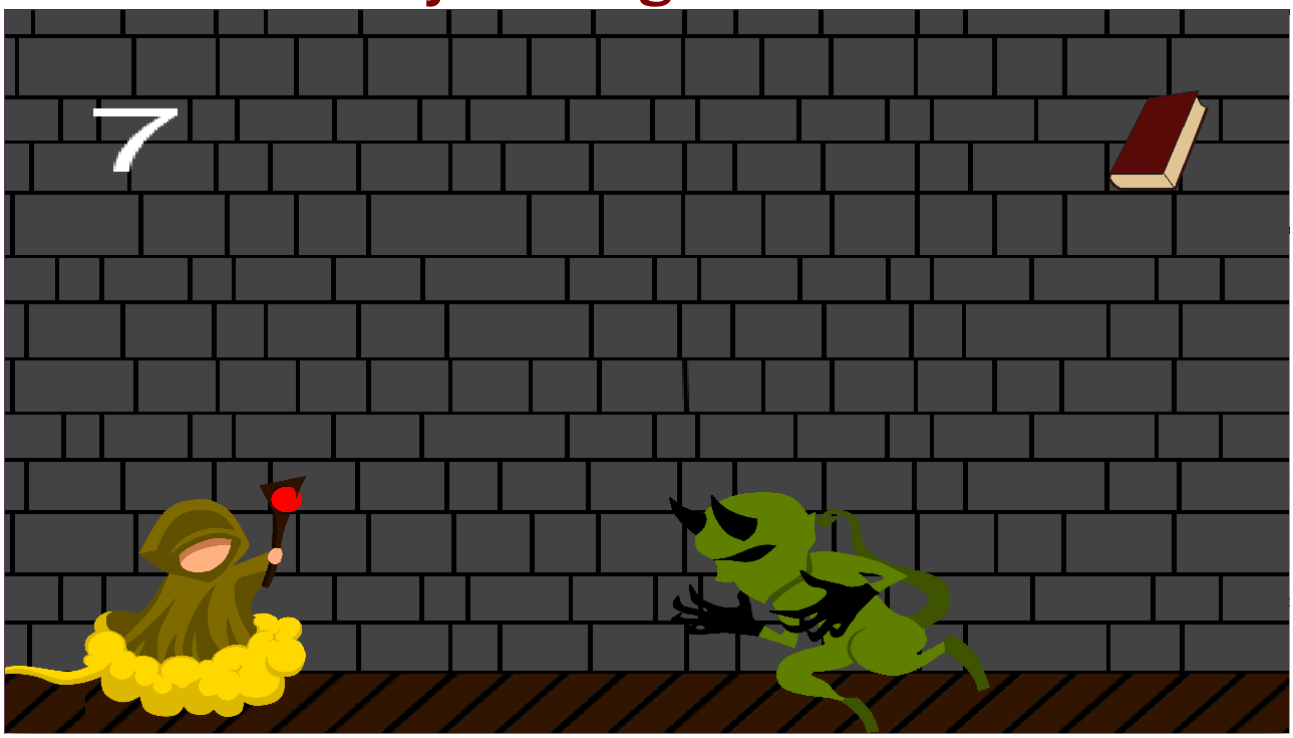
Projet L1 – CMI
Spécialité Informatique

Projet « Mage Runner »



Par :
PELERIN Aurélien, LAMOUCHI Ambre, AL-HAWAT Charbel
Soutenu le 27 Avril 2018

Sommaire

Introduction

Dans le cadre de la première année de Licence – Cursus Master en Ingénierie à la Faculté des Sciences de l'Université de Montpellier, se trouve l'UE HLSE205 – Projet CMI auxquels participent donc les élèves suivant des cursus en Informatique, Mathématiques et Électronique, Électrotechnique et Automatique.

Ce module d'enseignement a comme objectif de fournir aux élèves de l'apprentissage mais aussi de l'expérience dans la rédaction de rapports, la gestion de projets, mais surtout le travail en groupe. En effet, ces domaines sont utiles à tout élève aspirant à une position de chef de projet ou ingénieur dans sa carrière professionnelle.

L'UE requiert donc le dépôt d'un compte-rendu hebdomadaire sur le travail effectué par le groupe ainsi que la rédaction d'un rapport final, la production d'une vidéo de présentation et le support de soutenance prévue elle pour le Vendredi 27 Avril 2018.

En ce qui concerne notre groupe de travail (le G), sa formation ainsi que son départ sur le projet a été un peu particulière : en effet, au départ composé de seulement LAMOUCHI Ambre et de PELERIN Aurélien, qui s'étaient au départ lancé dans un autre projet qui fût annulé en raison de sa complexité, a vu ses rangs grossir en accueillant l'étudiant AL-HAWAT Charbel.

Nous nous sommes donc lancé dans un nouveau projet inspiré des jeux de types « endless runner » et plus particulièrement du jeu en flash Arcane Castle auquel avait joué PELERIN Aurélien bien plus jeune.

C'est ainsi que nous avons démarré notre projet subtilement intitulé « Mage Runner ». Mais quel est ce jeu ?

