Département Informatique
Année Universitaire 2003/2004

PROJET TUTEURE

Deuxième Année

Tobruk 1941

Présenté le 31 mars 2004

Par

- OPFERMANN Cédric
- ARNAUD Jean
- VIAL Mickaël
- MILLET Aurélien
- HAKKOU Hicham

Jury
- Monsieur COSNIER Jean (Tuteur)
- Monsieur GEROT Cédric (Examinateur)

Table des matières

Table des matières 1

Introduction 2

I) Noyau de l'application 3

A) Présentation des règles 3
1) Terrain 3
2) Unités de combat 4
3) Le Général 4
4) Séquence de jeu 4
5) Zone de contrôle 4
6) Mouvement 5
7) Combat 5
8) Scénario historique 6
B) Schéma de l'application 7
1) Détail des classes 7
2) Structures de données 9
3) Diagramme de classes généralisé 9
4) Graphe des dépendances 11
5) Version alphanumérique 11
6) Conclusion et bilan 12
C) Intelligence Artificielle 13
1) Algorithme de meilleur chemin (« pathfinding ») 13
2) Déplacement et combat 15

II) Gestion de l'affichage 17

A) Création et initialisation de la fenêtre 17
1) Fonctions 17
2) Impression d'écran 18
3) Problèmes rencontrés 19
B) Mise en œuvre de l'OpenGL 20
1) Création et affichage de la carte 20
2) Mouvements de la caméra et des unités 22

Conclusion 24

Références documentaires 25

Outils utilisés 25

Répartition des tâches 26

Résumés 27

Annexes 28
Annexe 1 : Cahier des charges 28
Annexe 2 : Fichier Tobruk.txt 34
Annexe 3 : Code source de l'application 36

Introduction

Notre projet tuteuré consiste en la création d'un jeu de stratégie au tour par tour en 3D jouable au clavier et à la souris et tournant sous Windows. Il s'agit d'une simulation de la campagne de Tobruk (plus grand port britannique en Libye pendant la Seconde Guerre Mondiale) de mars à mai 1941. L'un des joueurs contrôle l'armée britannique et l'autre, les forces de l'Axe du Général Rommel. Le théâtre des opérations est représenté par une carte 40 par 20 cases, chacun des vingt tours de jeu représentant deux jours. Les effectifs engagés seront de trois types : l'infanterie, les blindés et les généraux. Compte tenu de la faible participation de l'aviation, celle-ci sera ignorée. De plus, la nature du terrain influencera les déplacements et les combats.

A un autre niveau de réalisation, il est à noter que notre application peut se rapprocher de jeux récemment édités, tels que « Combat Mission III : Afrika Korps » ou « Afrika Korps vs Desert Rats ». Ces jeux proposent respectivement une simulation tactique rigoureuse, une précision encyclopédique et un aspect relativement réaliste tout en restant simple d'accès. Ce sont ces différents points positifs que nous avons essayé d'intégrer tout au long du développement de notre application. Bien entendu il nous est impossible d'atteindre le rendu et la profondeur de jeux commerciaux mais nous avons essayé de rassembler et de programmer ces aspects de jeu à notre niveau.

Notre plan se divise en deux grandes parties : le noyau et l'interface. Chaque partie comprendra les méthodes de raisonnement ainsi que les difficultés rencontrées lors de leurs développements.

En tout premier lieu, il convient de présenter les règles du jeu car c'est sur celles-ci que nous avons basé la programmation de notre noyau. Ensuite, nous expliquerons la programmation de ce noyau, ayant découlé de l'étude des règles. Puis, nous verrons plus en détail les procédures de mouvement et de combat qui ont été implémentées pour venir se greffer sur le noyau présenté précédemment.

Après cela, dans une deuxième partie portant sur la gestion de l'affichage, nous commencerons par parler de la création et de l'initialisation d'une fenêtre Windows dans laquelle sera affiché l'écran de jeu, ainsi que des menus. Enfin, nous présenterons le fonctionnement du moteur 3D de notre application, qui affiche le jeu à l'écran et qui vient se positionner dans la fenêtre créée au préalable, avec la gestion de l'affichage de la carte, des unités et des mouvements de caméra.

I) Noyau de l'application

A) Présentation des règles

1) Terrain

Un carré sur la carte correspond à environ 10 km².
Voici les différents types de terrains représentés dans le jeu, ainsi que leurs conséquences sur les points de mouvement des unités (PM) :

Mer
Infranchissable

Montagne
4PM

Colline
2PM

Désert
2PM

Marais
2PM

Plaine
1PM

Ville
1PM

Voici la carte du jeu :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
A																																								
B																																								
C																																								
D																																								
E																																								
F																																								
G																																								
H																																								
I																																								
J																																								
K																																								
L																																								
M																																								
N																																								
O																																								
P																																								
Q																																								
R																																								
S																																								
T																																								

2) Unités de combat

Les unités d'infanterie et de blindés sont regroupées en formation selon leur camp. Chaque pion d'infanterie et de blindés possède un facteur d'arme légère, un facteur d'arme lourde, et un potentiel de mouvement. Suite à des pertes au combat, les unités peuvent perdre des points de combat.

Un pion d'infanterie représente un à dix régiments (à raison d'un point de combat par régiment), c'est-à-dire l'équivalent d'une division. Un pion de blindés représente un à cinq bataillons (à raison d'un point de combat par bataillon), c'est-à-dire l'équivalent d'une brigade. Chaque point de combat équivaut donc à 600-1000 hommes.

Les unités ne peuvent jamais se "diviser" en deux unités plus faibles, ni "regrouper" leurs forces en un seul pion plus fort, sauf si elles sont regroupées dans une même armée commandée par un général.

3) Le Général

Les généraux permettent de regrouper deux unités adjacentes sous une même bannière. Le général apporte à ses troupes un bonus en attaque de 5%. A noter que des régiments ne peuvent être réunis que s'ils n'ont pas effectué de mouvement durant le tour.

Une fois réunis, les régiments disparaissent pour former une seule armée et se déplacent simultanément, à la vitesse de l'unité la plus lente. Les facteurs d'armes légères, lourdes ainsi que les points de combat sont additionnés. En cas de pertes, les deux régiments se partagent les dommages.

Enfin, quelque soit l'état du général, il est soumis aux mêmes règles que les unités, bien qu'il ne puisse attaquer par lui même.

4) Séquence de jeu

Tobruk 1941 se joue en vingt tours, chaque tour de jeu comprenant une seule phase. Un tour de jeu comprend l'activation du joueur allemand, puis celle du joueur britannique. A chaque nouvelle activation, toutes les unités du camp deviennent activables. Les unités peuvent faire un mouvement (et un seul, même s'il leur reste des points de mouvement) et combattre. Les pions deviennent ainsi activés et le resteront jusqu'au prochain tour. Les unités activées peuvent uniquement se défendre.

5) Zone de contrôle

Toutes les unités exercent une zone de contrôle dans leurs carrés adjacents.

Propriétés des zones de contrôle :

- Une unité qui pénètre dans une zone de contrôle ennemie stoppe son mouvement.
- Une unité qui recule dans une zone de contrôle ennemie suite à un résultat de combat perd un point de combat.
- Il est interdit d’aller directement d’une zone de contrôle ennemie à une autre zone de contrôle ennemie.
- Sortir d'une zone de contrôle ennemie coûte un point de mouvement.

6) Mouvement

Une unité peut décider d'utiliser la totalité ou simplement une partie de son potentiel de mouvement.
L'entrée dans un carré coûte un certain nombre de points de mouvement à l'unité qui se déplace.
Les influences des types de terrain sont indiquées en I) A.
Les points de mouvement ne sont pas cumulables d'un tour à l'autre. Les règles d'empilement doivent être respectées durant le mouvement (pas d'interpénétration d'unité).

7) Combat

Les unités, pendant leur phase d'activation, peuvent attaquer une unité adverse adjacente située dans leur zone de contrôle. L'attaque n'est jamais obligatoire. Le joueur en phase d'activation est appelé l'attaquant, son adversaire est appelé le défenseur.

Leur orientation est indifférente, ils sont considérés toujours de front.

Choc

- Une unité ne peut attaquer qu'une fois par tour, quand son camp est en phase d'activation.
- Une unité ne peut engager qu'une seule unité adverse dans un combat.
- Une unité peut se faire attaquer plusieurs fois par tour.

Procédure de choc

On effectue le ratio de la force de combat de chaque unité. Celle-ci dépend de l 'unité adverse engagée. Si l'unité adverse est de type « infanterie », on prendra le facteur d'arme légère, si elle est de type « blindés », on prendra le facteur d'arme lourde.

On multiplie ensuite avec le nombre de points de combats de l'unité pour obtenir la force de combat. Le ratio est toujours arrondi à l'avantage du défenseur.

L'attaquant jette un dé à 6 faces et différents modificateurs sont ensuite additionnés à ce jet. On consulte alors la table de combat pour appliquer le résultat.

Modificateurs de combat

- Défenseur en montagne ou en ville : -2 au jet de l'attaquant.
- Défenseur en colline : -1 au jet de l'attaquant.
- Défenseur en plaine : +1 au jet de l'attaquant.
- Blindés contre défenseur en ville : -2 au jet de l'attaquant (cumulable avec défenseur en ville soit -4).

Résultats des chocs

1..5 : nombres de points de combat éliminés sur les unités impliquées.
Quand le total de points de combat d'une unité tombe à 0, cette unité est éliminée.

Recul des unités

Une unité dont le recul est impossible par la présence d'unités amies ou ennemies dans les carrés adjacents ou de terrains qui lui sont infranchissables est éliminée.

