

|  |
| --- |
| **SPEAPING**  APICHESS |

Keyvan Melaini | Julien Hautier | Jean-Charles Gravaillac

|  |
| --- |
| **RAPPORT DE SOUTENANCE**  **Soutenance 1** |

Table des matières

[Introduction 3](#_Toc477193430)

[I) OpenGL 3](#_Toc477193431)

[II) Utilisation de OpenGL 4](#_Toc477193432)

[III) Fonctionnement de OpenGL 4](#_Toc477193433)

[IV) Affichage 5](#_Toc477193434)

[V) Shader 6](#_Toc477193435)

[VI) Matrices 7](#_Toc477193436)

[Conclusion 10](#_Toc477193437)

[Introduction 12](#_Toc477193438)

[VII) The chessboard 13](#_Toc477193439)

[VIII) The pieces 14](#_Toc477193440)

[a) The pawn 14](#_Toc477193441)

[b) The king 14](#_Toc477193442)

[c) The other pieces 15](#_Toc477193443)

# 

# Introduction

**Point positif :**

Projet vraiment intéressant qui nous a permis de mettre en pratique ce que on a vu en C cette année. De plus les documentations effectués nous ont permis d’enrichir notre culture personnelle.

**Point négatif :**

Mauvaise anticipation des structures qui entraîne une complexité accrue.

Segmentation du travail pas très concluante ce qui a entrainé partiellement un retard par rapport au calendrier du cahier des charges.

# Introduction étude de la faisabilité 3D

Pour le premier rendu nous avions a étudié la faisabilité en 3D du jeu d’échec grâce à une API. OpenGL c’est rapidement imposé comme l’API que on utiliserait si on devait faire un rendu en 3D. Vu que c’est la seul API réellement dédié au 3D même si avec SDL il est possible de faire de la 3D également. Nous allons donc maintenant étudier la faisabilité d’un jeu d’échec en 3D.

# OpenGL

OpenGL est une librairie qui exploite les ressources de la carte graphique d’un ordinateur permettant aux développeurs de programmer des applications 2D et 3D. Le principal intérêt d’OpenGL est qu’il est multiplateforme, donc un programme avec OpenGL sera compatible sur un linux come sur un Windows. Nous utiliserons la version 3.X.X de OpenGL, il existe également un autre API disponible, DirectX mais qui lui est utilisable uniquement sur Windows. OpenGL est donc une nécessité vu que nous devons développer et faire fonctionner le programme sous linux. De plus OpenGL est totalement gratuit mais pour OpenGL 3.X.X il faut une GeForce 8000 au minimum de chez Nvidia ou une Radeon HD, ou au-dessus, du côté de chez ATI.

Afin que OpenGL soit fonctionnel il faut avoir recours à la librairie SDL car OpenGL a besoin d’un environnement dans lequel fonctionner, il faut d’abord grâce à SDL créer cet environnement. La librairie SDL a en plus la particularité de fournir les outils pour gérer les évènements afin de savoir ce que fait l’utilisateur. SDL étant multiplateforme cela assurerait une compatibilité entre Windows et OpenGL.

Tout d’abord que ce soit en 3D ou 2D il faudra donc installer les librairies SDL 2.0 et OpenGL.

# Utilisation de OpenGL

Il faudra créer le contexte dans lequel OpenGL pourra s’exécuter pour cela il faudra donc d’abord par le biais de SDL créer une fenêtre. Pour cela il faudra lancer la bibliothèque grâce à la fonction SDL\_Init(), son unique paramètre lui permet de définir les composants que la bibliothèque doit initialiser. Une fois la bibliothèque initialiser il faudra créer une fenêtre via la fonction SDL\_CreateWindow(). Cette fonction retournera un pointeur sur une variable de type SDL\_Window. SDL\_CreateWindow a besoin de plusieurs paramètres (le nom à afficher, la position X/Y, la taille en largeur/hauteur et les options) afin de crée la fenêtre. Il est possible d’utiliser deux valeurs SDL\_WINDOWPOS\_UNDEFINED ou SDL\_WINDOWPOS\_CENTERED afin de déterminer la position de la fenêtre.

