

# C++ & OpenGL

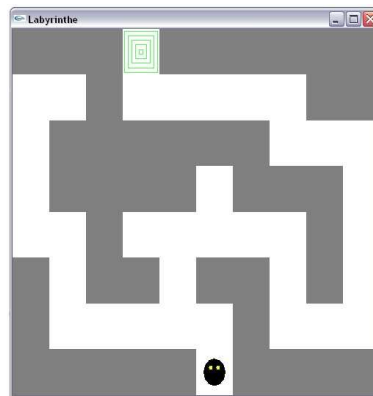
## Pratique de base – Mini projet Pacman

### Exercice 14 : revoir un peu tout

Le but de ce mini-projet est de réaliser un petit-jeu permettant de déplacer un avatar dans un labyrinthe, en essayant d'atteindre la sortie du labyrinthe sans croiser le chemin d'ennemis se déplaçant aléatoirement. Le jeu sera vu de dessus, en 2D. L'avatar sera dirigé par l'utilisateur grâce au clavier.

Dans le cas idéal, nous souhaiterions que notre jeu fonctionne avec des niveaux de taille variable, que l'utilisateur pourrait lui-même concevoir et utiliser dans son jeu. Nous réaliserons cet aspect du cahier des charges quand nous aurons déjà testé une application fonctionnelle avec un niveau de taille fixe codé « en dur ».

Voilà à quoi pourrait ressembler notre petit logiciel :



### Etape 1 : créer le projet, ajouter ce qu'il faut où il faut

Créez un nouveau projet vide de type *Application Console Win32*. Ajoutez un fichier source vierge, ajoutez-y la fonction `main` (vide pour le moment) et compilez votre projet. Cette manipulation a seulement pour but d'obliger Visual à créer le répertoire *Debug* : nous pourrions ainsi copier le fichier `glut32.dll` dans ce dernier (voir la suite).

Nous allons utiliser GLUT : les fichiers nécessaires à son utilisation ne sont pas présents nativement, il va falloir les ajouter à votre projet.

1. Récupérez le fichier zip cette adresse : <http://www.xmission.com/~nate/glut/glut-3.7.6-bin.zip>
2. Allez dans le répertoire qui contient le fichier `.cpp` de votre projet. Créez un répertoire nommé `GL` et un répertoire nommé `LIB`. Dans le premier, copiez le fichier `glut.h`, et dans le second le fichier `glut32.lib`.
3. Ajoutez cette ligne au début de votre programme : `#include "GL/glut.h"`
4. Faites un clic droit sur le nom de votre projet (dans Visual C++), sélectionnez « Propriétés », puis « Propriétés de configuration », puis « Editeur de liens », puis « Entrée ». A la ligne « Dépendances supplémentaires », écrivez « `LIB/glut32.lib` ».
5. Placez le fichier `glut32.dll` dans le même répertoire que l'exécutable de votre projet, à savoir « ...\\nom\_de\_votre\_projet\\debug ».