Cahier des charges

Pour se lancer dans le développement de notre projet, il fût primordial d'établir les règles ainsi que le fonctionnement de celui-ci.

Principe du jeu

Le principe du jeu est relativement simple, le joueur incarne un magicien se déplaçant de manière indéfinie vers la droite de l'écran et rencontre sur son passage des monstres qu'il doit détruire en utilisant le sortilège adéquat. Le but est donc pour le joueur de tenir le plus longtemps sans perdre la partie pour accumuler le plus de points possible.

Fonctionnement du jeu

Si nous venons de mettre au clair le principe du jeu, il est aussi important de mettre en lumière comment celui-ci doit fonctionner :

- Affichage d'un personnage sur le côté gauche de l'écran
- Déplacement d'un personnage de la droite vers la gauche de l'écran
- Déplacement d'un mur de fond vers la gauche de l'écran
- Mise à jour constante d'un jeu de variables (position ennemi, sort utilisé…)
- Comptage et affichage du score du joueur
- Redémarrage du jeu à la fin d'une partie

En ce qui concerne des fonctionnalités moins importantes on retrouve :

- Son lorsque le joueur lance un sortilège
- Affichage d'un effet pour indiquer le sort utilisé

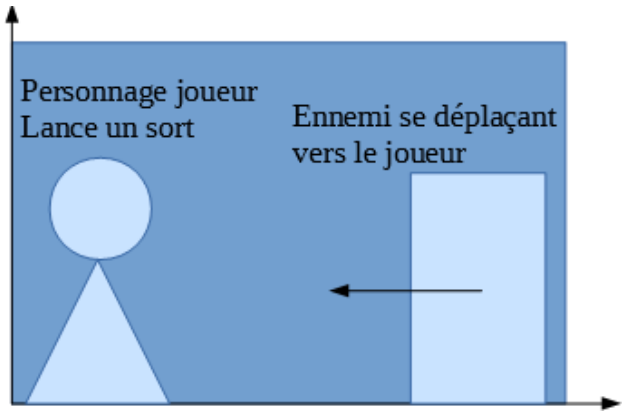


Figure 1 : Schéma du jeu n°1

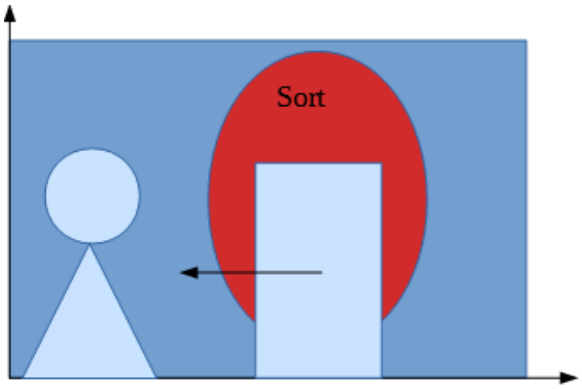


Figure 2 : Schéma du jeu n°2

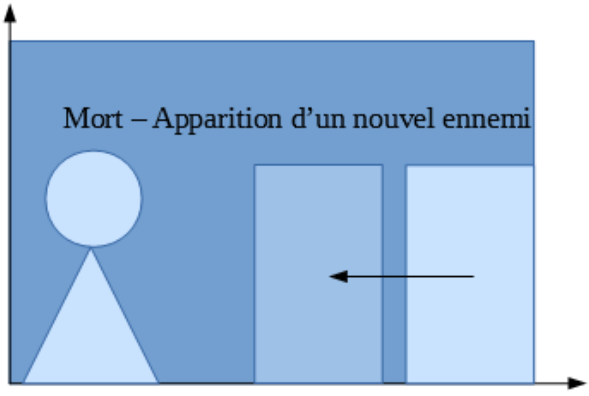


Figure 3 : Schéma du jeu n°3

Organisation du projet

Afin de mener à bien notre projet, nous avons dès le départ mis en place une organisation du travail à effectuer en répondant aux interrogations : Comment ? Pourquoi ? Qui ?

Organisation du travail

Le travail en groupe s'est révélé être une tâche particulière. En effet, chaque étudiant ayant ses compétences, sa manière de travailler et ses envies. Malgré cela, il a été aisé de se concerter pour donner un cap à notre projet et trouver notre concept de jeu donc. Le fait qu'un des membres du groupes ait déjà joué à un jeu de ce genre et que le projet en était grandement inspiré a permis à tout les membres du groupe de très facilement cerné le concept.

