LACHAUX Marie-Anne
MARIN Gautier
ETCHEVERRY Mayalen

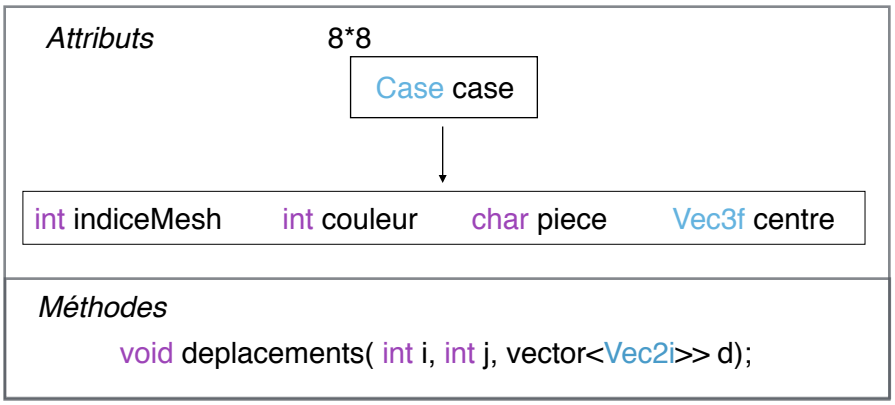
PROJET IGR 202 : JEUX D'ECHECS

I- LE JEU

Classes Damier et Case

Nous avons créer une classe « Damier »et une classe « Case ». Le damier a pour seul attribut un tableau 8*8 de Case.
La case a pour attributs la position de son centre, et des informations sur la pièce qui occupe cette case.

Class Damier :

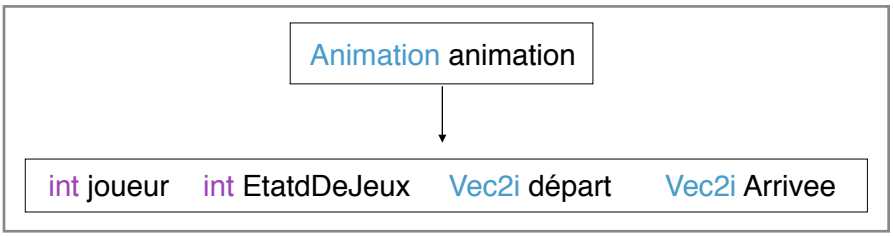


La couleur = 0 (resp. 1) si la case est couverte d'une pièce noire (resp. blanche). La pièce peut notamment valoir 'h' pour horse, 'k' pour king. Une case non occupée est une case de pièce «empty» ie piece='e'. Le centre de la case ne change pas vu que le plateau de jeux est fixe.

La méthode déplacements copie dans d les déplacements possibles de la piece de la case (i,j) du damier. Grâce aux attributs des classes Damier et Case, notre méthode peut prendre en compte la nature de la pièce et l'environnement, tous deux nécessaires au calcul des déplacements possibles.

Classe Animation

La classe animation permet de gérer une partie de jeux.



joueur = 0 si c'est aux noirs de jouer, 1 sinon.

Lorsque c'est à un joueur de jouer, il indique un départ et une arrivée.
départ(resp. arrivee) = (i,j) coordonnées dans notre damier de la case de départ (resp. arrivee).
Etat de jeux = 0 si le joueur sélectionne une case de départ. S'il a deja sélectionné sa case de départ et qu'il s'apprête à sélectionner une case d'arriver EtatDeJeux = 1.

Déroulement du jeu

Les blancs commencent la partie.
Pour déplacer un pion, le joueur utilise sa souris. C'est donc dans la fonction « mouse » qu'on gère l'interaction avec le joueur.
Quand le joueur clique sur un pion, on récupère les indices de la case associée.
Si la couleur du pion sélectionné est bien celle de l'équipe qui doit jouer, la case départ de l'animation devient la case de départ cliquée. Le joueur sélectionne ensuite une case d'arrivée. On vérifie grâce à la méthode déplacements de damier, que le déplacement est possible. Si c'est le cas on déplace le pion, c'est a dire on déplace le mesh. Dans la classe Mesh, on a fait une méthode déplacer. Bien sur on vérifie que la case d'arrivée n'est pas occupée par un pion de la même couleur que celui qui effectue son déplacement.
Après avoir effectué le déplacement du mesh, on change les attributs des cases départ et arrivée, pour qu'elles soient bien actualisées. Si la case d'arrivée est vide, on échange les attributs des 2 cases excepté le centre. Si une pièce a été mangée, la case d'arrivée prend les attributs de la case de départ. La case de départ devient vide.

