Université d'Angers

Réalisation d'un jeu d'échecs tridimensionnel

Auteurs :
Ma Tianning
Malcombre Nicolas

Projet encadré par :
Stéphan Igor

Année : 2018/2019
Projet : avril-mai 2019

Remerciements :

Nous tenons tout d'abord à remercier M. Stéphan de nous avoir conseillés, aidés et pour son implication sur toute la durée de ce projet. Il nous a permis d'enrichir notre projet en nous imposant des contraintes de réalisation, qui nous ont forcés à nous documenter, à nous pencher plus sur certains aspects que nous n'aurions pas forcément approfondis.

Nous remercions également M. Horenovsky pour sa démonstration très claire et pédagogique d'utilisation du solver SAT en C++, appliqué à la réalisation d'un sudoku. Ainsi que M. Geneva pour sa méthode efficace de génération et d'amélioration du terrain par un algorithme stochastique.

Par ailleurs, nous remercions M. Stéphan et M. Genest de nous avoir dispensé des enseignements essentiels pour la réalisation de notre projet.

Enfin, nous remercions toutes les personnes qui ont contribué par leur aide momentanée à améliorer notre projet.

Table des matières

1 Introduction

Au deuxième semestre de la licence 3 à l'université d'Angers, dans le cadre de l'unité d'enseignement 5, nous avons réalisé un projet tuteuré de huit semaines.

Nous avions eu l'idée de créer un jeu d'echecs tridimensionnel inspiré notamment par le plateau présent dans Star Trek et certains plateaux trouvés sur internet.

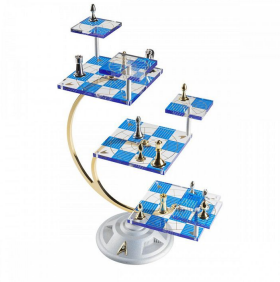


FIGURE 1 – Jeu d'Échecs Star Trek tridimensionnel

FIGURE 2 – Jeu d'Échecs avec un échiquier en 3D

Nous avons donc parler de notre projet à M. STEPHAN qui a accepté d'être notre tuteur. Le projet, lui, a en revanche été modifié au terme de l'entretien de début de projet, pour être plus consistant et intéressant programmatiquement parlant.

L'objectif de ce projet était donc de créer un jeu d'échecs jouable et complet en 3D à l'aide de la librairie OpenGL. Le jeu d'échecs avec les règles classiques doit se dérouler sur un terrain tridimensionnel généré à partir d'un ensemble de triangles. Les pièces doivent être des personnages réalistes et doivent effectuer une animation d'attaque pour tuer.

C'est ainsi que nous allons dans ce rapport, au travers des différentes parties, vous montrer et vous expliquer les étapes de ce projet, de l'idée au jeu "sha mata".

2 Présentation générale du projet

Le projet que nous devions réaliser était composé de deux parties distinctes : une première partie pour l'aspect graphique des personnages, une seconde pour la partie programmation des règles et génération du terrain. Les deux parties pouvant être effectuées conjointement, nous avons tout d'abord commencé par réfléchir à la structure globale du jeu d'échecs.

Les choix pour la réalisation :

Nous nous sommes premièrement mis d'accord pour réaliser ce projet en c++ 14. Ce qui a motivé ce choix sont les classes qui permettent de hiérarchiser et clarifier le code. D'autant plus que toutes les pièces telles le pion, la tour, le cavalier et les autres partagent des caractéristiques communes qui seront représentées dans une classe mère de toutes les pièces. Les caractéristiques qui diffèrent seront donc modélisées par la surcharge des méthodes lors de l'héritage.

Nous avons ensuite étudié les contraintes imposées par le sujet, qui sont communes à la programmation du terrain et à l'aspect graphique, pour définir la direction à suivre pour notre groupe.

Les contraintes :

1. Le terrain doit être construit à partir d'un ensemble de triangles
   (le triangle étant une des primitives graphiques d'OpenGL)
2. Les pièces doivent faire une animation.

Pour fixer la première contrainte, nous avons décidé que l'ensemble de triangles serait représenté par un vecteur de la classe *vector*. Il présente l'intérêt d'avoir une longueur variable et d'être facile d'utilisation. Lorsque nous construirons le terrain nous aurons également besoin de définir ce qu'est un triangle pour l'utiliser dans un vecteur.

L'animation des pièces qui constituent la seconde constrainte sera, elle, définie comme étant déclanchée par un booléen lors de son passage à vrai.

Les contraintes impactant nos 2 parties maintenant définies, nous avons construit ensemble la base des fichiers pour ouvrir une fenêtre OpenGL en c++. Chacun muni d'une copie de cette base nous avons ensuite effectué une partie du projet. Chacune des parties est détaillée dans la section 3, la section 3.1 pour la partie programmation des règles et génération du terrain et la section 3.2 pour la partie graphique en OpenGL.

3 Travaux réalisés et méthodes utilisées

3.1 Programmation des règles et génération du terrain

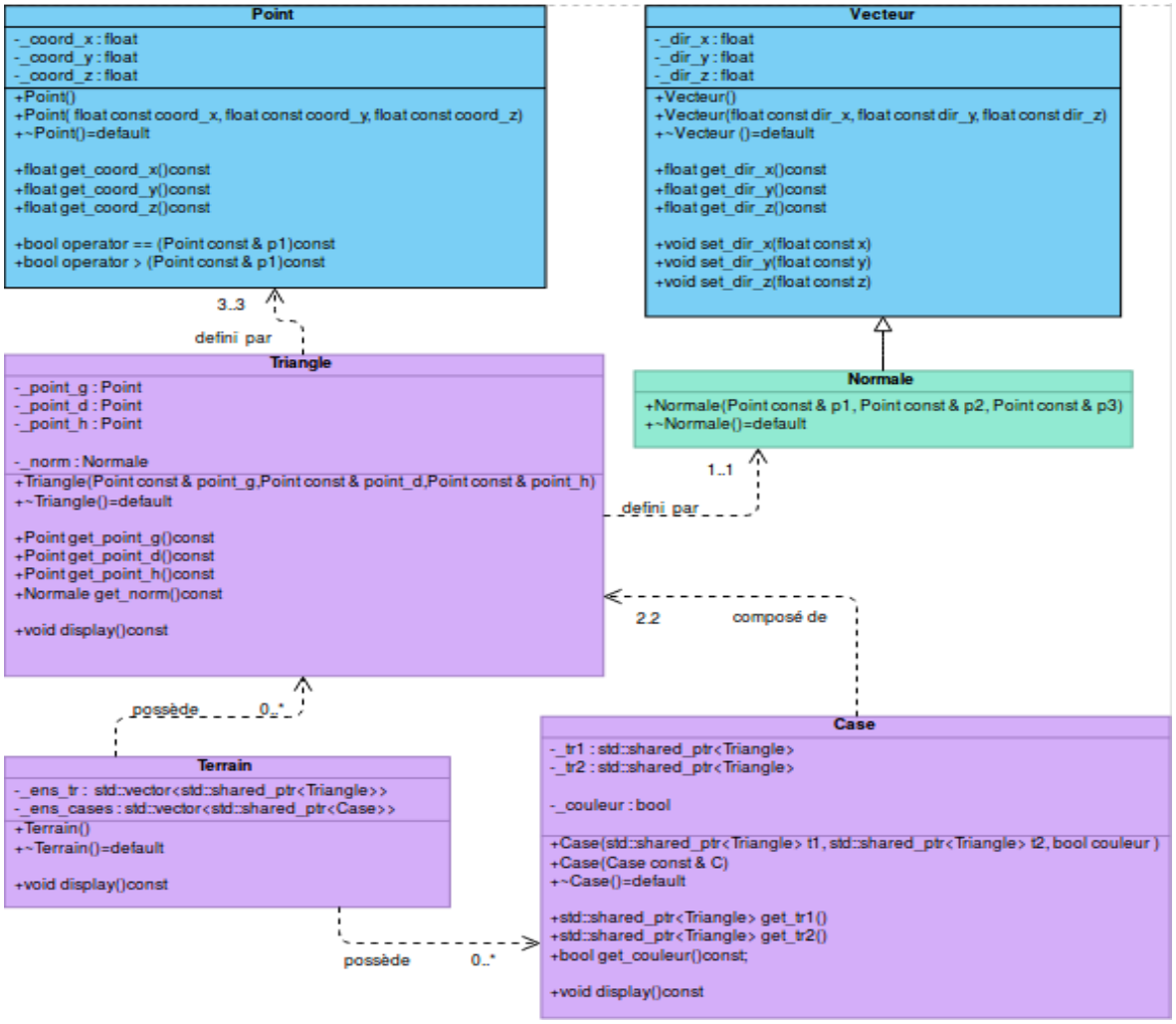
Dans cette section nous allons vous presenter en detail la création du jeu et des règles dans sa globalité.