La première valeur laisse le libre champ à la bibliothèque de placer la fenêtre où elle le souhaite, la seconde permet de toujours avoir une fenêtre centrée sur l'écran. Pour détruire la fenêtre on fera appel à la fonction SDL\_DestroyWindow() et pour arrêter la SDL on fera appel à SDL\_Quit(). Une fois ceci fait il faudra parametrer des attributs afin de rendre le programme compatible avec OpenGL. SDL\_GLContext contexteOpenGL permet de créer un context OpenGL afin d’utiliser l’API graphique et ses fonctions.

La fonction SDL\_GL\_SetAttribute permet de spécifier des attributs OpenGL à la SDL on pourra donc par le biais de cette fonction choisir par exemple la version de OpenGL que l’on souhaite utiliser. SDL\_GL\_DeleteContext permet de supprimer le context créer.

# Fonctionnement de OpenGL

OpenGL fonctionne grâce à un système de points(vertex) qu’il faut placer dans un monde 3D. Pour les placer dans un monde 3D il faut d’abord leur attribuer une position via 3 coordonnées : x en largeur, y en hauteur et z pour la profondeur. Chaque vertex doit avoir ses propres coordonnées. Les vertices (pluriel de vertex) sont la base, le socle du graphisme. Une fois défini les vertices sont reliés entre eux pour former des formes géométriques tel que des carrés, des triangles ou des lignes (liste non exhaustive).



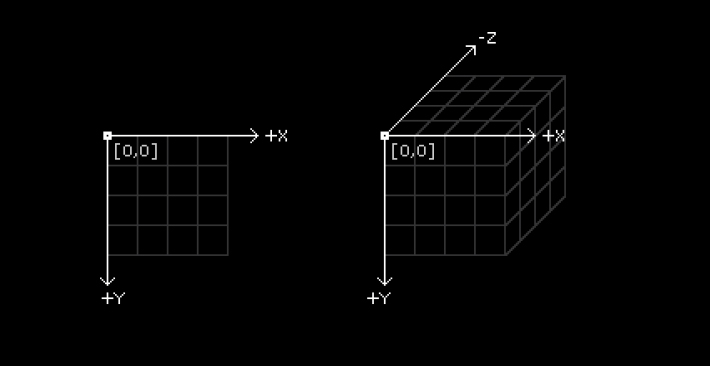
Teiller composé uniquement de vertices que l’on a relié sans application de texture.

Par la suite il faudra appliquer aux formes géométriques des textures afin d’avoir un jeu rempli.

OpenGl fonctionne avec une boucle principale dans laquelle tous les calculs sont faits. Il faudra à chaque affichage effacer l’écran car l’environnement aura changé puis réafficher un à un chaque élément de la scène. La fonction glClear permet d’effacer l’écran et SDL\_GL\_SwapWindow permet avec la SDL d’actualiser l’affichage.

Il existe « une » différence entre Windows car pour utiliser OpenGL sur Windows on est obligé de passer par une librairie au nom de GLEW. Cette librairie va nous permettre d’utiliser OpenGL 3.X.X mais il faudra au préalable l’initialiser dans l’entête du code source.

OpenGl fonctionne selon un repère en 2D ce repère a pour axes X et Y et en 3D X, Y et Z dont l’origine est le point de départ de ces axes.



Repère en 2D à gauche, et en 3D à droite

# Affichage

Pour afficher un modèle 3D il faut spécifier ses vertices puis les envoyer à OpenGL pour qu’il les traites. Or pour envoyer des informations il faut utiliser un tableau « Vertex Attribut ». Ils sont déjà inclus dans l’API, il faut appliquer des valeurs (coordonnées) puis l’activer. La fonction glVertexAttribPointer nous permet de gérer les VertxAttributs mais doit être activé grâce à glEnableVertexAttribArray.

Une fois que OpenGL sait quels vertices afficher, il faut lui dire l’utilisation qu’il doit en faire. Pour cela on utilise la fonction glDrawArrays qui permet de dire à OpenGL quels forment afficher selon les vertices donnés.