Table de combat

	RATIO Attaquant / Défenseur								
	1 contre 4	1 contre 3	1contre 2	2 contre 3	1 contre 1	3 contre 2	2 contre 1	3 contre 1	4 contre 1
- 3	5 R / 1	4 R / 1	4 / 2	3 R /2	3 / 2	2 R / 1	2 / -	1 R / -	1 / -
-2	4 R / 1	4 / 1	3 R /2	3 / 1	2 R / 1	2 / -	1 R / -	1 / -	-
-1	4 / 1	3 R /1	3 / 2	2 R / 1	2 / -	1 R / -	1 / -	-	- / 1
0	3 R /1	3 / 1	2 R / 1	2 / -	1 R / -	1 / -	-	- / 1	- / 2
1	3 / 1	2 R / 1	2 / -	1 R / -	1 / -	-	- / 1	- / 2	- / 2
2	2 R / 1	2 / -	1 R / -	1 / -	-	1 / 1	- / 2	- / 2	-/ 3
3	2 / -	1 R / -	1 / -	-	1 / 1	1 / 1	1 / 2	-/ 3	-/ 3
4	1 R / -	1 / -	-	1 / 1	1 / 1	1 / 2	1 / 2	1 / 3	-/ 3 R
5	1 / -	-	1 / 1	1 / 1	1 / 2	1 / 2	1 / 3	1 / 3 R	1 / 3 R
6	-	1 / 1	1 / 1	1 / 2	1 / 2	2 / 3	1 / 3 R	1 / 3 R	1 / 4 R
7	1 / 1	1 / 1	1 / 2	1 / 2	2 / 3	1 / 3 R	1 / 3 R	1 / 4 R	1 / 5 R

R = recul d'un carré.

8) Scénario historique

Axe

Afrika Korps

I/AK : P5	II/AK : P4	III/AK : Q4	IV/AK : Q3
pc=10.	pc=10.	pc=10.	pc=10.
al =8.	al=7.	al=7.	al=5.
aL=3	aL=2.	aL=2.	aL=4.
pm=8.	pm=8.	pm=8.	pm=8.

Force italienne

5 DI : K4	7 DI : K5	25 DCL : L5	2 DB : L4
pc=8.	pc=7.	pc=6.	pc=4
al=5.	al=6.	al=8	al=12.
aL=1.	aL=2.	aL=4.	aL=6.
pm=6.	pm=4.	pm=10	pm=6.

Britanniques

2 corps

26 Inf	27 Inf	14 Inf : H22	2 RC : J23	2nd AC
pc=8.	pc=8.	pc=8.	pc=8.	pc=3.
al=7.	al=7.	al=7.	al=6.	al=9.
aL=2.	aL=2.	aL=2.	aL=2.	aL=7.
pm=8.	pm=8.	pm=6.	pm=10.	pm=4.

Auxiliaires

3 BLF	Tobruk garnison : H23	12 Inf	Sidi G : H19
pc=4.	pc=5.	Pc=6.	Pc=4.
al=11.	al=5.	al=7.	al=5.
Al=5.	aL=1.	aL=2.	aL=1.
pm=6.	pm=4.	pm=8.	pm=4.

B) Schéma de l'application

1) Détail des classes

Etant donné le nombre important de méthodes dans l'ensemble des classes de l'application, nous avons choisi de détailler en priorité les méthodes qui nous semblaient importantes. De plus, les constructeurs sont tous présentés.

Après étude du cahier des charges, nous avons décidé de représenter les éléments suivants dans des classes :

Case :

Le plateau de jeu est divisé en un nombre de cases dépendant de la taille de la carte chargée. Une case connaît uniquement l'identifiant de l'unité qui l'occupe (éventuellement).

```
Case(int coord_x, int coord_z, int terrain=0);
```

Le constructeur de la classe Case prend en paramètres les coordonnées de la case, ainsi que le type de terrain identifié par un entier. On peut voir qu'une valeur par défaut est spécifiée pour le type du terrain, dans un souci d'optimisation. En effet la majorité des cases du plateau sont de type 'plaine' (code 0). Rappelons que le type du terrain a une importance considérable sur les déplacements et les combats dans Tobruk 1941.

```
int Inserer(Unite* unit);
int Supprimer(Unite* unit);
```

Ces deux méthodes permettent l'insertion et la suppression du lien entre une unité et la case concernée. Elles seront souvent appelées, en particulier lors des déplacements.

Ville :

Les villes sont les objectifs principaux dans Tobruk 1941. Une ville connaît la case sur laquelle elle se trouve, ainsi qu'éventuellement le joueur qui la contrôle.

```
Ville(char nom[32], Case* loc);
```

Le constructeur de la classe Ville, auquel on passe en paramètre la case où se situe la ville, ainsi que le nom de la ville.

```
Joueur* changerCamp(Joueur* camp);
```

Quand une ville est prise par un joueur, cette méthode est appelée. On lui passe en paramètre un pointeur vers le joueur qui vient de prendre la cité, et un pointeur vers le joueur occupant précédemment la cité est renvoyé. Ceci permet entre autres d'avertir ce dernier de la perte d'une cité…

Joueur :

A terme, notre application devrait permettre des parties ayant un nombre variable de joueurs. Un joueur est lié aux unités qu'il contrôle, ainsi qu'aux villes qu'il occupe.

```
Joueur(int numero, char nom[32]);
```

Le constructeur de la classe Joueur prend en paramètre un numéro qui servira d'identifiant à ce dernier, ainsi que la chaîne de caractère qui l'identifiera au niveau de l'IHM, car il est plus agréable (et réaliste) pour le joueur de visualiser un nom qu'un numéro.

```
inline int getNbVilles() {return this->villes.size();}
```

Cette méthode (définie « en ligne ») permet de connaître pour un joueur le nombre de cités contrôlées, ce qui est un élément déterminant pour la victoire.
A noter que l'attribut villes est un vecteur (voir chapitre suivant : « structures de données »), ce qui permet d'utiliser la méthode size() prédéfinie dans le langage.

```
inline void ajoutVille(Ville* city) {this->villes.push_back(city);}
```

Autre méthode « en ligne », la procédure ajoutVille permet d'ajouter la ville passée en paramètre à la liste des villes contrôlées par le joueur.

Unité :

Il existe deux grands types d'unités : les régiments d'infanterie ou de cavalerie (chars), et les généraux. Cependant ces deux types comportent certaines propriétés communes, comme leur capacité à se déplacer. Nous avons donc choisi de généraliser le type Unité, qui sera une classe abstraite, dont dériveront les classes Général et Régiment qui elles, seront instanciables.

```
Unite(int type, int pas, int deplacement, Joueur* camp, Case* loc);
```

Le constructeur de la classe Unite prend en paramètres le type d'unité, qui est soit l'infanterie soit les blindés, ainsi que le nombre de dégâts qu'elle peut subir (pas), ses points de mouvement (déplacement). Une unité possède forcément un Joueur qui la contrôle, ainsi qu'une Case où elle sera positionnée à l'origine, voilà pourquoi ces paramètres sont passés directement dans le constructeur.

```
virtual void Deplacer(Case* direction);
```

La fonction Deplacer est une des plus importantes du jeu. En passant une case en paramètre à cette méthode, on détermine si l'unité va simplement se déplacer (si la case destination est vide) ou alors attaquer (si elle est occupée par l'ennemi). C'est dans cette méthode que sera appelée la procédure de combat (voir chapitre traitant du sujet). A noter que la classe Unite étant abstraite, cette méthode est définie comme virtuelle pour permettre aux régiments et aux généraux de la redéfinir pour leur cas précis.

Régiment :

La classe Regiment dérive de la classe Unite. Un régiment est une unité qui possède un facteur d'attaque contre les chars et l'infanterie. Des régiments peuvent être ralliés sous une même bannière par un général.

```
Regiment::Regiment(int type, int pas, int deplacement, int attaque_inf,
int attaque_cav, Joueur* camp, Case* loc) : Unite(type, pas, deplacement,
camp, loc)
```

Le constructeur de la classe Regiment appelle celui de Unite, car Regiment dérive de cette classe. Cependant, certains attribut sont propres à Regiment, comme par exemple les facteurs d'attaque sur infanterie et cavalerie.

Général :

La classe General dérive de la classe unité. Bien que ne possédant pas la capacité d'attaquer, un général peut réunir sous son commandement plusieurs régiments, et leur accorder ainsi un bonus.

```
General::General(int type, int pas, int deplacement, Joueur* camp, Case*
loc, int bonus) : Unite(type, pas, deplacement, camp, loc)
```

Le constructeur de General ressemble beaucoup à celui de Regiment, car comme cette classe, General hérite de Unite. La différence vient du fait qu'un général ne possède pas de facteur d'attaque, mais un bonus qu'il accorde aux unités qu'il commande.

```
void AjouterRegiment(Regiment* reg);
void DetacherRegiment(Regiment* reg);
```

Les deux méthodes ci-dessus servent à ajouter ou enlever un régiment d'une armée commandée par le général. Des unités sous les ordres d'un général se déplacent ensembles, et profitent du bonus accordé par leur commandant. Pour plus de détail sur le rôle du Général dans Tobruk 1941, voir la partie règles du jeu dans ce rapport.

Terrain :

C'est à l'intérieur de la classe Terrain que les instances de toutes les classes du noyau sont créées. Ceci permet entre autre une interaction simple entre le noyau et l'affichage. Le constructeur de la classe Terrain n'est pas présenté ici car il ne fait rien d'autre qu'initialiser à 0 les attributs de l'objet.

```
Case*** ter;
std::vector <Unite*> unites;
std::vector <Joueur*> joueurs;
std::vector <Ville*> villes;
```

La classe Terrain peut accéder à l'ensemble des données (au niveau du noyau) définies lors de l'exécution de l'application. Les liens vers les unités, les joueurs et les villes sont des vecteurs (voir partie sur cette structure de données). Cependant pour gérer les cases du plateau, il nous fallait une structure de type matrice. Nous avons opté pour un tableau à deux dimensions. Ce tableau contient des pointeurs vers des instances de la classe Case.