3.1.1 Création des classes de base

Suite aux contraintes que nous avons fixées avant de commencer il a fallu définir une classe Triangle pour en faire un vecteur qui puisse définir un terrain.

Cependant, un terrain n'est pas composé directement de triangles mais de cases, qui sont quant à elles composées de deux triangles. C'est à ce moment que nous avons opté pour une modification du vecteur de triangles en un vecteur de pointeurs partagés de triangle : *shared_ptr<Triangle>*, ce qui évite ainsi plusieurs représentations identiques en mémoire d'un même triangle et des recopies inutiles.

On obtient donc ce premier diagramme de classe :



3.1.2 Génération du terrain

Pour générer le terrain nous avons créé une méthode qui prend en paramètre l'identifiant du terrain à créer et en fonction appel la méthode de construction associée. Les fonctions de construction ont pour seul et unique but de remplir le vecteur de triangles pour former un plateau tridimensionnel.

Ensuite, cet ensemble est analysé pour créer les cases par deux méthodes. Dans un premier temps nous allons vous expliquer la première méthode et ensuite pourquoi il fut nécessaire d'en créer une seconde.

Méthode 1 :

Cette méthode vérifie que l'ensemble de triangles n'est pas vide, car cela signifierait qu'il n'y a pas de terrain sur lequel jouer. Elle vérifie ensuite que le nombre de triangles n'est pas impaire, car dans le cas contraire il restera forcément à la fin un triangle seul pour faire une case.

Une fois les vérifications effectuées on prend arbitrairement le premier triangle de l'ensemble, en effet les triangles peuvent ne pas être ordonnés. On passe ensuite ce triangle à la méthode *faire_case* qui associe les triangles deux à deux pour créer les cases.

Mais comment associer les triangles adjacents ?

Cette question nous a posé beaucoup de difficultés. Voici quelques exemples de pistes infructueuses pour résoudre ce problème :

-piste 1 : Orienter chaques triangle pour l'associer avec celui plus au "nord". En appliquant une orientation arbitraire aux triangles, on détermine un triangle situé au "nord" mais cela ne marche pas. En effet, on ne commence pas toujours par un triangle qui doit être associé à celui plus au "nord". Dans l'exemple figure 3 le triangle 0 doit être associé au triangle 5 ce qui force le triangle 4 à être seul. En revanche, si le premier triangle était le triangle 3, l'association avec celui au "nord" aurait fonctionné.

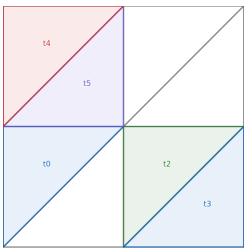


FIGURE 3

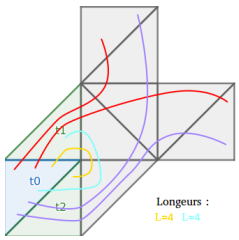


FIGURE 4

-piste 2 : Compter les triangles. Si l'on compte les triangles non-utilisés dans la direction "la plus courte" on peut savoir si l'associer au premier sera un succès (longueur impaire) ou un échec (longueur paire). Cela implique de déterminer tous les chemins de triangles possibles. Dans l'exemple figure 4 pour déterminer si on associe t1 ou t2 à t0 on doit faire 6 parcours. La figure 4 est également un contre exemple qui ferait que l'algorithme ne pourrait pas associer les cases t1 et t2 à t0 car les chemins les plus courts partant de chaque case sont de longueur paire.

La solution adoptée :

Nous avons en main un premier triangle aléatoire. Dès le choix du 2ème triangle il y a une chance sur trois pour que ce choix conduise à un échec comme l'illustre la figure 5 Nous avons également deux choix possibles pour tous les triangles suivants.

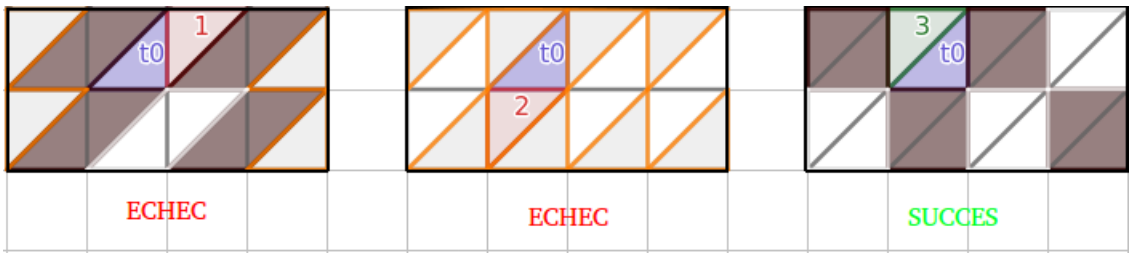


FIGURE 5 – choix du triangle 1,2 ou 3 pour associer

Le principe de l'algorithme est donc d'essayer de faire une case et si nous ne pouvons pas faire le plateau car la case était une mauvaise association, on annule et on essaye une autre association de case.

Voici l'algorithme qui permet de faire les cases :

faire_case

```
Prendre les triangles adjacents au _triangle
s'il n'y en a pas le _triangle ne peut pas être associé : return false.
pour chaque triangle adjacent :
|    si le triangle n'est pas dans une case
|    |    on sauvegarde les cases déjà crées.
|    |    on associe le triangle avec le _triangle en case de couleur _couleur
|    |    pour chaque triangle adjacent à la case ainsi formée :
|    |    |    si le triangle n'est pas dans une case
|    |    |    |    on appelle faire_case( triangle , couleur opposée à _couleur )
|    |    |    |    si faire_case() retourne faux on quitte la boucle : break.
|    |    si faire_case() est retourné faux
|    |    |    l'association en case des 2 triangles était mauvaise
|    |    |    on retourne à l'état de la sauvegarde.
|    |    sinon l'association en case était la bonne : return true.
si aucune association n'a marché : return false.
```

Comme vous avez dû le constater cet algorithme devient exponentiel en temps s'il ne fait pas un bon choix d'association. En effet il testera toutes les possibilités des cases après celle qui est mauvaise, avant de détruire la mauvaise association. Dès le début il a, tel que montré par la figure 5, 1 chance sur 3 de donner une solution en temps raisonnable. Chance qui diminue en fonction du nombre de triangles (car il y a plus de risques de se tromper). Le nombre d'appels lui est majoré par le nombre de triangles. Comme cette méthode ne donne pas forcément un résultat en un temps acceptable, nous en avons parlé à notre tuteur M. STEPHAN qui nous a expliqué que notre problème pouvait être résumé à **un problème de satisfiabilité** et qu'il existait déjà des outils dénommés **solver SAT** permettant de résoudre ces problèmes. C'est ainsi que nous avons créé la seconde méthode.