# Shader

OpenGL est une librairie qui utilise les ressources de la carte graphique, ressources que OpenGL va utiliser lorsque on fait appel à certaine fonction comme glDrawArrays. Lorsque on fait appel à une de ces fonctions une suite d’opération est déclenché qui est appelé pipeline 3D. Les différentes opérations sont :

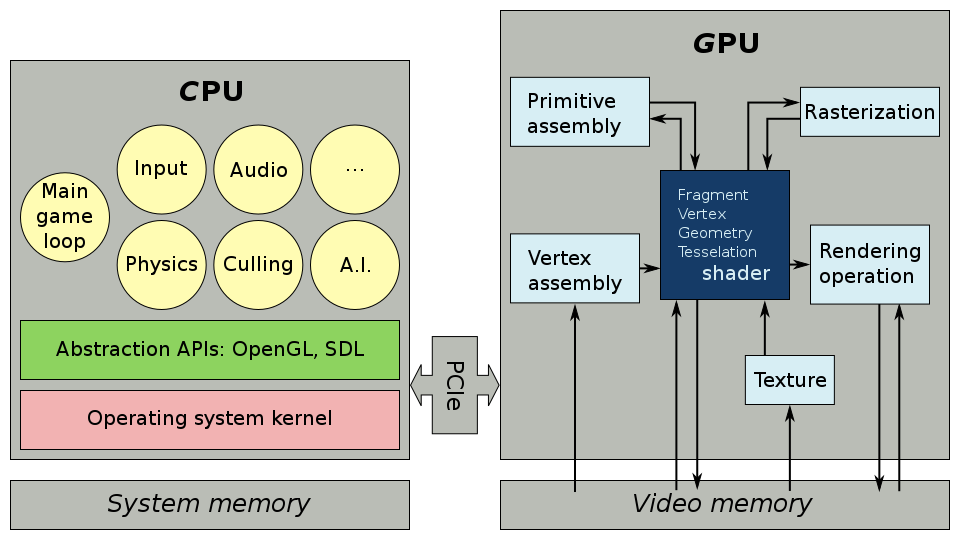
Définition des coordonnées (des vertices) qui est calculé par le CPU, après la définition des coordonnés et l’utilisation de la puissance de calcul du processeur.

On va maintenant faire appel à la puissance de calcul de la carte graphique (GPU) pour l’étape « Vertex Shader » qui va nous permettre de modifier un sommet ou bien de le valider. Vertex Shader sera rappelé autant de fois que le nombre de vertices envoyé.

La troisième étape consiste à convertir les formes géométriques en pixels cette opération est appelé « Pixellisation des triangles »

Une fois les formes géométriques convertie en pixels il faut définir la couleur de chaque pixel délimité pour les vertices. Par exemple pour une fenêtre de 800p\*600p, il faut définir la couleur de 480 000 pixels. Ceci est traité par l’étape « Pixel Shader » (ou Fragment Shader)

Une fois c’est opération effectué le rendu est envoyé par la sortie de la carte graphique jusqu’à notre écran afin de l’afficher.



Schéma

Un shader est un programme exécuté par la carte graphique, utilisé pour paramétrer une partie du processus de rendu réalisé par la carte graphique ou un moteur de rendu logiciel. Il sert à décrire l’absorption et la diffusion de la lumière (par exemple la luminosité sur l’échiquier) ou encore la texture à utiliser, les réflexion et réfraction (par exemple sur de l’eau), l’ombrage des pions et permet même d’appliquer des effets post-traitement.

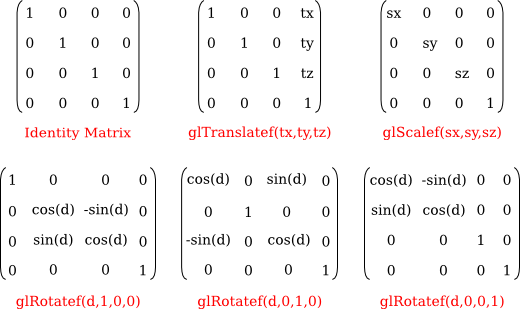
Le Shader étant un programme il doit donc avoir un code source et une compilation. Afin d’utiliser les shaders il faut passer par le constructeur « Shader() », il prend en premier paramètre vertex source qui est le chemin du Vertex Shader et en dernier paramètre fragment source qui est le chemin du code source de Fragment Shader. Pour rendre le Shader exploitable il faudra faire appel à charger() ce qui permettra de lire les fichiers sources et les compiler. La fonction de OpenGL glUseProgram() permet d’utiliser le programme Shader. Afin d’ajouter de la couleur à une forme géométrique on peut utiliser la SDL avec la fonction SDL\_MapRGB qui prend en paramètre la quantité de rouge, de vert et de bleu.