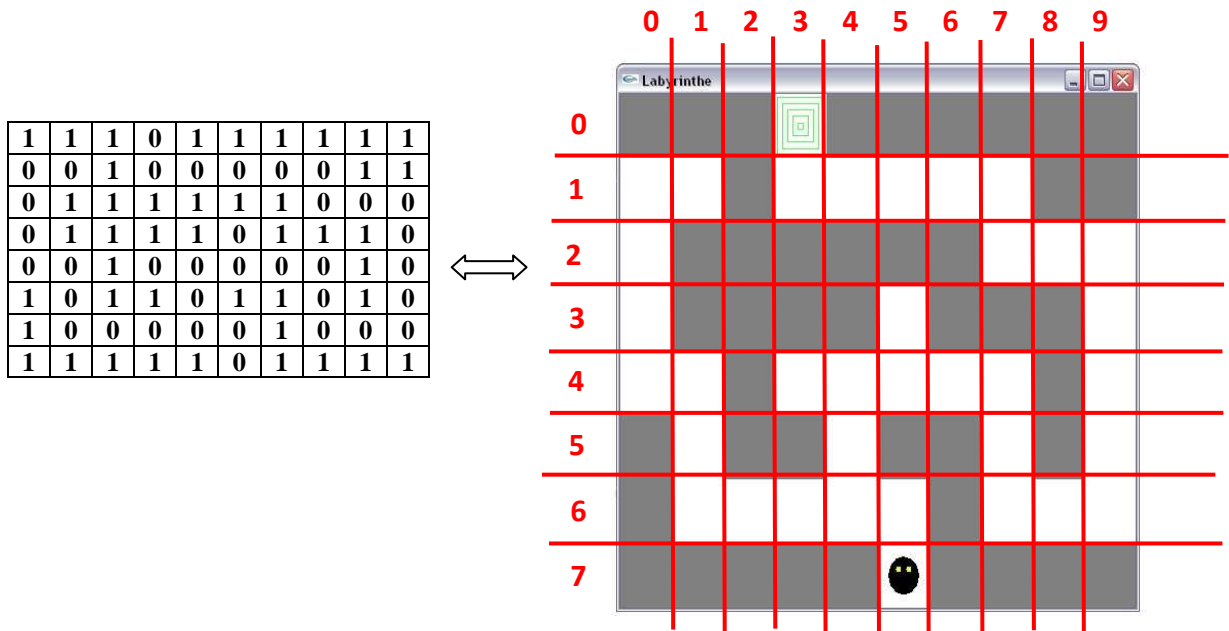
### Etape 2 : le niveau

Voilà, ce qui est nécessaire à l'utilisation d'OpenGL est au bon endroit, maintenant nous pouvons nous consacrer à la réalisation de notre petit jeu. L'idée générale est de savoir comment décrire un niveau, c'est-à-dire comment de dire de manière assez formelle où sont les murs et où se trouvent notre avatar et les ennemis.

De manière relativement intuitive, nous pouvons voir notre plateau de jeu comme une sorte de grille, dans laquelle une case de la grille peut être un mur ou un passage, et si elle contient un ennemi ou non. Un peu comme pour la bataille navale, il devient très simple de dire : *pacman* se trouve dans la case située à la ligne *i* et à la colonne *j*, et il ne peut se déplacer à droite par exemple, car la case (*i*+1, *j*) est un mur. En conséquence, quelle soit la physionomie du niveau, nous pouvons le décrire comme un tableau à 2 dimensions, qui a un nombre de lignes et un nombre de colonnes.

Si j'arrive à faire le lien entre un tableau qui représente le niveau dans mon programme et le niveau proprement dit (celui qui s'affiche à l'écran), il me sera très simple de dessiner n'importe quel type de niveau. Nous n'allons pas « dessiner » un niveau au gré du vent sous OpenGL, nous allons que retranscrire un schéma défini par ailleurs.

Par exemple, définissons au début du programme un tableau à 2 dimensions contenant des 1 et des 0 : le 1 symbolise un mur, le 0 un passage. Pour générer le niveau sous OpenGL, je n'aurai plus qu'à parcourir mon tableau case par case, regarder la valeur et dessiner un quadrilatère si c'est 1, rien sinon :



L'idée sous-jacente est d'avoir une méthode de construction du niveau en 2D totalement indépendante de la « physionomie » du niveau.

Pour tracer le niveau de jeu à partir de la matrice de caractères, on pourrait envisager plusieurs solutions. On pourrait tracer des lignes entourant des blocs des murs, ou tracer des lignes délimitant les allées. La plus simple, et qui est particulièrement adaptée à notre tableau, est de remplir les cellules « occupées » par des murs en dessinant un carré de couleur à leur position : la primitive géométrique adaptée est GL\_QUADS. La question est de savoir comment, étant données les coordonnées x et y d'une position dans le tableau, déduire les coordonnées des 4 points pour générer ma primitive.

→ Les vertices nécessaires pour tracer le carré correspondant à la cellule d'indice [i][j] ont pour coordonnées (i, j) (i, j+1) (i+1, j+1) (i+1, j). Faites vous un petit schéma pour vous aider à comprendre.