Méthode 2 :

Avant de décrire le fonctionnement de cette méthode, nous allons expliquer ce qu'est un problème de satisfiabilité et un solveur SAT.

Un problème de satisfiabilité ou problème de satisfaisabilité booléenne est le fait de dire s'il existe une solution qui vérifie une formule de logique propositionnelle.

Une formule de logique propositionnelle est composée des variables booléennes de valeur $True$ ou $False$ et de connecteurs booléens "*et*"($\wedge$), "*ou*"($\vee$), "*non*"($\neg$) exemple : $A \vee B \wedge \neg(A \vee B)$. Ici le problème est satisfiable, en effet pour $A = True$ et $B = True$: $(True \vee True \wedge \neg(True \vee True)) = True$).

Un solver SAT résout le problème et dit s'il existe ou non une valuation pour laquelle la formule de logique propositionnelle passée en paramètre est vraie. Si une valuation existe, il peut nous la renvoyer.

Passons désormais à la méthode. Comme la précédente, elle vérifie que l'ensemble de triangles n'est pas vide et que le nombre de triangles n'est pas impair. Ensuite, elle implémente un solver SAT.

Nous avons choisi le solver SAT "minisat" en version 2.2.0. Il présente l'avantage d'être performant (gagnant de toutes les categories industrielles lors de la compétition SAT en 2005), codé en c++ (ce qui facilite son intégration dans notre projet), open source sous licence Copyright ©, ce qui permet de modifier le code et est bien documenté sur internet. Nous l'avons intégré fichier par fichier, car beaucoup de parties de l'archive ne servaient que si on désirait l'installer pour l'utiliser dans un shell, et non si on l'utilisait comme une librairie. Une fois intégré, il faut l'implémenter. Pour cela il y avait deux solutions :

- Remplir un fichier texte au format DIMACS (un format pour les formules de logique) et ensuite le faire ouvrir par minisat qui aurait alors dit si le problème était satisfiable ou non.
- Interfacer directement un objet solver minisat par des variables et lancer sa méthode pour resoudre.

Nous avons opté pour la seconde méthode, plus perforante de part l'absence d'écriture/lecture dans un fichier. Nous ne savons pas s'il y avait possibilité de récupérer la valuation qui vérifiait une formule propositionnelle avec la première solution, car nous avons directement opté pour la seconde.

Pour interfacer minisat, nous avions à disposition des littéraux (mkLit et ~mkLit (équivalent à ¬mklit)) et des vecteurs de littéraux (Vec<Lit>). L'ensemble des clauses formées par ces littéraux doivent être au format CNF pour "Conjonctive Normal Form" c'est-à-dire que les littéraux d'une même clause doivent être reliés de par des "*ou*"($\vee$) et les clauses par des "*et*"($\wedge$). Nous avons donc défini les clauses avec pour litéraux : "Cij" qui signifie le triangle i et le triangle j forment une case et "Ni" la case i est noire.

Voici un exemple de clause et sa transformation en CNF :

La case avec le triangle i et le triangle j est la même case que celle formée avec le triangle j et le triangle i.

$Cij <=> Cji == (Cij => Cji) \wedge (Cji => Cij) == \underline{(\neg Cij \vee Cji) \wedge (\neg Cji \vee Cij)}$

Les autres clauses et leurs transformations sont disponibles en annexe.

Les N premiers littéraux, où N est le nombre de triangles, sont pour représenter Ni la couleur des triangles et ensuite dans une matrice N*N initialiser à -1 pour représenter

Cij. Par la suite si $matrice[i][j] = x$ et $x \neq -1$, cela veut dire que le littéral numéro x nous indique si les triangles i et j forment une case. Il suffit ensuite d'ajouter les clauses à l'objet solver avec : solver.addClause().

Voici un exemple d'ajout de clause :

$(\neg Cij \vee Cji) \wedge (\neg Cji \vee Cij)$
solver.addClause(mkLit($matrice[i][j]$) , ~mkLit($matrice[j][i]$)) ;
solver.addClause(~mkLit($matrice[i][j]$]),mkLit($matrice[j][i]$)) ;

On lance ensuite la résolution du problème avec solver.solve() qui nous retourne un booléan indiquant la satisfiabilité. Si le booléen est à $True$ alors on peut aller consulter les littéraux pour savoir quelles sont leurs valuations.

Voici un exemple de consultation de littéraux :
(solver.modelValue($matrice[i][j]$)==l_True)
si cette condition est vraie alors les triangles i et j forment une case.

La méthode SAT a comme inconvénient de ne pas pouvoir créer le terrain si trois cases sont collées à cause de la contrainte qui oblige 2 cases adjacentes à être de couleurs opposées, ceci est illustré par la figure 6. Nous avons donc mis la méthode 1 après la méthode 2. Cela permet d'avoir une création de terrain rapide dans la majorité des cas et ensuite de tenter de créer le terrain en maximisant la contrainte de couleurs au risque que ce ne soit pas en temps raisonnable.

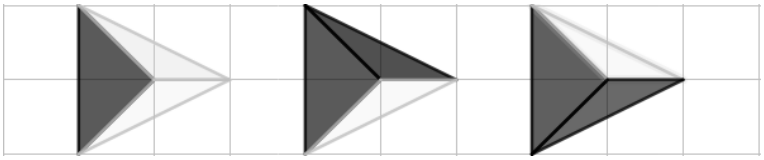
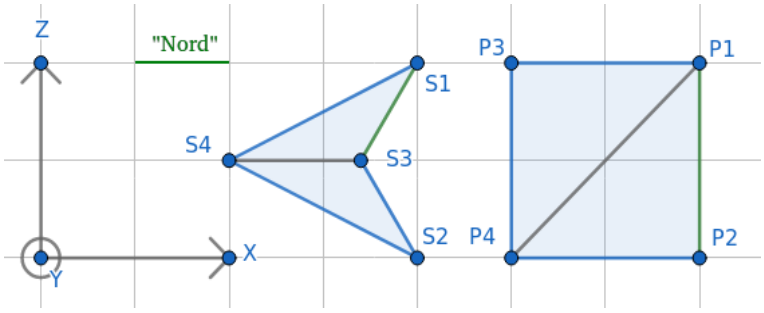


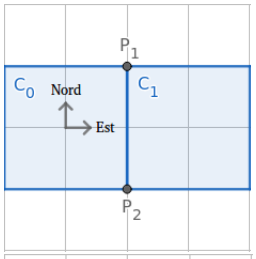
FIGURE 6

La création du graphe de terrain

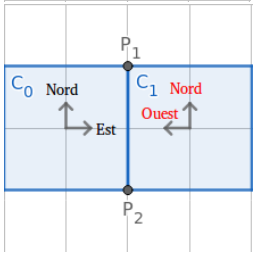
Nous avons donc défini sur la première case une orientation. En fonction des coordonnées des points de la case on établit un ordre, en priorisant l'axe x, puis y, puis z. Si les deux points de la case les plus au "nord" sont les 2 opposés dans la case, le "nord" sera alors entre le premier et le troisième point le plus au "nord".



A partir de cette case, on propage l'information sur l'orientation à toutes les cases adjacentes. Pour expliquer le fonctionnement, nous nous baserons sur un exemple concret.