2) Structures de données

Dans un souci de fiabilité et pour modifier ultérieurement le programme de manière aisée, nous avons utilisé certaines structures de données particulières dans notre application.

Le « Template Vector » permet de gérer efficacement tous les tableaux à une dimension, en particulier les tableaux de pointeurs pour réaliser les associations interclasses.
Enfin, le « Template Vector » fait partie de la « Standard Template Library », qui permet de standardiser l'écriture des programmes ainsi que sa compréhension par d'autres personnes.
Voici une utilisation concrète de ce template, par exemple dans la définition de la classe Joueur :

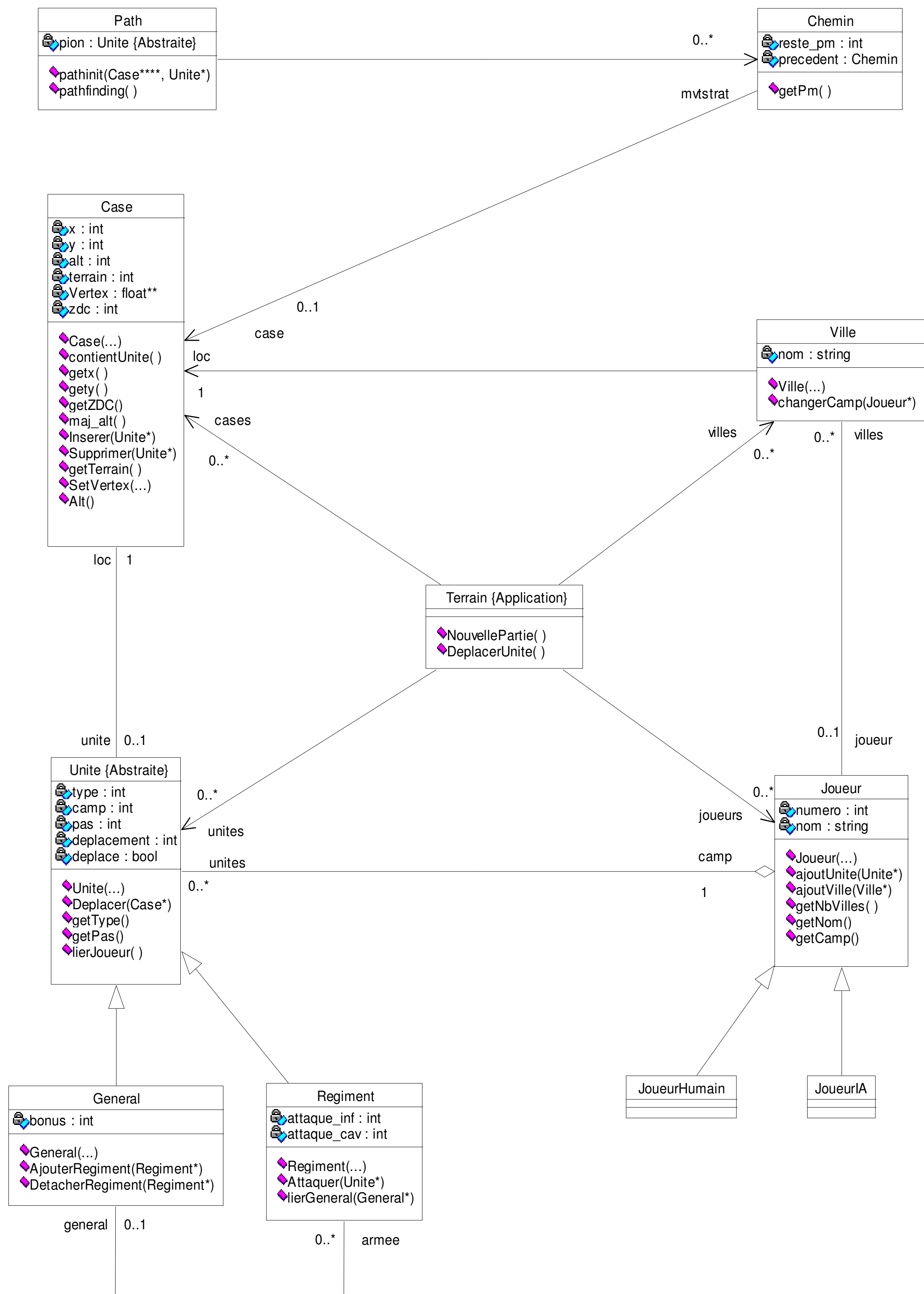
```
std::vector <Ville*> villes;
std::vector <Unite*> units;
```

Cette portion de code défini les vecteurs villes et units, respectivement vecteurs de pointeurs vers des villes et des unités.

3) Diagramme de classes généralisé

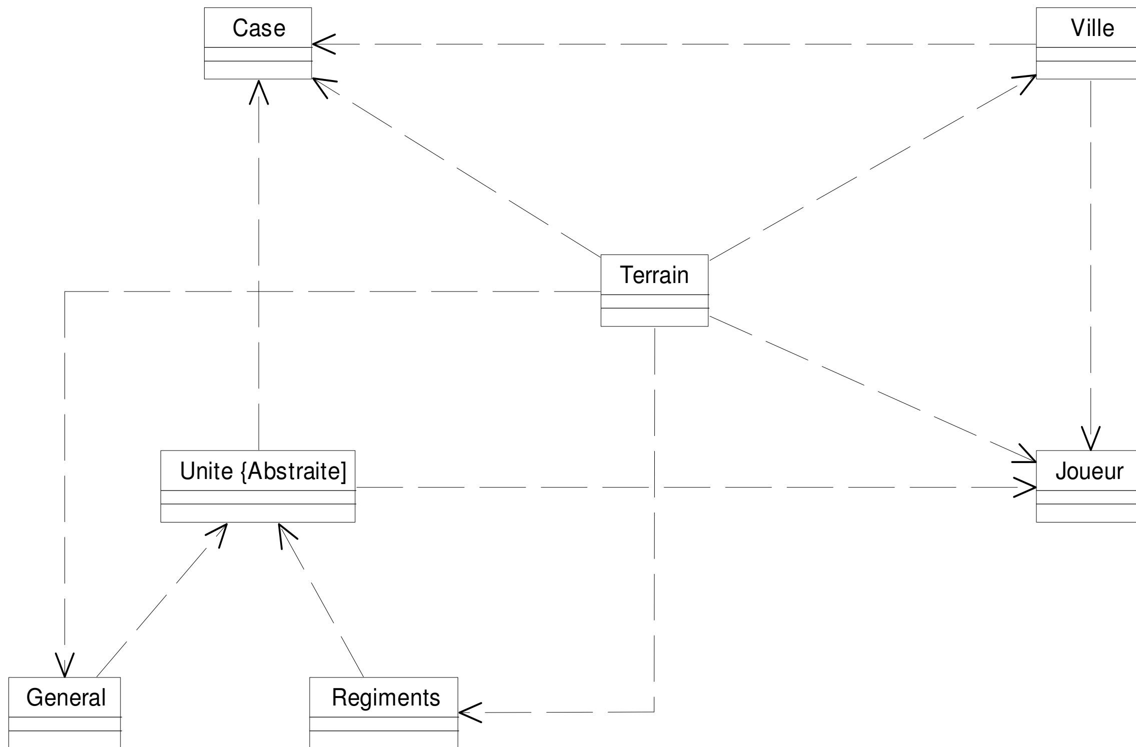
Dans le cadre de projets comme le notre, faire un diagramme de classe se révèle indispensable, afin ne pas omettre de liens entre les divers objets de l'application.

Ici, toutes les instances de classes sont crées par la classe Terrain, qui représente l'ensemble du système de jeu, unités et joueurs compris. Cette classe est donc considérée, au niveau du noyau, comme la classe application. Plus tard, la partie graphique encapsulera cette classe pour permettre une interaction aisée entre le noyau et l'IHM.


Path
pion : Unite {Abstraite}
pathinit(Case****, Unite*)
pathfinding( )
0..*
Chemin
reste_pm : int
precedent : Chemin
getPm( )
mvtstrat
Case
x : int
y : int
alt : int
terrain : int
Vertex : float**
zdc : int
Case(...)
contientUnite( )
getx( )
gety( )
getZDC()
maj_alt( )
Inserer(Unite*)
Supprimer(Unite*)
getTerrain( )
SetVertex(...)
Alt()
0..1
case
loc
1
cases
0..*
Ville
nom : string
Ville(...)
changerCamp(Joueur*)
villes
0..*
0..*
villes
loc
1
Terrain {Application}
NouvellePartie( )
DeplacerUnite( )
unite
0..1
0..1
joueur
Unite {Abstraite}
type : int
camp : int
pas : int
deplacement : int
deplace : bool
Unite(...)
Deplacer(Case*)
getType()
getPas()
lierJoueur( )
0..*
unites
unites
0..*
0..*
joueurs
camp
1
Joueur
numero : int
nom : string
Joueur(...)
ajoutUnite(Unite*)
ajoutVille(Ville*)
getNbVilles( )
getNom()
getCamp()
General
bonus : int
General(...)
AjouterRegiment(Regiment*)
DetacherRegiment(Regiment*)
Regiment
attaque_inf : int
attaque_cav : int
Regiment(...)
Attaquer(Unite*)
lierGeneral(General*)
JoueurHumain
JoueurIA
general
0..1
0..*
armee

4) Graphe des dépendances

Le graphe des dépendances montre quelles sont les classes qui doivent connaîtrent telles autres. Ceci est particulièrement important dans notre situation, car le diagramme de classes montre une structure en « boucle », ce qui pose d'importants problèmes d'inclusion dans les définitions des classes.