#### ⇒ A vous !

Je vous donne le squelette de la classe, à vous de le compléter en finissant d'écrire la matrice et en écrivant le corps de la méthode « DessinerNiveau » (c'est elle qui fait la traduction entre tableau et dessin des quadrilatères). Pour vous aider, je vous donne, p.3, le squelette du fichier source.

```

#include "GL/glut.h"

int NbLignes=3 ;
int NbColonnes=3 ;

char Matrice[3][3] = {'1','1','1','1','0','1','1','0','0'} ;

// Déclarations de fonctions
void LabyAffichage();
void LabyRedim(int x, int y);
void DessinerNiveau();

void main(void)
{
    // Gestion de la fenêtre
    glutInitWindowPosition(10, 10);
    glutInitWindowSize(500, 500);
    glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
    glutCreateWindow("Labyrinthe");

    // Gestion des événements
    glutDisplayFunc(LabyAffichage);
    glutReshapeFunc(LabyRedim);

    glutMainLoop();
}

// Événement d'affichage
void LabyAffichage()
{
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    DessinerNiveau(); // Affiche le niveau
    glFlush();
}

// Dessin du décor
void DessinerNiveau()
{
    // mettre l'affichage des murs ici
}

// Événement de redimensionnement
void LabyRedim(int x, int y)
{
    glViewport(0, 0, x, y);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (double)NbColonnes, (double)NbLignes, 0.0);
}

```

Prévoyez aussi une case de couleur spéciale qui sera la sortie de notre labyrinthe (ce n'est pas un vrai *pacman*). L'idéal serait que vous spécifiez 2 variables globales de plus donnant la position de la sortie. Le plus simple serait de rajouter un symbole différent de '0' et de '1' dans votre tableau initial, par exemple '2', pour que passiez un quadrilatère de couleur spéciale quand vous arrivez à cette case.

Tant qu'à faire, ajoutez-en encore 2 variables globales pour mémoriser la position de départ, nous nous en servons par la suite.

### Etape 3 : l'avatar

Le niveau est maintenant défini et affiché à l'écran. Il nous reste encore à ajouter notre avatar, puis à lui permettre de se déplacer comme l'utilisateur le souhaite. Nous allons pour cela définir une nouvelle classe, spécifique au *pacman*. Ajouter une classe n'est pas forcément nécessaire, mais cela nous fait réviser un peu...

La question qui se pose est qu'est-ce qui caractérise notre avatar, et quelles sont les fonctionnalités que nous aimerions qu'il possède. Il faut bien sûr restreindre notre réflexion au contexte du jeu. A priori, ce qui nous intéresse par rapport au jeu, c'est de connaître sa position. Les fonctionnalités sont de pouvoir se déplacer dans les 4 directions, a priori sans jouer au passe-muraille et sans sortir du niveau du jeu. Et bien entendu, il reste le problème de la

physionomie de notre avatar. Il pourrait sembler simple de lui donner sa physionomie dans le constructeur, en même temps que l'on va initialiser sa position de départ. Pour autant, cela n'est pas convenable : en effet, nous allons redessiner la scène et tout ce qu'elle contient à chaque appui sur une touche (si le déplacement est possible). Nous aurons donc besoin de redessiner *pacman*, qui aura toujours la même allure mais à une autre position. Si on met le dessin dans le constructeur, cela veut dire qu'à chaque déplacement on va créer une nouvelle occurrence de *pacman*. Nous allons faire une fonction de plus, qui permettra de dessiner *pacman* à une position paramétrable.

