Programmation OpenGL

Samuel Jacquet

Cahier des charges

Introduction

Description de la demande

Les objectifs
Faire un jeu 2D ressemblant à Jumping Banana, codé avec la bibliothèque OpenGL écrite en C.

Produit du projet
Il y aura un fichier exécutable .exe pour lancer le jeu et les fichiers dont le jeu dépend.

Les fonctions du produit

- Plusieurs cartes.
- Gestion des collisions entre le personnage et l'environnement.
- Gestion de la musique.
- On pourra se déplacer vers la gauche la droite et sauter.

Manuel d'utilisation :

Commandes

Voici la liste des touches utilisables :

Se déplacer vers la gauche	Flèche directionnel gauche
Se déplacer vers la droite	Flèche directionnel droite
Sauter	Flèche directionnel haut
Retourner au menu	Escape

Partie Technique

Introduction

Le jeu fonctionne de la manière suivante :
On commence par lancer le fichier main.c qui va lancer la musique, et afficher l'image du menu.

Les structures

Map
La structure Map, définie dans le fichier map.h, contient toutes les informations relatives à la carte.

- Le nombre de bloc en largeur et hauteur (ce qui permet de donner les limites de la carte) qui ont été définis dans le fichier .lvl qui contient le niveau.
- Un tableau à plusieurs dimensions, où est stockée la carte.
- Dans le tableau « used », à la même position qu'où a été définis le bloc dans le tableau « matrice », je mets la valeur à 1 correspondant à l'indice de texture pour la banane. Ensuite

dans les collisions, si on touche cette case, on modifie le score et l'indice de score vers néant. S'il ne vaut pas 1 il n'est pas utilisé.

```
typedef struct str_map
{
    int Nb_Block_W, Nb_Block_H;
    GLuint16** LoadedMap;
GLuint16** Used;
    int xscroll,yscroll;
};
```

Sprites et collisions

La structure Sprites contient :

- Une variable GLuint qui sert à contenir une image. Elle contiendra donc l'image de chaque sprites.
- La deuxième variable dit si on peut traverser le sprite ou pas.

```
struct str_texMap
{
    GLuint* texID;
    int isGoingThrough;
};
```

Les Fonctions

Main (main.c)

On définit la taille de l'écran et la position de l'image du menu. On fait de même pour le second écran qui sert juste à afficher des données pour le développer.
On lance la fonction audio_init qui va charger les musiques et le son et on lance la fonction audio_play avec comme paramètre « 1 » pour qu'il lance la musique en boucle. Ces fonctions seront décrites plus loin.
Ensuite on attend simplement qu'on défile une option (fonction game, rules, score, exit).

Audio_init (audio.c)

J'ai défini en variable global les variables qui contiendront les sons pour qu'elles soient accessibles facilement sur chaque fonction, plutôt que de les passer en paramètre.

J'ai initialisé FMOD et assigné les sons aux variables pour qu'on n'ait plus qu'à « play » la variable.

```
if (song==1)
    FMOD_System_PlaySound(sys, 0, music, 0, NULL); //Joue la musique
if (song==2)
    FMOD_System_PlaySound(sys, FMOD_CHANNEL_FREE, jump, 0, NULL);
if (song==3)
    FMOD_System_PlaySound(sys, FMOD_CHANNEL_FREE, lose, 0, NULL);
if (song==4)
    FMOD_System_PlaySound(sys, FMOD_CHANNEL_FREE, win, 0, NULL);
if (song==5)
    FMOD_System_PlaySound(sys, FMOD_CHANNEL_FREE, shroom, 0, NULL);
```

Note : Audio_stop (audio.c)

J'ai simplement utilisé les fonctions FMOD de libération des variables FMOD.

Game (game.c)

La fonction game est l'une des deux qui peuvent être appelée par le main. C'est la fonction de base du fonctionnement du jeu. Celle qui va lancer toutes les fonctions de chargements d'images et autres.

```
1.  void InitGame(void)
2.  {
3.      texID=loadTex(); // load texture.
4.      char *level = "../../data/matrice/niveau1.lvl"; // new map.
5.      m=loadMap(level); //load the map.
6.      L=newlist();
7.
8.      initBanana(m);
9.
10.     g=createGame("Monkey"); // new game.
11.     c=createCharacter(5);
12.     //o=createObject();
13.     e=createEnnemy();
14. …
```

D'abord on définit toutes les variables les plus importantes pour le jeu :

```
1.  GLuint window, window2;
2.  int v=1;
3.
4.  typ_state state=MENU; // to init game state.
5.  typ_state selectedMenu;
6.  typ_action action=RIGHT; // to init game menu.
7.
8.  typ_data *L=NULL; // list to manage Game Data.
9.  typ_map *m=NULL;
10. typ_game *g=NULL;
11. typ_character *c=NULL;
12. typ_character *e=NULL;
13. typ_object *o=NULL;
14.
15. GLboolean lauded=0;
16. GLboolean cleaned=0;
17. ------------------------------------------------
```