Légende :
Une flèche représente la relation « inclus », c'est-à-dire que la classe à l'origine de la flèche inclue celle qui se trouve à son extrémité.

5) Version alphanumérique

Etant donné que la partie graphique a été développée en parallèle avec le noyau, il a été nécessaire de construire une version de l'application exécutable simplement en mode console, afin de tester la validité du code.

Cette application servira à vérifier que les opérations d'instanciation, de mise à jour, et de liaison entre objets se déroulent sans problèmes, afin de s'assurer de posséder une partie opérationnelle solide avant d'implémenter l'affichage.

Enfin, et bien que cette version ne soit pas destinée à être présentée au terme du projet, elle permet surtout à plusieurs personnes de travailler en parallèle, et également de minimiser et de circonscrire les risques d'erreurs.

6) Conclusion et bilan

Nous avons tenu à commencer ce projet de manière méthodique, étant donné l'importance de la fiabilité à ce niveau de l'application.

Méthode de travail :

Le diagramme de classes ayant été défini en premier, la conception du noyau s'est avérée plus facile. Ce diagramme a cependant du être modifié de nombreuses fois au cours du projet, afin de satisfaire les besoins des différents membres du groupe, surtout en ce qui concerne le graphisme.

Nous avons surtout porté l'accent sur la production de code fiable, et si possible conforme aux recommandations vues en cours. Ainsi l'utilisation du « Template Vector », que nous avons appris à maîtriser, a permis de gagner un temps considérable lors de la programmation.

Il aurait été souhaitable d'étendre cette manière de fonctionner pour l'ensemble des structures de données, notamment les matrices, mais nous n'avions pas vu le modèle générique pour ces dernières en cours, et nous avons préféré programmer nous même ces parties, afin de ne pas perdre de temps.

Problèmes rencontrés :

Le principal problème rencontré au niveau de la conception du noyau a été l'édition des liens entre les classes. Après de nombreux tâtonnements, nous avons pu résoudre le problème grâce à un graphe des dépendances, qui comme son nom l'indique, défini les dépendances entre les classes. En effet, ce dernier a permis d'identifier clairement les classes qui posaient problème.

Enfin, la version alphanumérique de l'application a permis de s'assurer que l'ensemble des composants du noyau fonctionnait de manière conforme à nos attentes.

C) Intelligence Artificielle

1) Algorithme de meilleur chemin (« pathfinding »)

Méthode :

Pour avoir un algorithme de recherche de chemin puissant et pas trop gourmant en ressources, il a fallu l'optimiser. En effet, le parcours de tous les chemins possibles d'une unité est hors de question. Sachant qu'une unité peut avoir jusqu'a 12 points de mouvements, cela ferait un arbre de 8*7(^11) soit plus de 15 milliards de feuilles dans le pire des cas.

Le cas d'une matrice de même dimension que la carte et contenant le reste des points de mouvements de l'unité pour arriver dans cette case résout le problème des ressources. Mais elle ne permet pas de savoir pour une case qu'elle case est sa précédente.

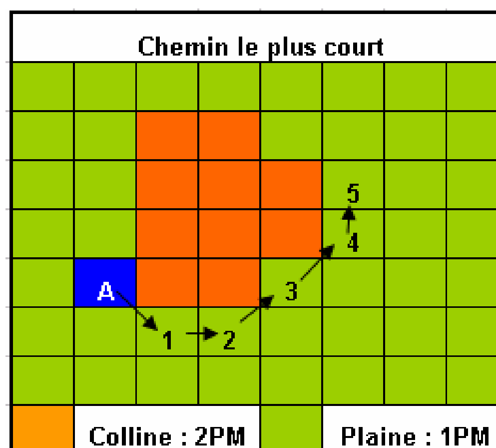
De plus, les zones de contrôles ne peuvent être respectées car on ne peut se déplacer d'une zone de contrôle ennemie à une autre alors que les deux cases seront remplies.

Un arbre imbriqué dans une matrice permet à la fois une économie de ressources (pas plus que pour la solution matricielle précédente) et une recherche du meilleur chemin possible. Partant du même principe que la solution matricielle, il suffit de rajouter la précédente case à chaque case remplie. On remplit d'abord les cases adjacentes à l'unité par rapport à la case de l'unité, puis on remplit les cases distantes de deux grâce aux cases préalablement remplies. Si une case est déjà remplie et qu'une procédure veuille la remplacer par une autre case n'ayant pas le même antécédent, elle devra vérifier si le reste de ses points de mouvement est supérieur à ceux de la case déjà présente.

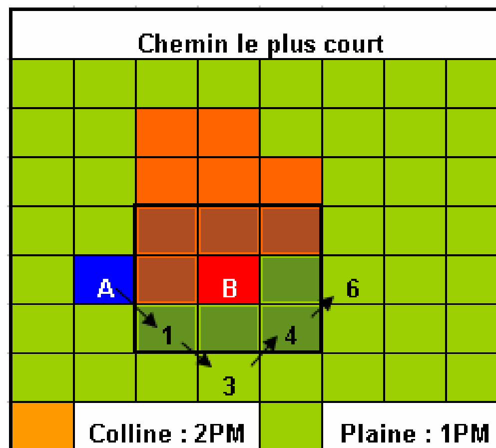
Ainsi, on peut gérer la totalité des règles de mouvement, y compris les effets des zones de contrôle.

Principe :

Il s'agit de trouver le meilleur chemin d'un point à un autre en prenant compte des règles de mouvement. Il est plus avantageux de se déplacer en plaine plutôt qu'en colline. Mais aussi les zones de contrôle limitent les déplacements directs. Voici quelques exemples d'application de ces règles :



Il est plus avantageux pour l'unité A de faire un détour par la plaine que de couper par les collines. En effet il ne lui en coûte que 5PM alors qu'il lui en coûterait 7PM en passant par les 3 collines.



L'existence d'une unité ennemie (B) sur ou à côté du chemin modifie celui-ci. L'unité A ne peut pas aller d'une case en zone de contrôle ennemie à une autre, et il lui en coûte +1PM pour en sortir. Le tracer du chemin doit être modifié.

Pour effectuer la recherche de chemin, nous devons donc créer une matrice de taille identique à la carte. Celle ci contiendra l'arbre de déplacement de l'unité et les meilleurs chemins pour toutes les cases.

La classe Chemin :

Avant de rechercher les meilleurs chemins, nous devons créer une matrice équivalente aux dimensions de la carte. Chaque case contiendra un pointeur vers un type de terrain (plaine, colline…), un pointeur vers le meilleur chemin pour arriver et le reste des points de mouvements de l'unité. Nous aurons ainsi un arbre restreint par la matrice.

Cette classe contiendra les méthodes de création, de destruction, de modification, et de consultation pour le restepm et le pointeur precedent.

La classe Path :

La class Path contient une matrice de case chemin, un pointeur vers la carte et un pointeur vers l'unité qui va se déplacer.

Elle contient des méthodes de création, de destruction, d'initialisation et une méthode pour calculer le bord le plus distant de l'unité.

La méthode pathfinding :

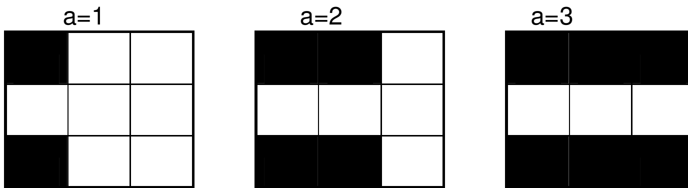
```
vector<Chemin*> pathfinding(Case *destination);
```

La méthode pathfinding renvoie un vecteur de Chemins pour permettre le déplacement graphique de l'unité.
Elle n'a besoin que de la case de destination pour envoyer un vecteur de déplacement.

Les entiers x et y contiennent les coordonnées de l'unité. L'entier portee permettra de savoir le nombre de parcours de périmètre nécessaires pour la couverture totale de la carte. Ceci permettra le remplissage de la matrice de chemin mvtstrat.
On crée un chemin début où l'on commencera à calculer, pour les cases du périmètre, le meilleur précèdent. Puis on remplit la case (x, y) de la matrice mvtstrat.

On remplit d'abord les bords haut et bas de la matrice, puis les colonnes droite et gauche.

Portee=1



Enfin, on retourne le vecteur du meilleur chemin pour la case destination.

La méthode Dir2case :

```
void Path::Dir2case(Case *carre, Chemin *precedent);
```

La méthode Dir2case remplit toutes les cases chemins adjacents à une case chemin en mettant leurs pointeurs précèdent sur elle si elles sont inexistantes où qu'elles ont un restepm inférieur à la sienne.

L'entier pen est la pénalité de mouvement si l'on sort d'une zone de contrôle. L'entier newcost est le reste des points de mouvement de l'unité pour la case chemin à rajouter. Le booléen noZdC sert pour déterminer la possibilité de mouvement.

Les entiers x et y sont les coordonnées de la case adjacente, xpre et ypre celle de départ.

La méthode dir2case remplit les cases chemins adjacentes à la case donnée en paramètre. Elle modifie les cases seulement si celles ci sont inexistantes où alors si elles ont un entier restepm inférieure à la modification. Elle ajoutera un pointeur à la case adjacente sur elle afin de calculer le meilleur chemin.

La méthode coordonnedeplacement :

```
vector<Chemin*> coordonnedeplacement(Chemin *arrive);
```

La méthode coordonnnedeplacement permet de mettre dans un vecteur le meilleur chemin pour la destination arrive.