#### ⇒ A vous !

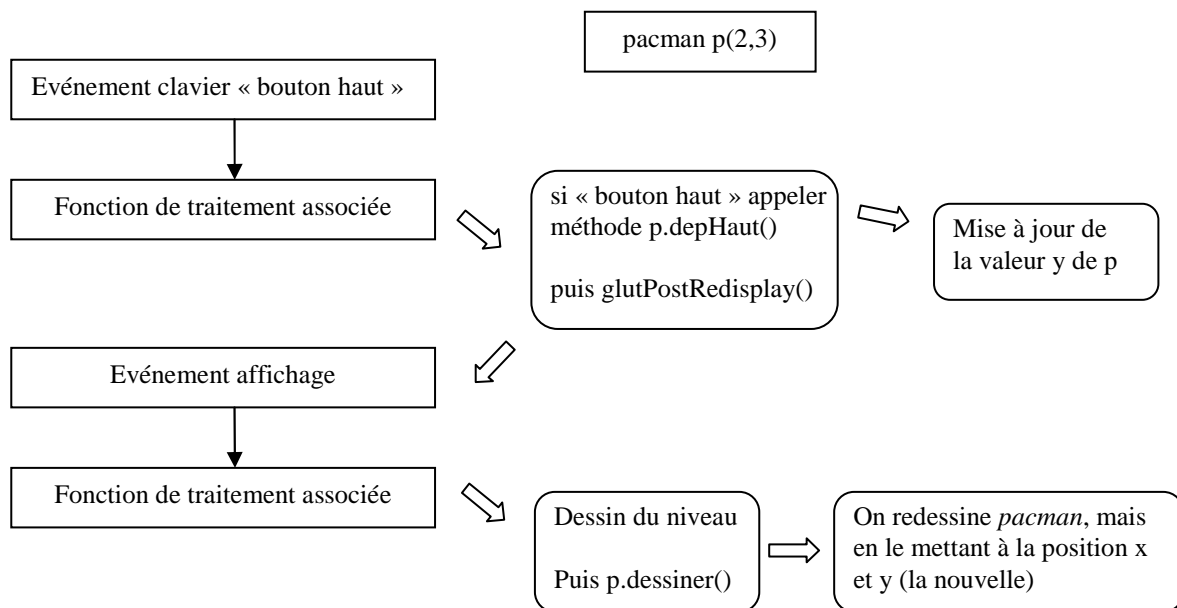
Créez une nouvelle classe (un nouveau fichier .cpp et un nouveau fichier .h) du nom que vous voulez (*pacman*, *avatar*, ...). Le *pacman* est donc défini par ses coordonnées *x* et *y*. Il possède un constructeur, permettant d'initialiser sa première position dans le labyrinthe. Il possède au moins 5 fonctions, qui sont :

- void *depGauche()* : décrémente de 1 la position en *x*
- void *depDroit()*, void *depHaut()*, void *depBas()*
- void *dessiner()* : permet de donner une forme à l'avatar, tout en le positionnant aux coordonnées *x* et *y*

Complétez le fichier .h

Avant de s'occuper de compléter le fichier .cpp, essayons de comprendre un peu le fonctionnement du jeu.

Au niveau du *main()*, et plus précisément de la boucle principale d'OpenGL, on attend des événements : redimensionnement, affichage et, dans notre cas, événement clavier. A chaque fois que l'utilisateur va appuyer sur une touche pour déplacer le *pacman*, l'événement sera récupéré, et la fonction que vous avez définie sera exécutée. Vous aurez bien sûr à faire une disjonction de cas en fonction de la touche. Si c'est la touche « haut », il vous suffira d'appeler la fonction *depHaut()* de la classe *pacman* : celle-ci, après avoir vérifié que le déplacement était possible, modifiera les coordonnées du *pacman*. A la fin de votre fonction événement clavier, vous demanderez alors explicitement le rafraîchissement de l'affichage. Cela aura pour effet d'appeler votre méthode d'affichage, dans laquelle vous appellerez la fonction *dessiner()* : le *pacman* sera redessiné, mais avec la bonne position.



Il faut donc que dans la fonction *dessiner*, vous déplacez le *pacman* à la position *x* et *y*. Comme ça, à chaque appel de *dessiner*, *pacman* est positionné à sa position courante. Pour la physionomie de notre avatar, contentez-vous pour le moment de le dessiner sous forme d'une sphère jaune, avec 2 petites sphères noires pour les yeux. En vue de « dessus », vous aurez l'impression de voir des cercles.