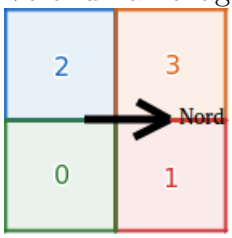


Ici C_0, qui a été orientée, communique à C_1 les deux points P_1, P_2 qu'elle a en commun avec elle. Elle communique également la direction dans laquelle est C_1 : «C_1 est à l'Est de C_0».



C_1 doit alors s'orienter pour être cohérente avec C_0. Comme P_1 et P_2 sont à l'Est de C_0, ils doivent être à L'Ouest de C_1. Une fois C_1 orientée, elle transmettra à son tour l'information à ses cases adjacentes (sauf C_0).

Voici un affichage du graphe ainsi obtenu, avec un damier de 4 cases :




```
,case 0( [1,N,2,N] )
,case 1( [N,1,3,N] )
,case 2( [3,N,N,0] )
,case 3( [N,2,N,1] )
```

3.1.3 Création du jeu et des règles

La première chose que nous avons faite c'est de créer une pièce (un simple cube) et d'essayer de l'inclure dans le terrain. Nous avions dans un premier temps fait gérer le jeu par le terrain. Les pièces étaient sur des cases et les cases appartenaient au terrain, l'ensemble des pièces appartenait au terrain. Mais pour dire telle pièce va de la case où elle est à telle autre, la pièce devait connaitre sa case, ce qui créait un cycle d'inclusion, preuve d'une mauvaise modélisation de notre part. Nous avons repensé et amélioré la modélisation pour obtenir cela :

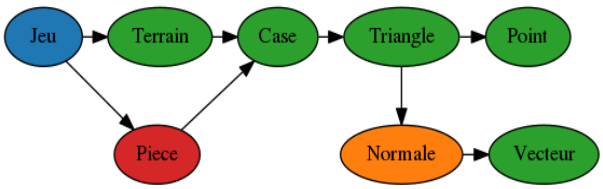


FIGURE 7 – schema d'inclusion des classes principales

Ainsi le "jeu" connait les règles, l'ensemble des pièces et gère le terrain. Le terrain ignore l'existence des pièces (on pourrait mettre des cailloux si on veut, seul le jeu et ses règles changeront). Une pièce connait la case où elle est posée et donc peut déduire comment elle doit être dessus. Le jeu réagit en fonction des actions pour donner des ordres de déplacement aux pièces et au terrain.

Nous avons ensuite défini les contrôles pour placer les pièces, les sélectionner, les désélectionner, sélectionner une autre pièce alliée, cela en fonction du joueur qui doit jouer et de l'état du jeu. Nous ne nous étendrons pas sur ces points, en revanche nous allons détailler un peu plus le fonctionnement des déplacements et attaques.

- Les déplacements et attaques :

Premièrement, lors de la sélection de la pièce le jeu, on regarde le type de la pièce et, en fonction, on appelle une méthode de calcul du déplacement intrinsèquement dépendante des règles que l'on applique au jeu. Une fois les cases où on peut aller connues, on leur demande de se colorer en rouge, cela pour signifier que l'on peut aller dessus. Si on sélectionne une case sans pièce et non colorée, il ne se passe rien car ce n'est pas un déplacement possible. En revanche, si on sélectionne une case rouge il y a deux choix :

- il n'y a pas de pièce ennemie $\mapsto$ on se déplace juste.
- il y a une pièce ennemie $\mapsto$ on attaque.

Dans les 2 cas on appelle la méthode qui, en fonction du type de pièce, cherche le chemin pour aller de la case sur laquelle est la pièce sélectionnée à la case rouge ou l'on veut se rendre. Le chemin défini, on l'envoie à la pièce avec la méthode de déplacement ou d'attaque en fonction de la présence d'une pièce ennemie à l'arrivée.

Pour un déplacement, la pièce suit le chemin que nous lui avons donné. Le jeu, lui, surveille et une fois que la pièce a terminé il change le tour de jeu.

Pour une attaque, la pièce fait une partie du déplacement puis fait son animation d'attaque. Le jeu qui surveillait fait mourir la pièce cible au moment où la pièce fait le mouvement fatidique. La pièce attend ensuite la fin de son animation et que le jeu lui dise qu'elle peut reprendre son déplacement et le finir.

- Evénement particulier :

le rock :

Le rock est le déplacement du roi près de la tour puis la disparition de la tour et sa réapparition de l'autre côté du roi. Ici le jeu attend simplement que le roi soit en place pour déplacer la tour en la tuant et ressuscitant de l'autre côté.

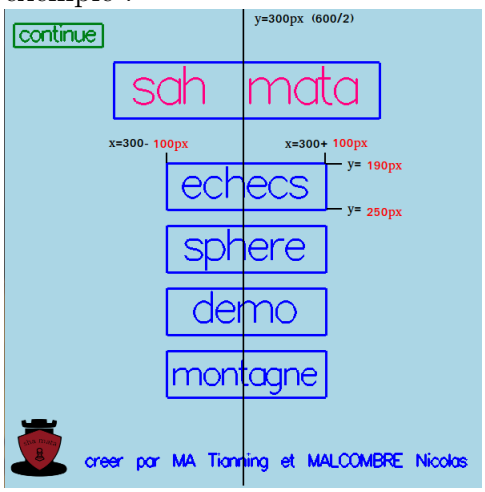
la promotion :

Si un pion atteint une extrémité du plateau il se réincarne en une pièce allié morte précédemment. Ici le jeu attend que le pion soit en position et ait fini tous ses déplacements et attaque pour afficher un menu cliquable permettant de sélectionner le type de la pièce a réincarner (cf. annexe figure 18) . Une fois le type validé on tue le pion et on ressuscite une pièce du type validé.

- Les menus :

Pour rendre l'utilisation plus facile et accessible nous avons créé des menus, dont les rendus sont disponibles en annexe. Les menus OpenGL sont créés en listes d'affichage et l'action de clic est gérée par le jeu. Les hitbox pour cliquer sont définies sur la fenêtre de taille minimale 600*600. Lors d'un redimensionnement, seule la hauteur et la distance à l'axe vertical central de l'écran varie. Ainsi, par un produit en croix on peut, en fonction de la hauteur, établir une hitbox pour les boutons.

exemple :



A la taille minimale (600*600), le carde "echecs" se situe entre y=190 et y=250, x=200 x=400 .
si on redimentionne avec une hauteur de 800 et une largeur de 1000 on obtient un cadre entre :
x=1000/2+((200-(600/2))*1000/600)=333 et x=666,
y=(190*800/600)=253 et y=333

L'autre type de menu est le menu glut (cf. annexe figure 19). Il était simple à créer, mais a posé problème lors de l'exécution. En effet, on désirait qu'il ne soit plus accessible lors de la partie or on ne peut pas le désactiver s'il est ouvert. Nous avons donc désactivé le lancement de la partie qui n'utilisait pas le menu (touche du clavier). Ainsi pour lancer la partie on doit cliquer dans le menu qui se fermera avant de se désactiver.

- L'intégration graphique des pièces :

Nous nous étions mis d'accord sur les méthodes, de ce fait nous avons pu plus facilement réunir nos deux parties.

Nous avons tout d'abord inséré les méthodes de création en listes d'affichage qui correspondent à des méthodes "static". Le but est qu'au lancement du jeu nous appelions les méthodes "static" des pièces, qui se chargent de créer les parties fixées communes en listes d'affichage dont le numéro est enregistré sous forme d'attribut "static". Ces listes d'affichage sont ensuite appelées grâce aux attributs "static" dans la fonction qui assemble la pièce (les Listing 1 et 2 section 3.2.2 et 3.2.3 montrent des exemples d'utilisation de ces méthodes et attributs static).