On déclare un vecteur vec où l'on stockera les cases chemins de l'arrivée vers le début. On déclare un vecteur vec2 où l'on stockera les cases chemins du début vers l'arrivée en inversant le vecteur vec. On retourne le vecteur de chemin du départ vers l'arrivée.

2) Déplacement et combat

Déplacement :

Cette partie traitera du déplacement des unités selon les règles telles qu'elles ont été énoncées dans la première partie de ce rapport.

L'algorithme concerné est implémenté dans la classe unité (car chaque unité, qu'elle soit de type général ou régiment, se déplace de la même façon). Comme il est possible de le constater dans le diagramme de classes, il s'agit donc de la procédure « `deplacer(*case)` ».

La procédure consiste à, dans un premier temps, « autoriser » le déplacement de l'unité concernée de la case où elle est située à l'autre case, passée en paramètre. Si le dit déplacement est possible, alors le potentiel de mouvement est automatiquement décrémenté en fonction des points de mouvement nécessaires au parcours du chemin entre les deux cases citées précédemment. L' « autorisation » de déplacement ainsi que les points de mouvements sont autant d'attributs de la classe unité, au même titre que le camp, les points de combat, et le type.

Le chemin, quant à lui, constitue une classe à part, reliée à la classe unité (voir Diagramme de Classes). La classe Chemin comporte entre autres un ensemble de cases, pour l'instant regroupées dans un tableau d'objets.
Une fois l'autorisation de déplacement accordée, la procédure effectue une boucle qui parcourt le tableau de cases de la classe Chemin. A chaque case parcourue, les coordonnées de cette case sont attribuées aux attributs de la case où est localisée l'unité. Cette boucle permet de faire figurer l'unité dans chaque case intermédiaire constituant le chemin ; elle fluidifie, pour ainsi dire le déplacement.

Combat :

L'algorithme de combat n'est pas encore complet ; toutefois, sa démarche a déjà été définie et il est en cours d'implémentation.

En voici le fonctionnement :
Précisons tout d'abord que l'algorithme de combat constitue le corps de la procédure « `attaquer(unite*ennemi)` » contenue dans la classe Regiment.

D'abord est calculé le ratio attaquant/défenseur en fonction de la nature des unités attaquante et défenseur :
Soient:

attaque_inf (att ou def) : le facteur d'arme légère d'une unité
attaque_cav (att ou def) : le facteur d'arme lourde d'une unité
PC (att ou def): les points de combats d'une unité.

Si l'unité attaquante est une unité d'infanterie:
- Si l'unité attaquée (défenseur) est une unité d'infanterie:
  Ratio = (attaque_inf(att)*PC(att))/(attaque_inf(def)*PC(def))

- Si l'unité attaquée (défenseur) est une unité de cavalerie ou une colonne de blindés:
  Ratio = (attaque_cav(att)*PC(att))/(attaque_inf(def)*PC(def))

Si l'unité attaquante est une unité de cavalerie ou de blindés:
- Si l'unité attaquée (défenseur) est une unité d'infanterie:
  Ratio = (attaque_inf(att)*PC(att))/(attaque_cav(def)*PC(def))

- Si l'unité attaquée (défenseur) est une unité de cavalerie ou une colonne de blindés:
  Ratio = (attaque_cav(att)*PC(att))/(attaque_cav(def)*PC(def))

Après le calcul du ratio est généré aléatoirement un entier compris entre 1 et 6. A ce nombre sont additionnés les différents modificateurs de combat spécifiés dans le rapport intermédiaire (type de terrain où se trouve l'unité attaquée, etc...). Selon le nombre ainsi obtenu et la valeur du ratio sont définis les dégâts subis par chaque unité impliquée dans le combat, conformément à la table de combat (voir chapitre sur les règles).

Signification du contenu des cases : le premier chiffre correspond au points de combat perdus par l'unité attaquante ; le deuxième chiffre (celui après le slash) correspond à ceux perdus par le défenseur.
Le "R" figurant après l'un de ces deux chiffres correspond à la nécessité pour l'unité concernée de reculer.

Gestion du recul : une fois le choc du combat « calculé », la procédure fait reculer l'unité concernée vers la première case n'étant pas une zone de contrôle ennemie. Pour cela, il a fallu modifier la classe case de manière à ce qu'elle compte parmi ses attributs un tableau d'objet dont chaque entrée pointe vers une case adjacente à la case en question. Ainsi, la procédure « `attaquer(unité*ennemi)` » effectue une boucle qui parcourt le tableau des cases adjacentes à celle ou se déroule le combat. Dès qu'une case n'étant pas une zone de contrôle ennemi, l'unité devant reculer s'y déplace.

Remarque: la case ou l'unité doit reculer voit ses coordonnées directement attribuées à celles de la case où doit désormais se trouver l'unité. En effet, la case étant adjacente, nul besoin de faire appel à la procédure « `déplacer(case*direction)` ».

<u>II) Gestion de l'affichage</u>

A) Création et initialisation de la fenêtre

La fenêtre de jeu à été programmée en OpenGL, cependant, pour une meilleure convivialité auprès des utilisateurs, cette fenêtre doit être incluse dans une fenêtre classique de Windows (puisque c'est sous ce système d'exploitation que nous choisi de développer l'application). Pour réaliser une telle fenêtre, il faut utiliser les fonctions que l'on trouve dans les librairies Win32 du soft de Microsoft.

Les fonctions contenues dans ces librairies sont très complexes et nous allons tenter d'expliquer le fonctionnement d'initialisation d'une fenêtre sans trop entrer dans les détails car cela reviendrait à explorer des centaines de fichiers de code. Nous allons nous attacher à ne décrire que les points les plus importants du code.

Les fichiers les plus importants se rapportant à la gestion de la fenêtre programmée en Win32 sont Main.h et Main.cpp. Nous allons d'abord commenter le fichier Main.h, puis nous verrons le contenu du fichier Main.cpp, qui utilise les procédures, fonctions et variables qui sont codées dans le premier fichier. En plus de ces deux fichiers, la fenêtre Win32 utilise principalement deux autres fichiers : Resource.h et Ressources.rc. Le premier ne sera pas décrit ici car il ne contient que des affectations de valeurs entières à des variables utilisées dans Main.cpp. Quant au deuxième, il sera évoqué brièvement.

Le lecteur peut bien entendu se référer à tout instant au code source de l'application donné en annexe afin de mieux comprendre les commentaires écrits ici et de voir l'intégralité des procédures décrites.

1) Fonctions

Main.h :

Les déclarations de variables concernent principalement la caméra, la souris, les unités et le terrain ainsi que des réels et des matrices utilisées par l'affichage OpenGL.

La première fonction : `int getfps() {}` permet de calculer le nombre d'images par seconde (dépendant de la puissance de la machine sur laquelle l'application tourne) afin de l'afficher ultérieurement.

La deuxième procédure : `void InitPixelFormat(HDC hDC) {}` initialise le format de pixel à l'intention de l'affichage OpenGL. A noter qu'à l'intérieur, on déclare l'utilisation de deux tampons dans la carte graphique : un qui reçoit ce qui est dessiné en OpenGL et l'autre qui va permettre l'affichage à l'écran.

La dernière procédure de fichier : `void RePaint() {}` a pour tâches principales de positionner la caméra et d'afficher le terrain. De plus, il y a dans cette procédure un appel à une fonction qui va inverser les deux buffers de la carte graphique évoqués précédemment. Cela va permettre d'afficher à l'écran tout ce qui aura été dessiné en OpenGL. Si cette procédure n'était pas présente, l'utilisation d'un unique buffer dessinerait à l'écran les éléments les uns après les autres, à mesure qu'ils seraient créés par le moteur graphique. Son utilisation va permettre un affichage rapide tous les éléments simultanément et rendre ainsi le jeu plus agréable pour l'utilisateur.

Main.cpp :

La procédure la plus importante de l'affichage se trouve dans ce fichier : `int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {}`.

En premier lieu, elle appelle la fonction `BOOL CreateGLWindow(char* title, int width, int height, int bits) {}`. Elle reçoit en paramètres le titre de la fenêtre, qui sera affiché dans la barre de titre et la résolution : largeur, hauteur et profondeur de bits. Elle va d'abord affecter aux coordonnées du point situé en haut à gauche de la fenêtre les valeurs (0,0). Cela va positionner ce point tout en haut à gauche de l'écran. Quant aux coordonnées du point situé en bas à droite de la fenêtre, elles seront affectées par les variables passées en paramètres à l'appel de la fonction.
Elle active ensuite les synchronisations horizontale et verticale puis appelle la fonction WindowProc, que nous décrirons plus loin.

Après cela, elle crée l'icône placée à gauche dans la barre de titre de la fenêtre ainsi que le type de curseur à afficher pour la souris et la couleur de fond de la fenêtre.
On peut voir ensuite l'affectation du nom du menu. On donne le nom sous lequel sont déclarés les menus dans Ressources.rc pour que la fonction puisse les retrouver en parcourant de fichier.

La condition sert à déterminer si la fenêtre sera créée en plein écran ou non.

Il y a ensuite deux appels à des fonctions qui vont chercher les menus et les raccourcis qui sont déclarés dans le fichier Ressources.rc.

Enfin, les deux dernières lignes servent à afficher la fenêtre au premier plan et à la rendre active.

L'autre fonction très importante de ce fichier est `LRESULT CALLBACK WindowProc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {}`. Elle va effectuer différentes actions suivant le message qu'elle va recevoir à son appel.

Dans le premier cas, elle va initialiser le format de pixel grâce à la fonction de Main.h ; sélectionner une police par défaut pour cette fenêtre, notamment utilisée pour écrire les FPS ; initialiser le random utilisé dans la gestion de la souris ; initialiser sept textures et convertir les fichiers bitmap en textures ; appeler le constructeur de la classe Terrain ; initialiser la caméra ; mettre à zéro un vecteur temporaire pointant sur des unités et enfin, initialiser et placer la souris.