Une fois que la personne quitte la partie, on sort du menu game. On libère les images et les structures et on retourne dans le menu principal.

```
1.  void freeAll(void)
2.  {
3.      // delete pointer.
4.      freeCharacter(c);
5.      freeGame(g);
6.      // delete textures.
7.      if (!cleaned){
8.          freeTextureGame(texID);
9.                freeMap(m);
10.     }
11.        cleaned=GL_TRUE;  // Want to make sure we only delete it once.
12.     // delete audio system.
13.
14.     glutDestroyWindow(window);
```

```
        glutDestroyWindow(window2);
}
```

loadTex (map.c)
Cette fonction charge l'image de chaque sprite avec la fonction ImageLoad et l'assigne à un chiffre qui sera celui utilisé dans le fichier qui contient le niveau.

Et finalement je définis pour chaque sprite si on peut le traverser ou non. Ce qui est nécessaire pour la gestion des collisions.

```
GLuint *texID=NULL;
```

```
GLuint *loadTex(void)
{
    GLuint* texID=NULL;
    texID=malloc(texNB*sizeof(GLuint));
    if(texID==NULL){
        fprintf(stderr, "Error : dynamic allocation problem.\n");
       exit(EXIT_FAILURE);
    }
    texID[0]=ImageLoad(MONKEY1);
    texID[1]=ImageLoad(MONKEY2);
    texID[2]=ImageLoad(MONKEY3);
    texID[3]=ImageLoad(MONKEY4);
    texID[4]=ImageLoad(MONKEY5);
    texID[5]=ImageLoad(MONKEY6);
    texID[6]=ImageLoad(MONKEY7);
    texID[7]=ImageLoad(MONKEY8);
    texID[8]=ImageLoad(MONKEY9);
    texID[9]=ImageLoad(MONKEY10);

    texID[10]=ImageLoad(SNAKE1);
    texID[11]=ImageLoad(SNAKE2);
    texID[12]=ImageLoad(SNAKE3);
    texID[13]=ImageLoad(BACKGROUND);

    texID[14]=ImageLoad(COIN1);
    texID[15]=ImageLoad(COIN2);
    texID[16]=ImageLoad(COIN3);
    texID[17]=ImageLoad(COIN4);

    return texID;
}
```

loadMap (map.c)
Cette fonction lit le fichier qui contient la location des blocs du niveau.

Le fichier contient d'abord le nom du niveau (sans espaces) puis un retour à la ligne, ensuite le nombre de blocs de la largeur du niveau et un espace et le nombre de blocs pour la hauteur du niveau puis retour à la ligne.

Ensuite on met le numéro correspondant à notre bloc (définis dans `ImageLoad(`*`Indice de l'image`*`)`), un espace et le numéro du bloc suivant, ainsi de suite jusqu'à ce qu'on ait mis le nombre de bloc en largeur, on fait un retour à la ligne et on écrit la ligne suivante. Il ne faut surtout pas oublier/rajouter d'espaces !

La nombre de blocs pour la hauteur (variable height) et la largeur (variable width) sont enregistré dans leur champ respectif de la structure Map. La carte est enregistrée dans le tableau à deux dimensions « matrice », aussi dans la structure Map.

```
typ_map* loadMap(char *level)
{
    typ_map* m;
       FILE* F;
       F=fopen(level,"r");
    if(F==NULL){
        fprintf(stderr,"Error : No such file or directory : %s\n",level);
        fprintf(stderr, "Error : %s\n",strerror(errno)); /* On affiche sur le canal d'erreur standard, "stderr". */
        perror("Error ");
        exit(EXIT_FAILURE); /* program left. */
    }

    int i,j;
    int w,h;
    int tmp;
    char *bufferName=malloc(sizeof(char)*50);
    if(bufferName==NULL){
        perror("Error ");
       exit(EXIT_FAILURE);
    }

    fscanf(F,"%s",bufferName); // level name.
       fscanf(F,"%d %d",&w,&h); // number of square on the width/height.

    m=createMap(w,h);
       m->xscroll=0;
       m->yscroll=0;

    for(i=0;i<m->height;i++)
    {
        for(j=0;j<m->width;j++)
        {
            fscanf(F,"%d ",&tmp);
            m->matrice[j][i]=tmp;
            m->used[j][i]=0;
        }
    }
       fclose(F);
       return m;
}
```

drawMap(map.c)

Cette fonction va afficher la carte, mais seulement où se trouve le personnage. Car on ne veut pas utiliser des ressources en plus pour afficher la carte qui se situe en dehors de la fenêtre.