- La création d'une sauvegarde :

Pour permettre aux utilisateurs de l'application de ne pas perdre la progression d'une partie, nous avons ajouté un système de sauvegarde sous forme de fichier texte. Une fonction permet l'enregistrement de l'état complet de la partie en cours. Si une action est en cours lors de la sauvegarde, un affichage d'erreur se produit (cf. annexe figure 21). La restauration d'une partie en cours provoque l'affichage d'un écran d'erreur (cf. annexe figure 22) dans 3 cas :

- Le fichier de sauvegarde est inacessible ou inexistant.
- Le fichier de sauvegarde ne crée pas de terrain.
- Il n'y a pas un roi dans chaque équipe.

3.2 Partie graphique en OpenGL 2.1

3.2.1 Génération du terrain par HeightMap

HeightMap, la carte des hauteurs ou carte des altitudes en français, est une méthode souvent utilisée pour la représentation du relief d'un terrain. L'idée d'une heightmap est d'associer une valeur de hauteur à chaque point de la carte en 2D afin de réaliser ensuite un rendu tridimensionnel.

Divers algorithmes permettent la génération aléatoire de la matrice de hauteur, en garantissant de ressembler à des terrains ou paysages naturels.

Dans ce projet, nous avons choisi la méthode proposée par M. Geneva, qui génère un terrain par un algorithme stochastique. Cette méthode consiste à dessiner aléatoirement les cercles de taille aléatoire sur la carte en 2 dimensions (plan XZ par exemple). Tous les points qui se situent à l'intérieur de ce cercle augmentent leur hauteur. Cela permet de créer un terrain aléatoire, lisse et proche de la topographie naturelle.

```
//...
int nbCercle = rand() % area + num * (8);
//Ajuster les valeurs de hauteur
for(int a = 0; a < nbCercle; a++){
//pour chaque cercle
    //les parametres du cercle
  x = rand() % 16;
  z = rand() % 16;
  radius = rand() % 20 + 1;
    //pour chaque point du terrain
  for (int i = 0; i <= 16; i++){
    for (int j = 0; j <=16; j++){
      //Calculer la valeur de y
      dz = z - i;
      dx = x - j;
    //Calculer la distance entre le point et le centre du
        cercle
      distance = sqrt((dz*dz) + (dx*dx));
      test = distance * 2 / radius;
      if (fabs(test) <= 1.0) {
            // si le point est dans le cercle, ajuster la
               hauteur
        cosVal = test * 3.14 / 180;
        hauteurs[i][j] += (0.3 / 2.0) + (cos(cosVal) * 0.3 /
           2.0);
      }
    }
  }
}
//...
```

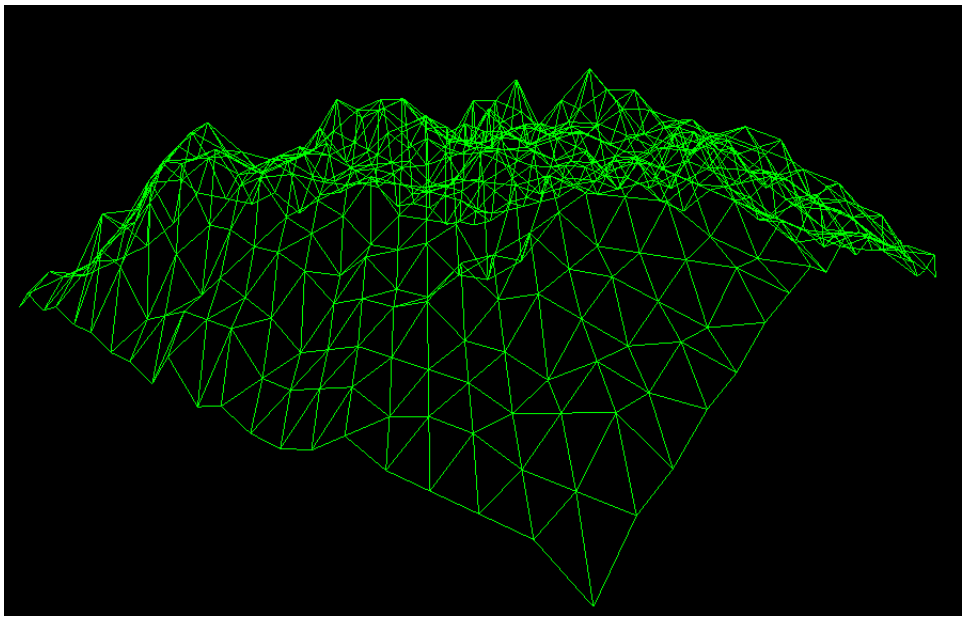


FIGURE 8 – Terrain généré par HeightMap

A noter que la génération du terrain par heightmap peut être aussi codée en utilisant la couleur (une intensité de gris) d'un fichier bitmap (voir figure ci-dessous) où chaque composante RGB du pixel correspond à l'altitude (hauteur) de ce point. Les zones blanches correspondent donc aux hautes altitudes et les zones sombres correspondent aux basses altitudes.

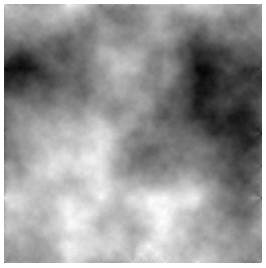


FIGURE 9 – Exemple de HeightMap

D'autres procédures courantes pour la génération du terrain fractal sont :
- l'algorithme Diamant-Carré, qui consiste à calculer la moyenne de quatre points aux alentours formant un carré afin de remplir la matrice de hauteur.
- l'algorithme de Bruit de Perlin, qui détermine la matrice en calculant les produits scalaires entre le vecteur de gradient et le vecteur distance.

Néanmoins, la limite principale d'une HeightMap est l'impossibilité d'attribuer plusieurs hauteurs par point de la carte. Il est donc impossible de créer des terrains intérieurs comme des grottes ou des tunnels. Cela étant, nous ne devions pas créer de grotte ou de tunnel donc l'utilisation d'une heightMap était possible dans notre projet. Nous avons choisi la méthode proposée par M. GENEVA pour l'implémenter au sein du terrain montagne dans notre projet.

3.2.2 Rendu optimisé par Display list

Les Listes d'affichage (Display Lists) sont utilisées dans le but d'optimiser la performance et la rapidité du rendu photo-réaliste. Elles sont souvent utilisées pour le rendu de grands objets qui ont besoin de beaucoup d'appels de "glVertex".

Une liste d'affichage peut stocker dans une zone mémoire réservée, une liste de commande de OpenGL, à laquelle nous pouvons faire appel quand nous voulons. Entre la commande "glNewList" et "glEndList", la liste d'affichage peut collecter toutes les données de vertex en un ensemble, puis le transmettre à la GPU qui le sauvegardera. L'avantage est de pouvoir faire appel directement aux données sauvegardées en mémoire sur la GPU au lieu de les recréer à chaque fois. Cela évite ainsi la transmission répétitive des données vers la GPU.

```
void Pion::load_piece()
{
    //...
  Pion::_piece_pion_lance= glGenLists(1);
  glNewList(Pion::_piece_pion_lance,GL_COMPILE); Pion_DrawLance(); glEndList();
  Pion::_piece_pion_pied= glGenLists(1);
  glNewList(Pion::_piece_pion_pied,GL_COMPILE); Pion_DrawPied(); glEndList();
  Pion::_piece_pion_tete= glGenLists(1);
  glNewList(Pion::_piece_pion_tete,GL_COMPILE); Pion_DrawTete(); glEndList();
  Pion::_piece_pion_main= glGenLists(1);
  glNewList(Pion::_piece_pion_main,GL_COMPILE); Pion_DrawMain(60); glEndList();
  //...
}
```

Listing 1 – Exemple d'utilisation de Display list dans ce projet

Cependant, les listes d'affichage ont quelques défauts. Le plus important d'entre eux, est qu'il est impossible de mettre à jour leur contenu. Autrement dit, une liste d'affichage n'est pas modifiable. Cela pose un problème important sur les objets animés. En effet, dans notre projet, pour faire l'animation des personnages, nous sommes obligés de dessiner à nouveau les parties qui concernent l'animation. Par contre, les listes d'affichage sont très adaptées pour les rendus statiques comme le terrain ou le menu de notre projet.