Elle peut aussi être sollicitée pour terminer l'application, elle va pour cela appeler la procédure `void Shutdown(HWND hwnd) {}` qui va fermer la fenêtre et quitter le programme.

Le programme peut encore lui demander de créer des coordonnées exploitables par l'OpenGL pour l'écran et la carte graphique. A noter qu'elle est appelée à chaque redimensionnement de la fenêtre.

Son rôle peut consister à exécuter diverses actions suivant quelle touche du clavier ou quel bouton de la souris a été activé par l'utilisateur, ou encore suivant quel menu de la fenêtre celui-ci a sélectionné.

Enfin, elle peut redessiner ce qui apparaît dans l'écran de jeu.

Nous en revenons à la procédure principale qui a effectué ces différents appels pour créer et initialiser la fenêtre. On y trouve la boucle du jeu : une boucle infinie qui va recevoir les messages qu'on lui transmet, c'est à dire les actions de l'utilisateur.

Elle va d'abord vérifier si un message lui a été envoyé. Si c'est le cas, elle va chercher à savoir ce que contient ce message puis à exécuter l'action qui lui correspond. Si elle n'a pas reçu de message, elle va redessiner ce qu'il y a à afficher en OpenGL.

Enfin, quand l'application est fermée, elle quitte la boucle et libère la mémoire qui avait été allouée au jeu.

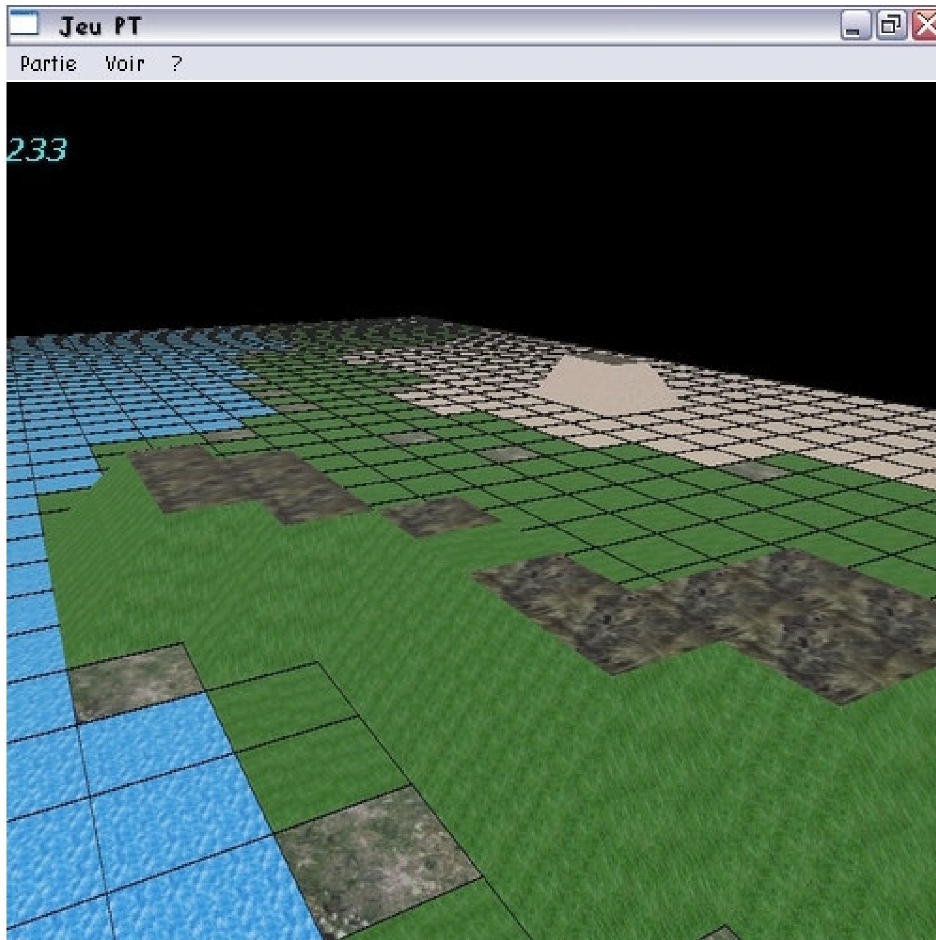
2) Impression d'écran

Cette image est un screenshot de l'application que nous avons développé. Elle montre l'affichage de la carte de jeu dans la fenêtre Win32.

On peut y remarquer que l'icône chargée par la fonction « LoadIcon(…) » appelée dans « CreateGLWindow » correspond à une icône de console MS-DOS car elle est l'icône par défaut et nous n'avons pas choisi de la modifier. On note aussi que la fenêtre se nomme « Jeu PT », cela est aussi du au paramètre passé à l'appel de la méthode « CreateGLWindow ».

En dessous, se trouvent les menus :

- Partie : Sauvegarde ou chargement de partie en cours ; création d'une nouvelle partie ; fermeture du programme.
- Voir : Affiche des renseignements sur certaines unités
- ? : Contient un « à-propos » et un fichier d'aide.



3) Problèmes rencontrés

Problèmes rencontrés :

Le plus complexe des obstacles que nous avons rencontré sur cette partie tient à l'idée que nous avions du jeu au début de sa réalisation. A savoir que nous voulions afficher un menu interactif à gauche de l'écran de jeu. Dans cette partie, nous voulions faire apparaître les informations relatives à la zone de jeu sélectionnée à ce moment par l'utilisateur. Cependant, il nous est rapidement apparu que la séparation de la fenêtre de jeu en deux fenêtres et principalement leur interaction était extrêmement difficile à réaliser. Nous avons tenté de résoudre ce problème pendant la majorité de la durée du développement de l'application mais n'y sommes pas arrivé.

L'autre problème vient des menus (ceux du haut de l'écran). Tout d'abord au niveau de la sauvegarde et du chargement. Nous n'avons été assez loin dans le développement pour pouvoir nous occuper de ces fonctions. En effet, leur développement aurait nécessité des modifications au niveau du noyau et cela n'a pas été possible car il y avait des méthodes plus importantes à développer. Enfin, nous voulions créer un fichier d'aide pour l'utilisateur, expliquant le fonctionnement du jeu et plus généralement de l'application mais, de même, ce rajout aurait été effectué après avoir développer tout le reste de l'application, un tel document étant d'une moindre importance.

B) Mise en œuvre de l'OpenGL

1) Création et affichage de la carte

Lors de l'initialisation de la fenêtre OpenGL, une procédure appelée build et appartenant à la classe Terrain permet de construire la carte à partir du fichier tobruk.txt (voir annexe). Cette méthode permet de positionner la caméra, de créer le terrain en lisant dans le flux f déclaré en lecture, d'appeler la méthode de mise à niveau des différentes cases et enfin de créer une liste d'affichage qui sera utilisée dans la sélection grâce à un clic sur le bouton de gauche de la souris.

Format du fichier tobruk.txt :

Pour bien comprendre le mécanisme d'extraction du terrain à afficher, il faut tout d'abord s'intéresser à la manière dans la carte est représentée dans le fichier.

Première ligne du fichier : `mapsize:40,20`

Ce passage du programme va nous permettre de connaître la hauteur et la largeur de la carte à charger. Ici, la hauteur est de 40 (axe des z) et la largeur de 20 (axe des x).

Dans le reste du fichier, toutes les informations concerneront les cases, il existe 7 types de cases différentes :

- s = mer
- d = désert
- p = plaine
- m = montagne
- c = colline
- m = marais
- v = ville

Nous allons maintenant voir le mécanisme de lecture et d'extraction du terrain grâce au flux f et les symboles qu'il contient.

Lecture du fichier tobruk.txt :

La variable surface que nous avons préalablement déclarée nous servira ici à récupérer les informations et les mettre sous forme de chiffres qui seront plus simples à analyser par la suite. Cette variable est en fait un pointeur sur un tableau de pointeurs sur des objets de type Case, il faut donc allouer un espace mémoire pour chaque pointeur :

Première instanciation, effectuée avant le parcours du fichier pour allouer dynamiquement assez d'espace mémoire pour créer plus tard la suite du tableau :

```
surface=(Case***)malloc(TAILLEMAPV*sizeof(int**));
```

La deuxième instanciation est effectuée pour chaque ligne (axe des x) et permet de nous allouer de la mémoire pour créer des pointeurs sur les cases de la carte :

```
surface[i]=(Case**)malloc(TAILLEMAPH*sizeof(int*));
```

Ensuite, on lit chaque caractère un par un et l'on teste le type de terrain auquel le caractère correspond pour créer la Case correspondante à ce terrain. Chacune des TAILLEMAPH*TAILLEMAPV cases sera pointée par un élément du tableau à deux dimensions :

```
surface[i][j]=new Case(i,j,1);
```

Le codage utilisé pour créer les instances de Case est le suivant :

- 's' = 1
- 'd'=2
- 'p'=3
- 'o'=4
- 'c'=5
- 'm'=6
- 'v'=7

Evaluation du terrain pour la mise à jour des altitudes :

Le principe de cette méthode est simple, il suffit d'incliner les cases qui sont à coté d'une case 'montagne' (lettre 'o' dans le fichier tobruk.txt ou 4 dans les instances de Case).

Pour cela, il faut parcourir l'ensemble des cases et tester celles qui sont de type montagne et ainsi rehausser les huit cases touchant celle-ci. Ce parcours s'effectue dans la méthode eval_terrain.
Les deux boucles imbriquées permettent de parcourir l'intégralité des Cases en partant des coordonnées (0,0) et en allant jusqu'an (TAILLEMAPV,TAILLEMAPH). Pour chaque case, on teste s'il s'agit d'une montagne, si oui, on effectue toutes les conditions de non dépassement du tableau surface (dans les cas où une montagne serait sur le bord de la carte) et on rehausse les huit cases adjacentes.