# Matrices

De bonne connaissance sur les matrices en particulier sur les opérations associées, sont indispensable pour l’utilisation de OpenGL. En en effet nous en avons besoin de plusieurs.

1. **Matrice de translation**

La matrice de translation permet de déplacer un ensemble de points ou une forme dans l’espace, cela nous permettre donc de faire bouger les pions par exemples. Voici une matrice de translation.

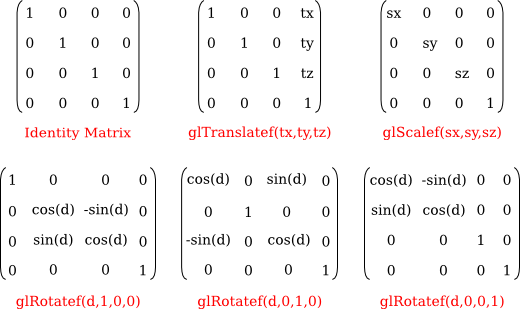


Matrice de translation

Avec tx, ty, tz les coordonnées x, y, et z du vecteur de translation.

1. **Matrice d’homothétie**

La matrice d’homothétie permet d’agrandir ou de réduire une forme géométrique (mise à l’échelle).

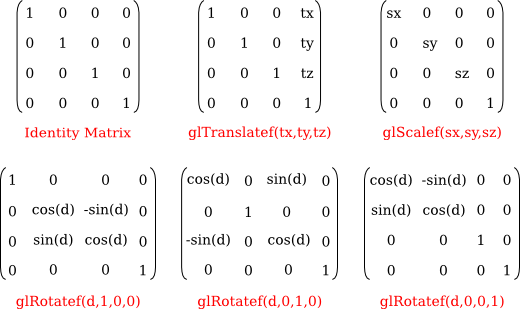


Matrice de mise à l’échelle

Avec sx, sy et sz qui sont les facteurs multiplicatifs des axes x, y et z.

1. **Matrices de rotation**

Les matrices de rotation mais nous ne devrions ne pas en avoir besoin.



Matrices de rotation

1. **Matrice de modèle**

Un modèle est défini par un ensemble de sommet. Les coordonnées des sommets du modèle sont définies par rapport au centre de l’objet. Pour le faire se déplacer on applique la matrice rotation\*déplacement\*redimensionnement à tous les sommets et ce à chaque image.

1. **Matrice de vue**

Afin de déplacer la caméra il faut déplacer le monde virtuelle (dans notre cas toute la scène du jeu d’échec) car la caméra reste fixe à l’origine de l’espace monde. Pour déplacer le monde il faudra introduire une nouvelle matrice.

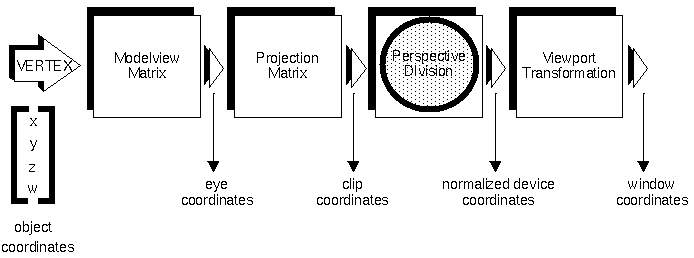
1. **Matrice de projection**

La matrice de projection permet de donner un effet de perspective en jouant sur l’axe Z.

1. **Accumulation des transformations : Matrice ModelViewProjection**

Cette matrice qui est le résultat de toutes les transformations est la résultante de la multiplication entre les matrices projection, vue et modèle.

Voici un schéma qui résume le mode de fonctionnement des différentes transformations :



Schéma

Pour utiliser les fonctions que nous aurons besoin afin de gérer c’est transformation on pourra faire appel à la librairie GLM (OpenGLMathematics).

