la version réseau car le serveur ne gère pas plusieurs environnements.

4.3 Création des sprites : Laurent Braud

Tous les sprites (cf. BERREZKER/devel/sprites/map), ainsi que les textures, ont été imaginés et dessinés par Laurent Braud, principalement sous *gimp*.

De plus un outil permettant de générer des sprites représentant la frontière entre deux types de terrains différents a été développé par Laurent Jouhet et Laurent Braud. (cf. BERREZKER/devel/map/textures/textpng.cc).

Il prend en entrée deux fichiers : un représentant le type de bordure (forme) que doit avoir la frontière, et un représentant la texture utilisée.

Il charge la forme de base (représentant une frontière horizontale avec la case du dessus), et crée à partir de cette forme un masque correspondant à le frontière spécifiée en ligne de commande. C'est la partie la plus importante de la génération des sprites : on applique au masque de départ des transformations affines (rotations, réflexions) ainsi que des min/max, pour obtenir un masque cohérent (pas de discontinuité).

Une fois le masque obtenu, on applique la texture, et on enregistre le fichier.

Il utilise *pngwriter* pour lire et écrire directement les fichiers '.png'. De plus, l'utilisation de *pngwriter* ne gérant pas la transparence, on utilisera *convert* (package *imagemagick*) grâce au script 'transparent.sh' pour l'obtenir.

4.4 Le placement des objets : Rémi Brochenin

Enfin, Rémi Brochenin a travaillé sur du code *caml*, à partir de celui de Aurélien Pardon, pour rajouter la possibilité d'insérer des objets dans les différents types d'environnements. Son algorithme utilise les cartes renvoyées par Aurélien, et essaye d'y insérer aléatoirement des objets (villages, coffres) selon leur taille et l'espace disponible sur la carte :

Les éléments de décor que l'on ne peut placer de façon totalement aléatoire sur la carte tels que les maisons (si l'on souhaite former des villages structurés) ou une partie de la carte nécessaire à une quête (le lieu où se trouve le méchant final entouré de ses sbires, ou autre) sont placés après l'exécution de l'automate cellulaire, par des "tuiles". Il s'agit donc d'un ensemble d'éléments de décor prédéfinis.

L'idée provient initialement d'une volonté de créer les cartes par la juxtaposition de tuiles (exclusivement, sans utilisation préliminaire d'un automate cellulaire). Cela serait possible en imposant des condition de juxtaposition plus complexes que celles décrites ci-dessous (par exemple imposer qu'un port se trouve à côté d'une étendue d'eau suffisamment grande, etc.).

Les "tuiles" que l'on souhaite placer sont divisées en deux ensembles : celles que l'on tient absolument à placer (c'est à dire qui doivent nécessairement se trouver sur la carte, comme le point de départ des joueurs) et celles que l'on aimerait placer en relativement grand nombre comme, par exemple, les coffres ou les monstres (le nombre exact n'est pas important, il est donc possible d'en placer un peu moins, et l'on cherche à les placer avec moins d'acharnement). Ces deux ensembles ainsi que la matrice fournie par l'automate cellulaire sont passés à la fonction principale, appelée "pose", qui renvoie une liste de noms de sprites avec leur emplacement ((int*int)*sprite). Les éléments de cette liste ont été placés s'il y avait assez de place libre, sans autre considération, si bien qu'il est possible que deux villages soient côte à côte (mais sans se chevaucher

cependant).

Pour les tuiles identiques dont le nombre est approximatif, on essaie seulement une fois de les poser ; et pour celles auxquelles on tient absolument, on énumère toutes les coordonnées possibles. Les deux ensembles passés à la fonction "pose" sont actuellement prédéfinis et simplistes (deux villages, des coffres, et des repaires de monstres, en attendant mieux).

Cette méthode n'est pas encore implémentée dans BerRezKer. (cf. BERREZKER/devel/map/tuiles/tuiles.ml)

Chapitre 5

Réseau

Codeurs : Benoit Boissinot, Alexis Ballier, Julien Robert et Nada Ahmidani.

5.1 Introduction.

Le but du module réseau est de fournir les méchanismes nécessaires aux communications entre les clients et le serveur. Entre autres : le login, le téléchargement des maps et l'update des objets sur tous les clients.

5.2 La librairie : ClanLib.

Le logiciel est basé sur la librairie ClanLib (`www.clanlib.org`) qui est une interface de haut niveau pour programmer des jeux. Elle nous permet de gérer les communications entre le client et le serveur.

Un point important de cette librairie est le fonctionnement par signaux, en effet on peut associer une fonction à un évenement donné (par exemple arrivée d'un paquet, ou deplacement de la souris) ce qui permet d'avoir un style de programmation évenementielle (et donc une programmation plus rapide et adaptée pour un jeu).