#### ⇒ A vous !

Commencer par compléter le fichier .cpp : d'abord, le constructeur, puis les fonctions de déplacement. Dans ces dernières, mettez à jour les valeurs de *x* et de *y* (+1, -1 ou rien en fonction du déplacement). Enfin, complétez la fonction de dessin :

```

void Joueur::Dessiner()
{
    glPushMatrix();
    // mettre le bon déplacement

```

```
// dessiner le pacman
glPopMatrix();
}
```

Pour faire une sphère, vous pouvez utiliser la fonction :

```
void glutSolidSphere (GLdouble radius , GLint slices , GLint stacks ) ;
slices et stacks vous permettent de donner le nombre de segments en latitude et longitude
```

Il vous reste à gérer le recueil des événements clavier avec `glutSpecialFunc(nom_de_votre_methode)` et de faire le traitement adéquat dans votre méthode suivant la touche appuyée. Voici un exemple pour la touche « haut » :

```
void traitementClavier(int key, int x, int y)
{
    if (key == GLUT_KEY_UP) p.depHaut();
}
```

#### ⇒ A vous !

Créez un nouvel objet de type `pacman`, associer à l'événement clavier votre fonction de traitement, et complétez votre fonction de traitement pour les 4 touches directionnelles. N'oubliez pas d'inclure l'entête de votre classe `pacman`.

Testez votre jeu...

Effectivement, *pacman* joue au passe-muraille, et ne s'occupe pas des murs ou des bornes de notre niveau. Car dans les fonctions `depXXX()` de la classe `pacman`, nous avons modifié les coordonnées, sans vérifier si les déplacements étaient possibles. L'amélioration est assez facile à trouver : avant de mettre à jour `x` et/ou `y`, il faut vérifier si le déplacement projeté ne correspond pas à un mur ou une fin de niveau. Il suffit pour cela de s'appuyer sur notre tableau de niveau.

Le problème est que ce dernier est défini dans la classe contenant la méthode `main()`, et qu'il n'est pas accessible dans la classe `pacman`. Heureusement, il y a une solution, il suffit de redéclarer dans le fichier `pacman.cpp` le tableau comme une variable globale, en ajoutant le mot clé `extern` :

```
extern int NbColonnes ;
extern int NbLignes ;
extern char** Matrice ;
```

#### ⇒ A vous !

Gérez correctement le déplacement et testez votre jeu

Il ne reste plus que 2 grandes choses à faire : d'une part, vérifiez à chaque événement clavier si le joueur a gagné, c'est-à-dire s'il est arrivé dans la case de sortie. Je vous laisse le soin de gérer cela. Enfin, il faut ajouter des ennemis, sinon cela est trop facile.

L'idée, pour l'ennemi, est de se déplacer de manière aléatoire, régulièrement mais indépendamment des déplacements de notre avatar. Nous allons pour cela utiliser un timer, c'est-à-dire une temporisation. Il suffit d'ajouter le traitement d'un événement qui se produit à chaque fin du timer, et une fois que le traitement est fait (déplacer l'ennemi), réarmer le timer.

Dans votre fichier contenant la fonction `main()`, ajoutez `#include <time.h>`

Puis la gestion de l'événement : `glutTimerFunc(500, LabyTimer, 0);` Ici, *LabyTimer* correspond à la fonction associée à l'événement. A la fin de fonction de traitement (modifier de manière aléatoire la position de l'ennemi), réarmer le timer : `glutTimerFunc(500, LabyTimer, 0);`

#### ⇒ A vous !

Faites une nouvelle classe ennemi (elle ressemble beaucoup à la classe `pacman`, à part pour la forme de l'ennemi) et gérez son déplacement comme cela a été défini au dessus.

Rajoutez la gestion des collisions : si `pacman` et ennemi à la même position, l'utilisateur a perdu

Vous pouvez bien entendu améliorer, rajoutez des ennemis, ou améliorer le rendu, à votre convenance.