# Conclusion

La faisabilité en 3D du jeu d’échec est tout à fait possible grâce à l’API OpenGL combiné à GLM et SDL. OpenGL a en plus l’avantage d’être multiplateforme à l’instar de DirectX. Bien que la modélisation 3D nous attire beaucoup et semble vraiment intéressant, étant néophyte dans la modélisation que ça soit 3D ou 2D nous préférons partir sur un jeu 2D plutôt qu’en 3D. De plus nous préférons concentrer nos ressources sur l’IA qui représente une grosse partie du travail à accomplir. Mais rien ne nous empêchera à l’avenir de revenir sur la 3D s’il nous reste du temps et surtout si la partie 2D a bien était assimilé au préalable.

Si le choix du 2D est validé, nous nous tournerons vers l’API SDL qui est une API graphique fait spécialement pour les jeux en 2D, de plus on en a besoin pour crée un context dans lequel OpenGL devra s’exécuter donc c’est nouvelles connaissances seront réutilisables dans le futur pour passer un cran au-dessus, c’est-à-dire à la 3D.

Keyvan Melaini | Julien Hautier | Jean-Charles Gravaillac

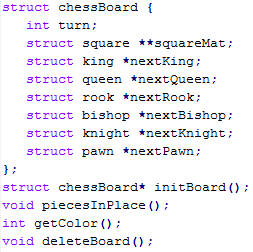
Structures and fields

definition

# Introduction

This will entertain you about all the structures we will use, what are their fields and associated functions, and what are they aimed for.

# The chessboard

This is our Chessboard Structure.

First, please consider that we could add here any “rule” field to set game’s options such as the “en passant” rule. They would probably be simple “integer” field (0 or 1).

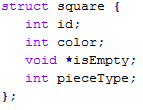
SquareMat is basically the 8X8 matrix that is filled with squares, which are also one of our struct that you will see just after.

The “turn” parameter tells us which color should play on each turn, also a simple “integer”.

Then we have all our lists, which are pointers to linked lists of structures for each type of piece.

We have an initBoard() function that will create our board and pieces, the piecesInPlace() function that moves all the pieces where they should be at the beginning of the game, a deleteBoard() function that will “free” our board at the end of the game, and a getColor() function that tells us the color of a piece on a distinct square.

That is all for the moment.

The square structure that fills our board is very simple. It contains its “id” which is the square place (for example a simple integer “25”, with a 1st number that would represent the row and the 2nd for the column).

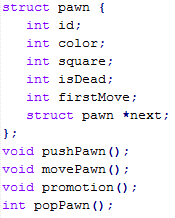
The color tells us if we are on a black or a white square, again an integer.

The “isEmpty” parameter tells us if the square is empty or not, we will have a pointer to NULL on it if it is empty, and on the piece struct otherwise, which is why we use a void type.

Finally, we have an integer “pieceType” that we’ll use to know which structure to cast for the “isEmpty” void pointer.

# The pieces

### The pawn

Our pawn structure is unique and will contain an “integer” id (each different pawn will have its own id).

Another integer will define the color of the piece.

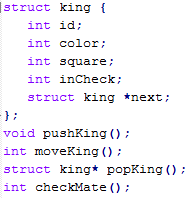
Then we will have the integer id of the struct square on which the piece is placed.

The firstMove integer define whether the Pawn can move 2 squares forward or not (in case of its 1st move).

Finally, “next” is a pointer to the next pawn, since we use linked lists.

The pawn also has the possibility to promote himself during the game (the promotion rule), which allows him to be replaced by any type of piece. This is what the promote() function is aimed for.

### The king

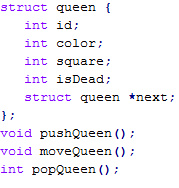
 The king structure is also special, as we need to manage the “inCheck” statement.

Thus, the king will have a special “inCheck” field that will tell us if the king piece is in check or not.

We will also have an additional function checkmate(), that we will use when the king is “inCheck” to know if this is the end of the game.

### The other pieces

All the other structures are mostly the same with different names, as the following:

The isDead argument is common to all pieces except the king. It tells us if the piece has been taken or not.

We already described all the parameters before.

The “push” and “pop” functions are basically creating and deleting function for linked lists.

The “move” function that we use for all our pieces, is basically not a move function but will check whether the move that the player wants to make is possible or not. Considering the integer that the function returns, we will proceed on moving the piece.

Chronologie (groupe/individuel) et répartition des tâches :