La liste d'affichage n'est pas la seule technique pour améliorer la performance et la vitesse des présentations graphiques. Les autres techniques telles que VA (Vertex array), VBO (Vertex buffer object) et FBO (Fram buffer object) peuvent permettre de gagner aussi beaucoup de temps de calculs en évitant à OpenGL des allers et retours inutiles vers la GPU.

Dans notre projet, nous travaillons sur la version 2.1 OpenGL. Or certaines fonctions pour utiliser ces méthodes ne sont accessibles qu'à partir de la version 3.0 d'OpenGL, ce qui rend leur utilisation impossible. Par ailleurs, les listes d'affichage sont plus simpleS et efficace à utiliser, donc nous avons choisi cette méthode pour obtenir un rendu graphique optimisé.

3.2.3 Choix pour la modélisation et l'animation des objets

Comme certaines librairies (comme assimp) pour le chargement d'un modèle sur les postes de l'université ne sont pas disponibles, nous avons dû créer les personnages grâce à la librairie GLUT en définissant chaque vertex. Nous avons ensuite pu ajouter et créer des animations souhaitables pour chacune des parties des pièces.

La modélisation des pièces dans notre projet se fait vertex par vertex. Nous travaillons d'abord sur les formes primitives comme le cercle, le cube, la sphère et le cylindre en utilisant les triangles. Nous avons choisi de recréer ces formes primitives au lieu d'utiliser les fonctions des formes proposées par la librairie GLUT. En effet, nous avions besoin d'avoir accès à toutes les composantes de l'objet pour pouvoir les texturer, ce que ne permet pas l'utilisation ces fonctions.

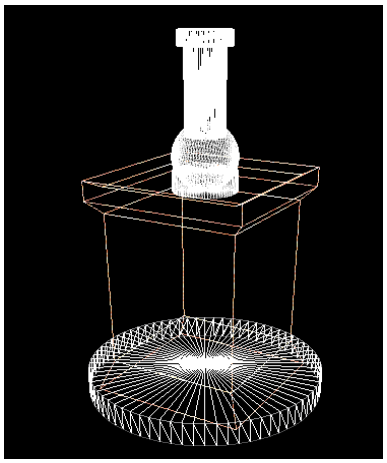


FIGURE 10 – Prototype de la tour

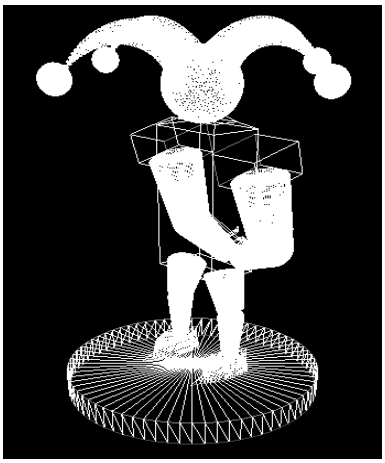


FIGURE 11 – Prototype du fou

De plus, pendant la modélisation, les vertex de chaque face de l'objet sont strictement conformés au sens contraire des aiguilles d'une montre (counterclockwise). Dans notre projet, nous n'avons pas utilisé de lumière mais uniquement une lumière ambiante générale. Il est en revanche toujours possible d'ajouter les calculs des normales et les lumières de manière aisée grâce à la modélisation des vertex en counterclockwise.

FIGURE 12 – Modélisation finale des pièces

Dans le but d'avoir un rendu optimisé et un jeu fluide, nous divisons les pièces en plusieurs parties. Les parties fixes, celles qui ne sont pas concernées par l'animation, sont mises dans les listes d'affichage. Les parties animées seront de ce fait constituées, des parties fixes reliées par les transformations qui évolueront progressivement pour réaliser l'animation (cf. Listing 2). Les transformations sont : la translation, la rotation et la mise à l'échelle.

```
void Cavalier::DrawCavaliers() {
    //...(Changement des valeurs pour les transformations selons differentes
        phases de l'animation)...
    glPushMatrix();{
        //=================Partie cheval====================
        //...(Transformation)...
        Cavalier::Cavalier_DrawCheval();
        //...(Transformation)...
        //=================Partie cavalier=================
        glPushMatrix();{
            glPushMatrix();{
                //...(Transformation)...
                glCallList(_piece_cavalier_Jambe1);//DrawJambe(45);
                 //...(Transformation)...
                //=======================================
                glCallList(_piece_cavalier_corps);//DrawCorps();
                 //...(Transformation)...
                glCallList(_piece_cavalier_epaule);//DrawEpaule();
                glCallList(_piece_cavalier_brasGauche);//DrawBrasGauche();
                 //...(Transformation)...
                glCallList(_piece_cavalier_tete);//DrawTete();
                 //...(Transformation)...
                glCallList(_piece_cavalier_Jambe2);//DrawJambe(-45);
                 //...(Transformation)...
                glCallList(_piece_cavalier_epaule);//DrawEpaule();
                Cavalier_DrawBrasDroite(); // Animation interne
                //...(Transformation)...
            }
            glPopMatrix();
            //...(Transformation)...
            glCallList(Piece::_piece_piece_pied);//DrawPied();
            //...(Transformation)...
            glCallList(Piece::_piece_piece_pied);//DrawPied();
            //...(Transformation)...
        }
        glPopMatrix();
        //...(Transformation)...
    }
    glPopMatrix();
    if (get_joueur()) glCallList(Piece::_piece_piece_baseArgent);
    else glCallList(Piece::_piece_piece_baseOr);
}
```

Listing 2 – Exemple de la modélisation pour la pièce cavalier

Dans le monde cinématographique, l'animation est réalisée par une séquence d'images. Une image (frames) est projetée toutes les 24 secondes pour représenter une animation. Si vous regardez 24 images différentes par seconde, le cerveau va les mélanger en une animation fluide. Les animations sur ordinateur utilisent le même procédé. Les écrans généralement s'actualisent (dessinent à nouveau l'image) pour fournir approximativement 60 à 76 images par seconde. Certains peuvent même atteindre 120 images par seconde. Manifestement, 60 images par seconde représentent une animation fluide, c'est pour cette raison que nous pouvons réaliser l'animation des pièces sur notre projet.

En OpenGL, les transformations sont réalisées principalement par l'instruction d'OpenGL comme glRotatef et glTranslatef. La difficulté essentielle pour faire une animation cohérente et correcte est de placer correctement les transformations. Etant donnée qu'OpenGL est une machine d'état et les transformations se font sur la scène, nous pensons à remettre dans l'origine à chaque fois après une transformation, en utilisant les transformations inverses pour "annuler" la transformation précédente. C'est de cette façon que nous pouvons faire une transformation correcte.

Il existe également, une autre possibilité pour l'animation qui consiste à sauvegarder la pièce dans son intégralité à chaque instant de l'animation en listes d'affichage. Cependant, l'inconvénient est le nombre de listes dont nous avons besoin qui serait très important. Imaginons un rendu à 60 images par seconde, une animation de deux secondes a donc besoin d'au moins 120 listes. Il y aura en revanche des parties fixes qui seront redondantes dans chaque liste d'affichage ce qui n'est pas forcément souhaitable si la partie fixe est relativement volumineuse.