Une fois les boucles horizontale et verticale terminées, on renvoi la surface à la méthode build.

Création de la liste de sélection :

Cette liste va servir à récupérer les coordonnées du clic de sélection d'une case. Le principe est simple, il faut empiler des noms que l'on donne à des objets (ici des cases) pour ensuite pouvoir les récupérer lors du clic grâce à la fonction adéquat. Tout d'abord, on emploie les fonctions classiques de génération de liste :

- `list_=glGenLists(1);`

Cette fonction nous permet de créer un certain nombre de listes d'affichage (ici une) et de récupérer le numéro de la première qui sera stockée dans la variable list_.

- `glListBase(list_);`

On défini ensuite le numéro de la première liste dont on va se servir, dans cet exemple, ce sera donc le numéro de la première liste qui a été créée précédemment.

- `glNewList(list_,GL_COMPILE);`

On indique ici que l'on souhaite commencer à créer une liste d'affichage de numéro list_ et en mode GL_COMPILE (ce mode est utilisé si l'on souhaite uniquement 'enregistrer' la liste sans l'exécuter, cela aurait été fait grâce à la constante GL_COMPILE_AND_EXECUTE mais celle-ci est inutile ici).

- `glEndList();`

La fin de la liste est indiquée plus bas cette fonction et sert à clôturer la liste d'affichage numéro list_.

Nous venons de voir le fonctionnement d'une liste d'affichage, il ne reste plus qu'à voir l'implémentation des noms pour la sélection.
Le premier constat que nous pouvons faire sera qu'il faut une boucle pour parcourir toutes les cases dans l'ordre souhaité. Vient ensuite le placement des fonctions nous permettant de mettre en œuvre les noms :

- `glLoadName(i);`

Cette fonction sert à remplacer la valeur qui se trouve en haut de la pile des noms, en l'occurrence, cette fonction remplacera la valeur des z afin de créer cet axe.

- `glPushName(j);`

On insert ici en nom dans la pile (le nom j), cela nous servira à récupérer la valeur des x en ordonnées sur notre plan.

L'étape suivante consiste à créer l'objet que l'on souhaite associer à ce nom, ici, on créer la surface représentant la case (un carré).

- `glPopName();`

Comme pour les matrices, après chaque invocation d'un empilement, il faut dépiler le nom une fois que l'objet désigné par les noms i et j a été dessiné.

Affichage de la carte :

Le principal élément visible par l'utilisateur dans ce jeu est certainement la carte, nous allons donc voir comment celle-ci est appelée puis affichée.

Dans le fichier main.h, un objet t de type Terrain est crée et l'appel à la création de la carte s'effectue au moment de la création de la fenêtre Win32 grâce à la méthode build que nous avons étudiée précédemment.
La boucle d'affichage appel à chaque passage la fonction RePaint dans laquelle on trouve la méthode afficher de la classe Terrain. C'est donc à cette invocation que le terrain va être dessiné dans le backbuffer.
L'affichage du terrain s'effectue case par case dans l'ordre inverse à la lecture du fichier de carte tobruk.txt pour que l'image soit parfaitement cohérente avec les coordonnées d'affichage et le fichier. Pour chaque case on testera donc le type de terrain dont il s'agit (montagne, mer, ville,…) puis on effectuera les diverses opérations sur chacune d'elle. Nous allons donc détailler ce qu'il se passe pour une case :

- ```
  GLfloat mat_[4]={0.6f,0.6f,1.0f,1.0f};
  GLfloat mat_0[4]={0.0f,0.0f,0.0f,1.0f};
  ```

On défini ici les deux vecteurs qui vont nous servir à modifier les paramètres de couleur de la case (ils contiennent ici les informations RGBA d'une couleur).

- ```
  glBindTexture(GL_TEXTURE_2D, texture[0]);
  ```

Cette fonction permet de charger une texture en mémoire enfin de pouvoir s'en servir et donc l'appliquer à une surface.

- ```
  glMaterialfv(GL_FRONT,GL_SPECULAR,mat_);
  glMaterialf(GL_FRONT,GL_SHININESS,10.0);
  ```

Ces deux fonctions permettent de modifier les paramètres de reflets des différentes cases. La première affecte la matrice mat_ au paramètre de réflexion spéculaire de l'objet. La valeur passée en paramètre par la deuxième fonction permet de définir la concentration ou brillance du reflet.

Les lignes suivantes permettent de définir l'objet carré servant de support à l'application de la texture. Puis en change la valeur de la réflexion spéculaire pour lui passer en paramètre une matrice 'nulle' qui permettra de remettre sa valeur à zéro.

2) Mouvements de la caméra et des unités

La classe Camera :

Cette classe permet de stocker toutes les informations concernant les angles de vues, les coordonnées du point de vue ainsi que le point vers lequel elle est centrée. Une instance de cette classe est déclarée dans le fichier main.h puis le constructeur de cette classe est appelé par l'événement de création de la fenêtre Win32 :

- ```
  c=new Camera(t.getCamPos().x, t.getCamPos().y,
  t.getCamPos().z, t.getCamDir());
  ```

Dans la boucle d'affichage (fonction RePaint()), on fait appel à la méthode set() de la classe Camera pour placer la caméra sur le terrain :

- ```
  gluLookAt(x_,fy_,z_,xc_,yc_,zc_,xh_,yh_,zh_);
  ```

Evènements et méthodes de déplacement :

Pour les déplacements parallèles au terrain (sur l'axe x ou z), on fait appel à la méthode moveXZ() qui permet de se déplacer latéralement dans la limite du terrain pour ne pas déborder.