Il faut commencer par prendre la coordonnée x de la carte par rapport au personnage (on a une variable pour ça, xscroll et yscroll) et on le divise par la taille d'un sprite dans le but de connaître le combientième sprite est disponible en partant de la gauche. On fait la même chose avec la coordonnée y de la carte et avec les 2 résultats, on a la coordonnée x et y du tableau qui contient la carte. Donc on a plus qu'à lui dire d'afficher aux coordonnée x et y de notre carte, l'image qui correspond au numéro stocké dans le tableau.

Deux variables globales sont stockées dans map.h, `TILE_NUMBER` et `Square_size`.

```
#define MaxX 40
#define MaxY 40
```

```
3.  #define Square_size 10.0
4.  #define WIDTH Square_size*MaxX
5.  #define HEIGHT Square_size*MaxY
6.  #define WIDTH2 150
7.  #define HEIGHT2 150
8.
9.  #define BORNE_INFERIEUR 1
10. #define BORNE_SUPERIEUR 10
11. #define bananaNB 10
12.
13. #define MAP1 "../../data/image/JB_96.png"
14. #define MAP2 "../../data/image/JB_96.png"
15. #define MAP3 "../../data/image/JB_96.png"
16. #define MAP4 "../../data/image/block1.png"
17. #define BANANA "../../data/image/JB_87.png"
18.
19. #define TILE_NUMBER 4
```

Je les ai définies globalement car elles doivent être accessible facilement, et modifiées facilement aussi car le nombre de sprite ou la taille d'un sprite peut changer. `TILE_NUMBER` sert à l'allocation mémoire de la structure qui contient toutes mes sprites. Comme ça si j'ajoute un Sprite, je modifie juste ce nombre et il sera automatiquement modifié quand je fais le malloc de la structure Sprites. Il faut aussi définir quelque part. Je n'ai pas beaucoup de textures donc je les ai définies aussi globalement.

`Square_size` est souvent utilisé pour des calculs sur la carte. Par exemple pour savoir le combientième bloc c'est depuis la gauche en connaissant sa distance depuis le bord de la carte ou pour savoir chaque combien de pixel il faut afficher un bloc sur la carte.

Pour afficher sur toute la fenêtre il suffit de mettre ce code dans deux for, qui vont de minimum x à maximum x de la fenêtre et même chose pour y. Et on incrémente de sorte qu'on passe d'un sprite à un autre.

`WIDTH` et `HEIGHT` sont deux autres variables globales définient dans game.h, elles servent à définir la taille de la fenêtre. Elles sont aussi définies en variable global pour pouvoir les modifier facilement.

```
1.  // fonction qui affiche les murs et les plateformes
2.  void drawMap(typ_map *m)
3.  {
4.          int i,j;
5.          int minX, maxX, minY, maxY;
6.          int tileNumber=0;
7.          minX=m->xscroll/Square_size-1;
8.          minY=m->yscroll/Square_size-1;
9.      maxX=(m->xscroll+WIDTH)/Square_size;
10.     maxY=(m->yscroll+HEIGHT)/Square_size;
11.
12.         for(i=minX; i<=maxX; i++)
13.         {
14.                 for(j=minY; j<=maxY; j++)
15.                 {
16.                         if (i<0 || i>=m->width || j<0 || j>=m->height)
17.                         {
18.
19.                         }
20.                         else
21.                         {
22.                                 glPushMatrix();
```

```
                                    glTranslatef(i*Square_size-m->xscroll,
    j*Square_size-m->yscroll,0.0f);
                                    tileNumber=m->matrice[i][j];
                                    glBindTexture(GL_TEXTURE_2D, m-
    >tiles[tileNumber].texID); // bind our texture.
                                    if(tileNumber)
                                           drawRect(0,0,Square_size,Square_size);
                                    glBindTexture(GL_TEXTURE_2D, 0);
                                glPopMatrix();
                        }
                }
        }
}
```

freeMap(map.c)

Après avoir chargé les images les structures en mémoire, il faut les supprimer :

```
void freeMap(typ_map *m)
{

        if(m->tiles!=NULL){
               glDeleteTextures(1,&m->tiles[0].texID);  // Delete The Shader Texture.
               glDeleteTextures(1,&m->tiles[1].texID);
               glDeleteTextures(1,&m->tiles[2].texID);
               glDeleteTextures(1,&m->tiles[3].texID);
               glDeleteTextures(1,&m->tiles[4].texID);

               free(m->tiles);
        }
        if(m!=NULL){
         for(int i = 0; i < m->width; i++){
             free(m->matrice[i]);
             free(m->used[i]);
         }
         free(m->matrice);
         free(m->used);
         free(m);
    }
}
```

Character (game.c)

La plupart du temps, il suffit d'une position posX et posY, auquel on définis une taille sizeX et sizeY utile pour la gestion des collisions.