Enfin, une méthode plus avancée et plus technique est de faire l'animation en définissant un squelette après le chargement du modèle selon un fichier OBJ, ce qui permet de dissocier la tâche de modélisation pour les infographistes et celle de l'animation pour les animateurs. Cette méthode s'appelle « Skeletal».

Pour chaque sommet du squelette, on définit les parties influencées par ce sommet et une fois cela défini, nous pourrions animer le squelette et les sommets du modèle suivront. Etant donné que les animations dans notre projet ne sont pas très complexes et que les librairies ne sont pas disponibles sur les postes de l'université, nous avons finalement choisi la première méthode pour présenter l'animation.

4 Conclusion

Notre projet prend fin sur la réalisation du jeu d'échec jouable et complet sur un plateau classique et les plateaux tridimensionnels, permettant de jouer en multi-joueurs tour par tour.

La réalisation de ce projet nous a permis de mettre en pratique beaucoup de connaissances acquises lors de la formation et même d'aller plus loin. Par exemple, dans la partie graphique nous avons pu approfondir le fonctionnement de la souris, la création des menues, les animations, les listes d'affichage qui n'ont pas été abordées dans le cadre des cours.

De plus, dans la partie de gestion des règles, nous avons rencontré au départ un problème de conception et de modélisation sur l'ensemble du jeu. Après réflexion et discussion, nous avons choisi une meilleure représentation de notre jeu. Cela nous a permis d'obtenir une vision plus précise de la modélisation d'un projet et des erreurs possibles lors de sa réalisation, ce qui sera bénéfique pour nos futurs projets, qu'ils soient personnels ou professionnels.

Ce projet nous a permis de découvrir une des applications concrètes d'un solver SAT et des formules propositionnelles, ainsi que d'utiliser les outils que nous avions vus lors du cours de logique avec M. Stephan.

Cela nous a permis de découvrir le travail en groupe avec chacun des objectifs différents pour la réalisation des tâches, mais un seul et unique objectif final : le projet achevé et fonctionnel. L'évolution des tâches au cours de ce projet est consultable dans la section 5. Le travail en groupe nous a aussi permis de nous améliorer dans la clarté du code, il fallait en effet que nous puissions aisément comprendre et relire le code de l'autre.

Ce que nous retiendrons de ce projet c'est qu'il fut une première expérience très formatrice et enrichissante sur les plans technique et humain.

5 Gestion du travail

No	Partie graphique en OpenGL (par MA Tianning)	Partie du développement en C++ (Par Malcombre Nicolas)
	Semaine 1 : 01/04/2019 - 05/04/2019	
1	Ajout du fonctionnement de souris dans la fenêtre OpenGL	Création de la base en C++14 pour l'initialisation
2	Création du terrain simple (Damiers) et intégration dans les classes	Création des classes pour la modélisation du terrain
3	Création d'un prototype du personnage (avec les formes primitives GLUT)	Création de fonctions spéciales aux classes
4	Création des formes primitives par un ensemble de triangles	Tentatives de génération du terrain avec le vecteur de triangle
5	Tentatives de charger des modèles OBJ pour la modélisation	Suite tentatives de génération du terrain avec le vecteur de triangle
6	Documentation sur Display list	
	Semaine 2 : 08/04/2019 - 12/04/2019	
1	Display list, listes d'affichage hiérarchisées et réécrire les personnages	Suite tentatives de génération du terrain avec le vecteur de triangle
2	Documentation sur la génération du terrain complexe	Conception théorique du graphe de déplacement
3	Ajouter les animations possibles au personnage (prototype)	Réalisation de l'orientation arbitraire des cases
4	La modélisation éventuelle de la pièce (Pion)	Réalisation du graphe déplacement et débogage
5		Documentation sur les méthodes et algorithmes SAT
	Semaine 3 : 15/04/2019 – 19/04/2019	
1	La modélisation de la pièce (Tour)	Recherche des clauses CNF et début de la fonction SAT
2	La modélisation de la pièce (Fou)	Suite réalisation de la fonction SAT
3	La modélisation de la pièce (Dame)	Mettre au propre le code – discussion sur la conception du jeu.

4	Ajouter les animations possibles à la pièce (Tour)	Sélection des cases
5	Ajouter les animations possibles à la pièce (Fou)	Placement des pièces et tentative sélection pour deplacement
6	Améliorer les fonctions pour les formes primitives (Cylindre/sphere)	
	Semaine 4 : 23/04/2019 – 26/04/2019	
1	Tentatives de texturer entièrement la pièce pion	Orientation des pièces manière 1
2	Finir l'animation sur la pièce Fou	Graph d'inclusion DOT – déplacement pion et tour de jeu
3	Modélisation des armes pour Dame	Tour de jeu
4	Ajouter les animations possibles à la pièce (Dame)	Deplacement / attaque toutes les pieces 1
5	Modélisation de la pièce Roi	Deplacement / attaque toutes les pieces 2
6	Ajouter les animations possibles à la pièce (Roi)	
	Semaine 5 : 29/04/2019 – 03/05/2019	
1	Modélisation de cheval du cavalier	Gestion des pieces clarifiée pour l'attaque et le deplacement
2	Mettre en commun pour la pièce pion	Rock et debut promotion pion
3	Modélisation de la pièce cavalier	Promotion pion manière 1
4	Animations de la pièce cavalier	Promotion pion manière 2
5	Recherche des textures	Début des listes d'affichage et écriture GLUT (bitmap/solid)
	Semaine 6 : 06/05/2019 – 10/05/2019	
1	Appliquer et améliorer la liste d'affichage sur toutes les pièces	Ecran sélection plateau menu glut placement piece
2	La création du Menu (click droite) par GLUT	Debug menu start alors que ouvert
3	Texturer entièrement les formes primitives	Orientation des pièces manière 2
4	Texturer complètement les pièces	Orientation progressive

5	Mettre en propre les codes pour la modélisation	Début intégration personnages – Ecran de sélection de Promotion
	Semaine 7 : 13/05/2019 – 17/05/2019	
1	Intégration de toutes les pièces dans les classes (en ajoutant toutes les listes d'affichage dans la classe jeu)	Ecran de victoire - Correction de bug promotion - placement auto des pieces
2	Amélioreration du terrain généré par HeightMap pour que le terrain soit lisse	Sauvegarde en fichier texte
3	Tentatives de créer skybox pour espace	Debug de la sauvegarde et début du rapport
4	Commencer à rédiger le rapport (la structure, les parties)	Rapport pour la structure en Latex
	Semaine 8 : 20/05/2019 – 24/05/2019	
1	Rapport	Rapport
2	Préparation de la soutenance	Préparation de la soutenance

6 Bibliographie et Webographie

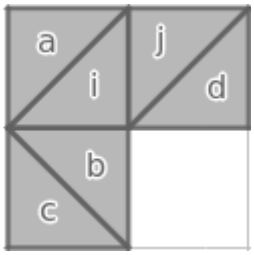
Type	Description/Auteur	Lien/Titre
Site	Cours L3 de M. Stéphan	Igor Stéphan / Enseignement
Livre	S.Dave, S.Graham, K.John, L.Bill	«OpenGL Programming Guide»
Source	Geneva Smith	Génération de terrains
Site	Yahiko	Génération aléatoire de terrains
Site	Bertrand Masson	Tutoriel sur les tableurs en latex
Site	Martin Hořeňovský	Modern SAT solvers (Part 1 of N)
Site	Cours L3 de M.Genest	David Genest / Enseignement