5.3 Architecture générale du réseau.

Tout d'abord il faut initier un session entre le serveur et les clients, cela se fait à l'aide de l'objet CL_NetSession. Grâce à ce méchanisme, le serveur sera notifié (par un signal) des connexions et deconnexions des clients (et inversement, les clients seront notifiés en cas de deconnexion du serveur).

Ensuite, on peut considérer deux sortes de communications dans le jeu :

- les communications synchrones (par exemple le login),
- et les communications asynchrones (les déplacements des objets).

Les communications synchrones se font à l'aide des NetStreams. En effet ils proposent un mode de communication orienté connection, ce qui nous a permis de mettre en place un méchanisme de login, c'est également utilisé pour les changements et les chargements de cartes ainsi que les collisions entre objets.

Les communications asynchrones se font grâce aux NetPackets lorsqu'il s'agit de communications de bas niveau . Sinon on utilise les NetObjects qui permettent de répliquer un objet entre plusieurs clients et le serveur. Les NetObjects sont détaillés dans le paragraphe suivant.

5.4 Les "Netobjects".

Les NetObjects permettent de répliquer un objet à travers le réseau. En effet on associe à chaque objet que l'on veut répliquer un NetObject. Pour envoyer une mise à jour, il faut construire un NetPacket qui sera envoyé à tout les clients qui possèdent ce NetObject, de leur côté les clients détermineront, à partir du paquet, l'action à entreprendre (déplacement, changement d'état, ...).

Un NetObject situé sur le serveur envoie un broadcast à tous les clients pour indiquer un déplacement ou la mort d'un objet, un NetObject situé sur un client envoie un message au serveur (qui s'occupera de le broadcaster si besoin) lorsque le joueur deplace son personnage.

5.5 La communication entre joueurs.

Les joueurs peuvent communiquer entre eux grâce au protocole IRC. En effet la bibliothèque Clanlib propose des fonctions de base pour integrer des communications par IRC. Au lancement du jeu, le client sur connecte sur un channel pre-déterminé et il peut communiquer avec les autres joueurs connectés. De plus les administrateurs du jeu peuvent communiquer avec les joueurs par ce biais en utilisant un client IRC classique.

5.6 Bilan du module.

L'implémentation des communications entre le client et le serveur a pu être faite relativement rapidement, les difficultés rencontrées ont principalement comme raison le manque de coordination entre modules. En effet, une interface commune n'a pas été définie dès le début du projet et les besoins des différents modules n'ont pas été exprimés suffisamment tôt. En bref le manque de coordination sur l'ensemble du projet s'est fait particulièrement ressentir dans ce module et a rendu l'implementation plus difficile qu'elle n'aurait dû l'être.

Annexe A

Mini How-To pour compiler et lancer BerRezKer

Auteurs : Alexis Ballier et Benoit Boissinot.

A.1 Dépendances.

BerRezKer nécessite :
- Un environnement de programmation 'viable' : gcc (testé avec le 3.3 et 3.4), pkgconfig, make, glibc, etc...
- ClanLib 0.7, testé avec clanlib-0.7.8.
- Pour le serveur : MySql en théorie, pour l'instant on pourrait se contenter d'enlever cette dépendance.
- Une carte accélératrice OpenGL est fortement recommandée.

A.2 Compilation.

Dans le répertoire devel/ du CVS :

```
make
```

compile le client.
Dans le répertoire devel/serveur du CVS :

```
make
```

compile le serveur minimal (sans SQL), nécessaire maintenant au fonctionnement de BerRezKer.

A.3 Lancement de BerRezKer.

Pour lancer le serveur :
Se mettre dans le répertoire devel/ du CVS et taper :

```
./bin/berrezkerd
```

Le serveur fournit une mini-console, il suffit de taper :

```
halt
```

dans cette console pour arreter le serveur proprement, mais "Ctrl-C" marche toujours.
Pour lancer le client :

```
./bin/berrezker
```

Les paramètres de connection réseau sont définis dans "devel/main.xml" si jamais on veut se connecter à un autre serveur que "localhost", ce qui parrait normal pour un jeu en réseau.
Ensuite, entrez un login et un mot de passe et appuyez sur "Create Login" pour créer un login ; Si votre login est déjà créé, appuyez simplement sur login après avoir rentré les bons login et mot de passe.

A.4 Installer ClanLib sous Gentoo.

En root :

```
emerge clanlib
```

Disponible sur les architectures : x86, ppc, sparc, alpha et amd64 en stable.

A.5 Installer ClanLib sous Debian.

Classique `./configure`, `make` et `make install`.