La variable texSelected est définie à 1. Cette variable correspond à l'image du personnage affiché, on l'utilise dans la fonction drawChar et est mis à jour dans updateRender (Loop->timer(30milisecondes)->Redisplay).

LibererChars (char.c)

J'ai simplement fait un « glDeleteTextures » pour chaque image du personnage, et un « free ».

```
void freeTextureGame(GLuint *texID)
{
    if(texID){
        glDeleteTextures(texNB, texID);
        texID=0;
    }
}
```

```
1.  void freeCharacter(typ_character *c)
2.  {
3.      if(c){
4.          free(c);
5.          c=0;
6.      }
7.  }
```

updateRender (char.c)
La fonction updateRender va afficher un signe en mouvement. Si on se déplace vers la droite il affiche les 3 images d'un singe en direction de la droite en alternant d'image chaque fois qu'on passe 30 fois dans la fonction (sachant qu'on passe une fois dans la fonction toutes les 4 millisecondes environ ça donne un bon effet de mouvement), même chose pour la gauche. Si on ne bouge pas on affiche l'image de base.

Le tout est mis dans deux if et passer dans une fonction timer :

```
1.  void updateRender(void)
2.  {
3.      if(c->textureDelay==5)
4.      {
5.          c->texSelected++;
6.          if(c->texSelected==10)
7.          {
8.              c->texSelected=2; // Texture indice for the monkey.
9.          }
10.         c->textureDelay=0;
11.     }
12.     c->textureDelay++;
13. }
```

updatePosition (Event.c)
C'est la première fonction de la série de fonction qui va gérer les collisions.

On prépare deux variables qui contiendront le mouvement qui a été fait, on ne l'applique pas encore au personnage justement parce que s'il y a une collision il faudra annuler le déplacement.

Donc on fait la première variable vx et la deuxième vy et si on se déplace vers la gauche, on fait vx moins le déplacement, si vers la droite on fait vx plus le déplacement.

Note : pour le changement de mes textures, j'utilise une variable action de type énumération qui varie suivant la touche pressée.

Si on saute on fait vy – le déplacement mais j'ai dû trouver un moyen pour qu'il redescende à un moment. J'ai donc fait une variable qui s'appelle v_grav, qui incrémente le vecteur y de 0,8 à chaque fois qu'on passe dans la fonction puis de redescendre lorsqu'il passe le niveau 0 et alors on ne peut plus monter. Donc la « gravité » le fera redescendre.

Ensuite on lance la fonction adjustment pour trouver le sol avec une meilleur précision.

updateScroll (Event.c)
Cette fonction sert à gérer la variable xscroll et yscroll qui définissent l'endroit de la fenêtre par rapport au joueur. Ca fait en sorte que le joueur soit toujours au milieu de l'écran, sauf si on est prêt du bord de la carte.

```
1.  void updateScroll(void)
2.  {
```

```
3.      // Mapscroll.
4.      m->xscroll = c->posChar->posX + c->posChar->sizeX/2 - WIDTH/2;
5.      m->yscroll = c->posChar->posY + c->posChar->sizeY/2 - HEIGHT/2;
6.
7.      if(m->xscroll<0)
8.          m->xscroll=0;
9.      if(m->yscroll<0)
10.         m->yscroll=0;
11. /*
12.     if(c->posChar->posX + c->posChar->sizeX/2 > WIDTH/2){
13.
14.         if(c->action==LEFT)
15.             test--;
16.         else if(c->action==RIGHT)
17.             test++;
18.         else
19.
20.         m->xscroll += test;
21.     }
22. */
23.        if(m->xscroll>m->width*Square_size-WIDTH-1)
24.         m->xscroll=m->width*Square_size-WIDTH-1;
25.        if(m->yscroll>m->height*Square_size-HEIGHT-1)
26.         m->yscroll=m->height*Square_size-HEIGHT-1;
27.
28. }
```

Bug

- L'écran secondaire fait parfois cracher le jeu de temps en temps.

Conclusion

Jumping Banana était un bon choix, c'est un projet faisable pour un débutant en programmation qui demande tout de même pas mal d'apprentissage et de travail. Ça donne de bonnes bases pour continuer dans les jeux-vidéos mais aussi dans le développement en général (manière de coder, approche, langage C...).

Ce projet m'a beaucoup plu, le développement m'intéresse beaucoup, plus encore dans les jeux vidéo. L'OpenGL est un langage intéressant bien qu'il ne laisse d'autre choix que d'utiliser les variables globales. Ce projet m'a motivé à continuer dans cette voie, je pense faire d'autres jeux et apprendre d'autres langages/librairies.