Outils pour la réalisation du projet

Type	Description/Auteur	Lien/Titre
Site	logiciel de mathématiques	GeoGebra
Site	Création de diagrammes de classes	Free Visual Paradigm Online
Site	Overleaf, Éditeur LaTeX en ligne	Overleaf
Site	Convertisseur pour la texture	convertisseur jpg en ppm

7 Annexes

Logique propositionnelle



Contraintes pour le triangle i :

équivalence des 2 cases avec le même triangle :

$$C_{ij} <=> C_{ji} \ ; \quad C_{ia} <=> C_{ai} \ ; \quad C_{ib} <=> C_{bi}$$

i doit forcement être rattaché à un triangle :

$$C_{ij} \vee C_{ia} \vee C_{ib}$$

et un seul :

$$\begin{array}{lll} C_{ij} & => & \neg C_{ia} \ \wedge \ \neg C_{ib} \\ C_{ia} & => & \neg C_{ij} \ \wedge \ \neg C_{ib} \\ C_{ib} & => & \neg C_{ij} \ \wedge \ \neg C_{ia} \end{array}$$

Si nous avons C_{ij}, alors i et j sont de la même couleur :

$$\begin{array}{lll} C_{ij} & => & (N_i \wedge N_j) \vee (\neg N_i \wedge \neg N_j) \\ C_{ia} & => & (N_i \wedge N_a) \vee (\neg N_i \wedge \neg N_a) \\ C_{ib} & => & (N_i \wedge N_b) \vee (\neg N_i \wedge \neg N_b) \end{array}$$

Les autres triangles doivent être de couleurs différentes :

$$C_{ij} \wedge N_i => \neg N_a \wedge \neg N_b \quad ; \quad C_{ij} \wedge \neg N_i => N_a \wedge N_b$$
$$\ldots$$

Clauses en CNF

- $C_{ij} <=> C_{ji}$

$$\underline{(C_{ij} => C_{ji}) \wedge (C_{ji} => C_{ij})}$$
$$(\neg C_{ij} \vee C_{ji}) \wedge (\neg C_{ji} \vee C_{ij})$$

- $C_{ij} \vee C_{ia} \vee C_{ib}$
- $C_{ij} => \neg C_{ia} \wedge \neg C_{ib}$

$$\underline{\neg C_{ij} \vee (\neg C_{ia} \wedge \neg C_{ib})}$$
$$(\neg C_{ij} \vee \neg C_{ia}) \wedge (\neg C_{ij} \vee \neg C_{ib})$$

- $C_{ij} => (N_i \wedge N_j) \vee (\neg N_i \wedge \neg N_j)$

$$\neg C_{ij} \vee [(N_i \wedge N_j) \vee (\neg N_i \wedge \neg N_j)]$$
$$\neg C_{ij} \vee (N_i \wedge N_j) \vee (\neg N_i \wedge \neg N_j)$$
$$[(\neg C_{ij} \vee N_i) \wedge (\neg C_{ij} \vee N_j)] \vee (\neg N_i \wedge \neg N_j)$$
$$[(\neg C_{ij} \vee N_i) \vee (\neg N_i \wedge \neg N_j)] \wedge [(\neg C_{ij} \vee N_j) \vee (\neg N_i \wedge \neg N_j)]$$
$$\underline{[((\neg C_{ij} \vee N_i) \vee \neg N_i) \wedge ((\neg C_{ij} \vee N_i) \vee \neg N_j)] \wedge [((\neg C_{ij} \vee N_j) \vee \neg N_i) \wedge ((\neg C_{ij} \vee N_j) \vee \neg N_j)]}$$

- $C_{ij} \wedge N_i => \neg N_a \wedge \neg N_b$

$$\neg(C_{ij} \wedge N_i) \vee (\neg N_a \wedge \neg N_b)$$
$$(\neg C_{ij} \vee \neg N_i) \vee (\neg N_a \wedge \neg N_b)$$
$$\underline{[(\neg C_{ij} \vee \neg N_i) \vee \neg N_a] \wedge [(\neg C_{ij} \vee \neg N_i) \vee \neg N_b]}$$

Exemples du processus de la modélisation

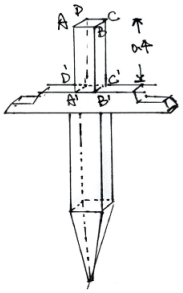


FIGURE 13 – Modélisation pour les dagues de pièce fou

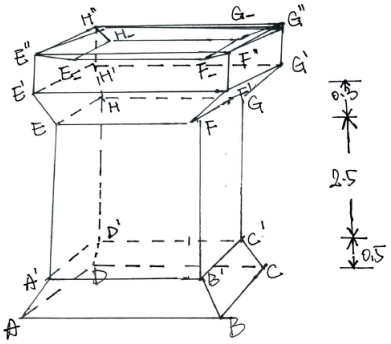


FIGURE 14 – Modélisation pour prototype de tour

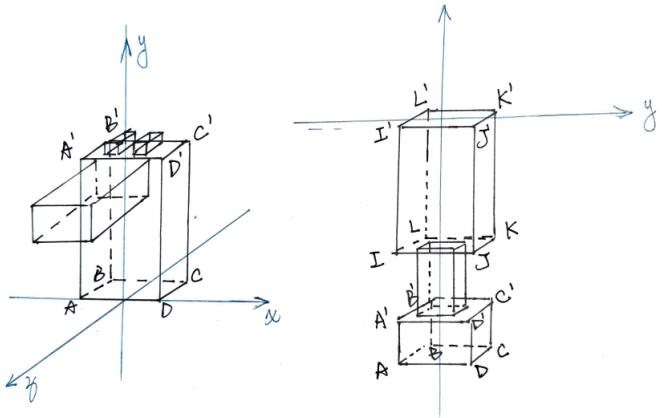


FIGURE 15 – Modélisation pour tête et pied du cheval

Affichage graphique des menus

FIGURE 16 – Menu d'affichage

FIGURE 17 – Menu pour la sauvegarde

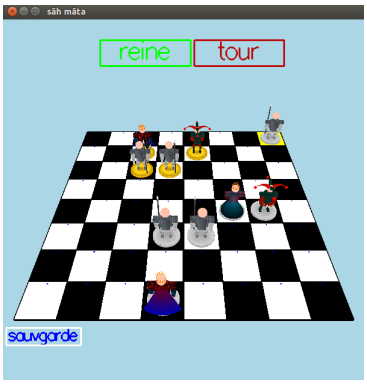


FIGURE 18 – Menu de promotion
(quand un pion atteint le bout du terrain)

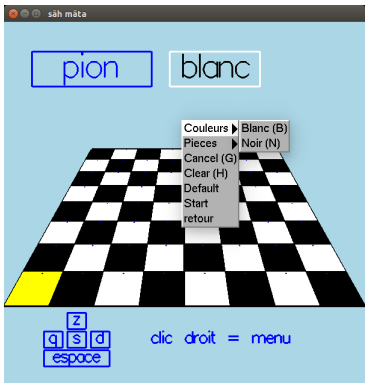


FIGURE 19 – Menu clic droit (GLUT)

FIGURE 20 – Fin de partie

Il existe également des affichages d'erreurs.

FIGURE 21 – Erreur de création d'une sauvegarde

FIGURE 22 – Erreur de chargement de sauvegarde

Exemple d'un fichier de sauvegarde

terrain - 1
joueur - 1
vivante :
case - 29 joueur - 1 move - 1 reine
case - 22 joueur - 1 move - 0 roi
case - 21 joueur - 0 move - 1 roi
morte :
case - 48 joueur - 0 move - 0 pion
case - 59 joueur - 0 move - 1 cavalier
case - 38 joueur - 1 move - 0 roi