II- PARTIE GRAPHIQUE

Mesh et SuperMesh

Nous avons fait une classe SuperMesh qui prend en argument un vecteur de Mesh.
Pour obtenir nos pièces, nous avons récupérer un fichier .obj qui contient tous les pions et les répartit en différents groupes. Nous avons implémenté une méthode loadobj de la classe SuperMesh en s'inspirant de la méthode loadoff. Cette méthode charge le .obj et répartit les groupes en différents mesh. Tous ces mesh sont rassemblés dans le SuperMesh.
Nous avons fait plusieurs mesh pour pouvoir les déplacer individuellement, et appliquer des textures différentes.

Pour faire le mesh du plateau de jeux, nous avons créé « à la main » les sommets, les faces et les coordonnées de texture. Nous avons également fait «à la main » la texture du damier sur PixelMator.

Dans la méthode draw(), nous ne considérons que les cases du damier dont les pièces ne sont pas « empty »; et nous dessinons les mesh associés en récupérant « indiceMesh » de la case. Cela permet de ne plus dessiner les pièces une fois qu'elles ont été mangées.

Shaders

Nous avons décider de calculer le rendu par pixel en programmant directement le GPU à l'aide de Shaders.
Nous voulions par ce choix:

- une meilleure performance que celle limitée par une exécution sur CPU de l'ensemble des calculs
- par dessus tout la possibilité de choisir puis modifier à notre guise tous les différents paramètres de réflectance, texture et ombres pour notre rendu

Ainsi, nous avons implémenté les différents éléments suivants à travers nos shaders:

Un modèle de réflectance diffus et spéculaire de Blin-Phong:
Pour cela on a définit deux variables *attribute* (et non pas uniformes car différentes pour chaque sommet) kd et ks dans notre vertex shader.
Ensuite on leur donne la valeur désirée dans notre *DrawScene()* du main. (Pour ce faire on a du modifier notre classe glProgram et ajouter des méthodes permettants de passer au fragment Shader des variables attributes sur le modele de setUniform).
Enfin on récupère ces données dans le fragment shader et on y implémente la BRDF de Blin-Phong.

Une texture:
Après avoir donné le modèle de réflectance, nous voulions plaquer des textures sur nos objets.
Pour cela nous avons procédé en plusieurs étapes.

1) Création d'une classe Texture.
Dans la méthode Texture::charger() on récupère une image jpg,tga,… (cela grâce à la bibliothèque SDL_image) dans une surface SDL. On copie les pixels dans une texture.

2) Dans notre Vertex Shader
On définit une variable *attribute* vt qui va contenir les coordonnées de textures de chaque sommet de notre scène.

3) Dans notre Main
On crée des textures, on les charges dans *init()*, on les envoie au fragment shader en tant que variable uniforme.
Dans notre *drawScene()* on les bind/debind à la texture courante au moment de dessiner chaque pièce.
Pour les coordonnées de texture, on les a définies à la main pour notre damier et pour les pions on récupère les coordonnées vt de notre .obj.
On les envoie donc au fragment shader pendant le drawScene.

4) Dans notre fragment Shader:
On récupère les variables *attribute* vt et les texture en *uniform sampler2D*.
Pour calculer la réponse couleur de nos objets, le fragment Shader va donc multiplier le kd par la couleur de notre texture au coordonnées vt.

Cube Map

Pour texturer le fond de notre scène, nous avons utilisé les Cube Map.
Pour ne pas voir qu'il s'agit d'un cube, nous avons désactivé l'éclairage sur le Cube. En effet, deux faces perpendiculaires n'étant pas éclairées de la même façon par une lumière ponctuelle, les angles sont visibles. Pour les rendre invisible, on donne le vecteur nul comme attribut ks des sommets du cube. Dans le fragment shader, si ks=0, on revoit uniquement la couleur issu de la texture. Ainsi, pour le Cube Map on s'affranchit des «*cos(i) ». On a ainsi une luminosité uniforme sur tout le cube, les angles sont rendus invisibles.

III- DIFFICULTES ET AMELIORATIONS

Faute de temps, nous n'avons pas réussi à debuguer notre algorithme d'ombre par lancé de rayon. Une fois debugué, nous voulions implémenter une structure hiérarchique pour accélérer le calcul, qui est très très lent. Avoir un calcul rapide est nécéssaire pour déplacer les ombres en même temps que les pièces, car il s'agit de recalculer la shadow map à chaque frame.
Nous envisageons également de faire des déplacements progressifs avec « CurrentTime ».
L'échec et mat , le roque et l'interface de jeux (score, démarrage, changement de couleur des pions au click) restent à implémenter.
On aimerait également simuler des réflexions miroirs grâce à la Cube Map, et ajouter des effets spéciaux pour améliorer le projet.