Pour regarder autour de soi-même grâce à la souris, on utilise plusieurs méthodes et évènements. La première consiste à détecter si le bouton de droite de la souris a été appuyé, cette action place le booléen click_droit sur vrai ce qui permettra de s'en servir lors d'un mouvement de la souris. Ce mouvement est détecté par la fonction Souris_Bouge(). Lorsque le bouton de droite est enfoncé, le booléen est vrai et on entre donc dans la condition pour prendre les coordonnées de la souris grâce est la fonction :
  ```

GetCursorPos(&pt). On analyse ensuite s'il y a eu un mouvement et l'on transmet ce mouvement aux méthodes rotateH() et rotateV(). Ces deux fonctions modifient les angles de vue de la caméra active soit verticalement soit horizontalement.

Pour se déplacer verticalement maintenant, on utilise comme pour les changements d'angles de vue un booléen (click_milieu) pour détecter si le bouton du milieu de la souris a été cliqué puis on effectue un appel à la méthode translate() de la classe Camera. Cette fonction très simple ajoute juste les déplacements aux coordonnées actuelles et vérifie si la hauteur (axe des y) reste comprise dans une certaine fourchette.

Lors du relâchement d'un bouton de la souris, le booléen repasse à faux ce qui désactive la possibilité de déplacer la souris pour changer le point de vue.

Placement des unités :

L'affichage des unités s'effectue juste après celui du terrain, il est donc appelé autant de fois. Ce qui crée l'impression de déplacement est donc un affichage consécutif de plusieurs images des unités à des coordonnées différentes. Plus il y a d'images par seconde, plus l'animation paraît fluide.

Grâce à l'attribut ter de la classe Terrain, on peut connaître toutes les unités se situant sur la carte ainsi, bien sur, que ses coordonnées. Il suffit donc de définir la hauteur de l'objet à afficher et effectuer une translation pour le placer au bon endroit sur la carte.

Conclusion

Le jeu est l'une des raisons principales du développement des nouvelles technologies informatiques, il prend donc une place considérable tant au niveau de la durée d'utilisation qu'au niveau économique. Notre projet étant axé sur cette voie, il nous a été facile de percevoir les difficultés qui étaient rencontrées par les développeurs et les designers pour arriver à concilier un délai de réalisation assez court et de bonnes performances au niveau de la qualité du jeu.

Notre groupe était composé de cinq personnes et nous avions quatre mois pour finaliser ce projet alors qu'un jeu actuel occupe plus d'une centaine de personnes pendant environ un an. Nous sommes tout de même parvenus à programmer une application fonctionnelle regroupant toutes les exigences du cahier des charges et ayant un aspect et une ergonomie moderne et conviviale. Le principal problème qui fut rencontré lors de ce développement fut le regroupement des deux parties principales (interface graphique et noyau de l'application) qui ont été développées séparément.

Ce projet, en plus de nous avoir permis d'acquérir des informations précieuses et fortement utiles pour la suite de nos études, nous a apporté plusieurs sortes d'expériences :

- La capacité à travailler, s'organiser et se coordonner dans un groupe d'étudiants, de fournir un maximum de travail, tout cela dans le respect du planning et des délais et une plus grande facilité à rechercher l'information pour certains d'entre nous.

- La vision d'un projet d'une envergure supérieure à ceux effectués en TP.

- Un approfondissement des langages et des techniques informatiques vues en cours.

Nous tenions à remercier Messieurs Belkatir et Gouliand (tous deux enseignants à l'IUT 2 de Grenoble, Département Informatique) pour les connaissances indispensables qu'ils nous ont apporté en programmation C++ et qui ont été utilisées lors de ce projet, ainsi que Mr Cosnier (enseignant à l'IUT 2 de Grenoble, Département Informatique) pour les conseils et les rectifications qu'il nous a fourni.

Références documentaires

Livres :

OpenGL 1.2
Le langage C++

« Le Langage C (2ème édition) » écrit par Brian W. Kernighan et Denis M. Ritchie et édité en 1997chez Masson

« Indispensable pour... C++ » écrit par Bernard Frala et édité en 1999 chez Marabout Informatique

Sites Web :

www.gametutorial.com
www.opengl.org
www.ultimategameprogramming.com
www.functionx.com/win32
www.3d-cafe.com

http://www.gametutorials.com/Tutorials/Win32/Win32_Pg1.htm
http://www.alrj.org/docs/3D/opengl/OpenGLII/OpenGLII.htm
http://win32.planet-d.net/tut_w/chap1.htm
http://www.functionx.com/win32/
http://texel3d.free.fr/win32/index.html

Outils utilisés

- Microsoft Visual C++ 6 (version d'évaluation distribuée PC Team Hors Série n°8)
- Dev-C++ 4.01
- Microsoft Office Word et Excel (versions de l'IUT)
- Rational Rose C++ 4.0 (version d'évaluation fournie par l'IUT)

<u>Répartition des tâches</u>

Cédric :	- Partie graphique (jeu : terrain, unités…) et interaction avec l'utilisateur (entrées claviers, souris…) - Rédaction du cahier des charges
Jean :	- Conception / programmation du noyau (classes et interaction entre les classes)
Mickaël :	- Ecriture des règles - Ecriture de l'algorithme de « pathfinding »
Aurélien :	- Création et initialisation d'une fenêtre Win32 - Mise en forme du rapport et rédaction du cahier des charges
Hicham :	- Ecriture de l'algorithme de déplacement et de combat

Résumés

Résumé :

Ce projet est un jeu de stratégie au tour par tour, développé en C/C++, Win32 et OpenGL. Le jeu est jouable à la souris et son interface est intégrée dans une fenêtre où apparaissent aussi, en haut, deux menus. Un menu général et un autre, d'aide à l'utilisateur.

L'action se déroule en Afrique du Nord, au début des années 1940, pendant la seconde Guerre Mondiale.

Les joueurs auront la possibilité de choisir entre les troupes de l'Axe (Afrika Korps et Italiens) ou les forces Alliées (armée Britannique). Chaque camp possède différents types d'unités, comme l'infanterie et les blindés. Chaque type d'unité possède des potentiels d'attaque, de défense et de mouvement qui lui sont propres. Ces caractéristiques pouvant être modifiées par le type de terrain où se trouve l'unité. Les groupes d'unités ne peuvent se séparer en deux autres groupes, ni fusionner avec un autre groupe.

L'aspect réaliste de ce jeu vient du fait qu'une unité ayant été endommagée au cours d'un combat perd une partie de son potentiel d'attaque. De plus, les informations concernant les forces en présence dans l'Histoire sont tirées de la réalité ; un effort a été fait sur ce point.

Mots clés :

Jeu de stratégie au tour par tour, Jouable à la souris, C/C++, Win32, OpenGL, Potentiel d'attaque, Potentiel de défense, Potentiel de mouvement, Terrain, Blindés, Infanterie.

Abstract :

This project presents a turn-taking wargame which has been developed in C/C++, Win32 and OpenGL languages.

The interface will be clicked-playable and integrated in a window where two menus appear on the top. One is a general menu (new game, save, load, quit) and the other one is a helping menu. The action takes place in North Africa, in the early 40's, during the World War II.

The players will be able to choose between Allied (British army) and Axis troops (Afrika Korps and Italians). Each troop has several types of units (infantry and armoured columns). Each type of unit has a certain potential of attack, defence and moving. All this characteristics may be influenced by the type of field were the unit is located. The units don't have the ability to divide themselves into two weaken ones or to regroup into stronger ones.

The realistic aspect of this game is that a unit which has been damaged during an attack loses some of its attack potential. Moreover, the data concerning the present forces in History is taken from reality; a real effort has been made in that area.

Keywords :

Wargame, Clicked-Playable, C/C++, Win32, OpenGL, Potential of attack, Potential of defence, Potential of moving, Field, Armoured Columns, Infantry.

Annexes

Annexe 1 : Cahier des charges

Table des matières

Introduction 30

Objectifs 30

Besoins non fonctionnels 31

- Interface utilisateur 31
- Configuration requise par l'application 31
- Environnement réseau 31
- Développement de l'application 31

Besoins fonctionnels 31

Fonctions principales de l'application 32

- Module « ? » 32
- Module « Fichier » 32

Evolution de l'application 32

Annexes 33

- Glossaire 33
- Composition de l'équipe 33
- Planning de réalisation 33

Introduction

Dans le cadre de notre Projet Tuteuré, notre groupe, dirigé par Opfermann Cédric a pour but de réaliser un jeu vidéo sur ordinateur. Ce jeu doit être un jeu de stratégie au tour par tour en 3D pouvant permettre à deux utilisateur de « s'affronter » au cours d'une partie sur un même appareil.

L'action se déroulera en Afrique du Nord en 1941 et opposera les forces Alliées (armée Britannique et population indigène) aux forces de l'Axe (troupes du Général Rommel et armée Italienne).

Le but du jeu sera de contrôler les villes présentes sur la carte. Pour arriver à cela, le joueur doit pouvoir déplacer ses unités sur le terrain et les faire combattre.

Objectifs

L'objectif principal de l'application est sa stabilité et sa fiabilité.

La prise en main du jeu devra être simple et accessible à tous types d'utilisateurs, les réglages devront se limiter aux réglages des touches utilisées pendant la partie.

Pour une utilisation plus aisée du logiciel, son développement devra permettre une jouabilité correcte sur un ordinateur de faible puissance.

Besoins non fonctionnels

Interface utilisateur

Du fait de la facilité de prise en main du logiciel, l'ergonomie devra être simplifié. Le menu devra se résumer actions principales de sauvegarde, chargement et fermeture de l'application et devra aussi contenir un fichier d'aide à l'utilisation expliquant le fonctionnement du jeu.

Configuration requise par l'application

- La configuration minimale adoptée est celle du Pentium 3 333 MHz avec 128 Mo de RAM.
- La résolution d'affichage minimale sera de 640*480 en 256 couleurs.

Environnement réseau

L'application fonctionnera en mode mono-poste.

Développement de l'application

- Environnement de développement : Microsoft Visual c++ 6.0.
- Environnement d'exécution : Windows
- La partie conception de l'application s'appuiera sur le logiciel gratuit Rational Rose C++ 4.0.

Besoins fonctionnels

L'application doit permettre :

- Un chargement rapide d'une nouvelle partie.
- La sauvegarde et le chargement d'une partie.

Fonctions principales de l’application

L’application comprendra principalement deux modules accessibles à l’aide de menus :

- Un module d’aide.
- Un module « Fichier » classique pour la gestion de la partie : création d’une nouvelle partie, sauvegarde de la partie en cours et chargement d’une partie sauvegardée.

Module « ? »

Ce module a pour but de permettre à l’utilisateur de prendre connaissance des fonctionnalités du logiciel ainsi que des règles du jeu.

Module « Fichier »

Ce module correspond aux fonctions classiques d’un jeu de ce type :
Création et initialisation des données nécessaires à une nouvelle partie ;
La sauvegarde et le chargement des données d’une partie ;
Quitter l’application.

Evolution de l’application

L’application pourra être améliorée en intégrant par exemple :

- Une intelligence artificielle qui permettra au joueur de se mesurer à l’ordinateur
- Une gestion du jeu en réseau local et sur Internet.

Annexes

Glossaire

Environnement de développement : c'est le logiciel qui permettra d'écrire l'application dans un langage déterminé.

Environnement d'exécution : c'est le système d'exploitation (comme par exemple Windows, Unix, Linux…) qui accueillera l'application.

C++ : c'est le langage de programmation, créé par Bjarne Stroustrup en 1982, dans lequel sera écrite l'application.

Mono-poste : Une application fonctionnant en mode mono-poste n'intègre pas de fonctionnalité réseau permettant, par exemple, le jeu multi-joueur distant.

Intelligence artificielle : Le fait d'inclure à l'ordinateur des méthodes pour « penser », la machine doit être programmée avec des connaissances lui permettant de jouer une partie comme le ferait un joueur humain.

Composition de l'équipe

Opfermann Cédric, Arnaud Jean, Vial Mickaël, Millet Aurélien, Hakkou Hicham : Elèves de seconde année en section Informatique à l'IUT2 de Grenoble.

Planning de réalisation

Dates	Travail à effectuer
11 décembre 2003 – 10 mars 2004	Réalisation des parties respectives
10 mars – 17 mars	Mise en commun et rédaction du rapport
17 mars – 29 mars	Préparation de la soutenance

Annexe 2 : Fichier Tobruk.txt

```
mapsize:40,20
map[
ssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssss
sssssssssvppppppssssssssssssssssssssssss
ssssspvppccoooccvppsssssssssssssssssssss
ssssvcccccooocccvppsssssssssssssssssssss
ssspccoococccccccccpvsssssssssssssssssss
sspccoocccccccccccpppsssssssssssssssssss
svpccoocccccppppccpppvpvpvsssssssssssssss
spppcccccpppppppvppppppppppppvssssssssss
spvpcccpppppppvppppppppppppccpvsssssssss
spppppvpppppppppppdddvpppppcccvsssssssss
spvppppppppppppdddddddppvpddcccppvpvppvp
sppppppppppvdddddddddddddddddvcppppppppp
sspppppppppppddddddddddddddddcccppcpppcp
ssppppppppddddddddddddddddddddccccpcccccc
svppppppdddddddddddddddddddddvcccpcccccc
smvpppddddddddddddddddddddddddccpccccccc
mmpppddddddddddddddddddddddddddccccccccc
mpppdddddddddddddddddddddddddddcccccdddd
ppppdddddddddddddddddddddddddddddccddddd
]
```

Annexe 3 : Code source de l’application