Suite au choix du concept du jeu, nous nous sommes penchés sur ce que nous devrions accomplir pour mener le projet à bien. L'organisation du travail a donc bénéficié de l'apport d'un diagramme de Gantt qui nous a apporté de nombreux intérêts dont :

- Le suivi de l'avancement du projet
- La mise en lumière des retards
- Permettre la mobilisation de ressources humaines sur les problèmes rencontrés

Élément	Responsable(s)	Date de début	Date de fin
Programmation			
Système de jeu	Charbel / Aurélien	16/03/18	22/03/18
Programmation des graphismes	Ambre / Aurélien	23/03/18	05/04/18
Menu	Charbel	23/03/18	30/03/18
Graphismes			
Sprite de magicien	Ambre	16/03/18	22/03/18
Sprites de démons	Ambre	16/03/18	22/03/18
Sprite de sorts	Aurélien	17/03/18	22/03/18
Autre			
Acquisition et implantation des effets sonores	Aurélien	05/04/18	07/04/18

Figure 4 : Diagramme de Gantt du projet

En vue de la quantité de travail nécessaire, les séances de TD ont surtout servis d'occasion d'échanger sur l'état du projet et de s'entraider sur son avancement tandis que la majeure partie de travail se faisait en hors-présentiel.

Choix des outils de développement

Si certains considèrent le jeu-vidéo comme un art, c'est qu'en sa conception se dégage quelque chose d'artisanal, et à tout bon artisan il faut de bons outils.

C'est donc pour ça que nous avons choisi un certain environnement de travail pour parvenir à réussir notre projet. Il s'agit là d'un langage de programmation, d'une librairie multimédia et logiciels. S'il peut y avoir des variations dans certains cas, notamment sur les logiciels employés, le langage de programmation ainsi que la bibliothèque multimédia utilisés doivent impérativement être les mêmes entre les étudiants participant au projet.

Pour éviter les conflits entre les systèmes d'exploitation, il a été convenu que l'écriture du code se fera sur un ordinateur Linux, sans employer ni Windows, ni Mac. Ceci a permis de ne pas avoir à adapter le programme sur chacun des OS, et par conséquent de gagner du temps sur la programmation.

Dès le commencent d'un projet de logiciel, l'une des premières interrogations fût celle du langage de programmation à utilisé. Il en existe un très grand nombre, qu'il s'agisse de la famille du C(C/C++/C#), du Java, du Javascript ou bien encore du Python etc. Le langage choisi se devait donc de répondre à deux critères indispensables :

- Être adapté à la création d'un jeu-vidéo
- Correspondre aux compétences en programmation des membres de l'équipe

Ces critères nous ont par conséquent contraints à nous tourner vers le C++. En effet, étant le sujet de notre UE HLIN202 – Programmation Impérative, l'intégralité de l'équipe se retrouvait familière avec ce langage. De plus, ses capacités de gestion de mémoire en font le meilleur langage pour créer des logiciels efficaces qui n'occasionnent pas de problèmes de performances à cause de fuites de mémoires.

Cependant, l'utilisation du C++ relevait en réalité plus du C. En effet, Charbel et Ambre n'étant pas à l'aise avec les concepts de la Programmation Orientée Objet, nous avons décidé de rester sur du C. Par ailleurs, les concepts de la POO ne se sont pas révélés indispensables ou leur absence trop handicapante. Concernant le choix de la librairie multimédia, il eu d'abord une hésitation entre la SFML et la SDL, mais la décision entre l'une ou l'autre fut rapidement expédiée puisque l'utilisation de la SFML dépend de concepts propres à la POO en C++. C'est donc naturellement que notre choix s'est penché vers la SDL.

Compte tenu de la simplicité du programme, nous nous somme contenter de compiler notre programme à l'aide du terminal de commande Linux et de « g++ », nous avons donc jugé peu utile l'emploi d'IDE comme Code::Blocks contrairement à ce nous imaginions au départ du projet. Ceci a permis une liberté à chacun en ce qui concerne le choix de l'éditeur de texte à employé qui a été unique pour chaque membre du groupe :

- Charbel : Emacs
- Ambre : Sublime Text
- Aurélien : Atom

En ce qui concerne la production visuelle nécessaire au projet, celle-ci a nécessité l'emploi du logiciel Paint.NET qui se place dans la veine des logiciels d'imagerie tels que Photoshop, GIMP… L'emploi de Paint.NET relève du choix d'Ambre, seule responsable des créations graphiques, qui a voulu se servir d'un logiciel dont elle maîtrisait certains aspects. Par ailleurs, le projet était pour elle une occasion en or d'approfondir sa maîtrise du logiciel.

Analyse du projet

Organisation des méthodes et des variables

Notre programme étant un jeu-« vidéo », il y a donc évidemment une composante vidéo dans la programmation. Cependant, nous avons jugé important de différencier ce que nous appelons l'environnement de jeu, qui correspond aux variables de jeu (la position de l'ennemi, le sort employé…) sur lesquels s'appliquent les règles par le biais de méthodes, et le jeu de variables et de méthodes graphiques qui sont eux dépendants des variables de l'environnement de jeu.

Cette analyse nous permet donc de découper notre boucle de jeu en trois parties distinctes :

- La lecture et la mémorisation des entrées utilisateurs
- La mise à jour de l'environnement de jeu (éventuellement en fonction des entrées)
- La mise à jour du jeu du rendu graphique (en fonction de l'environnement de jeu)

Ainsi, nous séparons les méthodes dans 3 fichiers différents et nous les réemployons toutes directement ou indirectement dans la fonction main du programme.

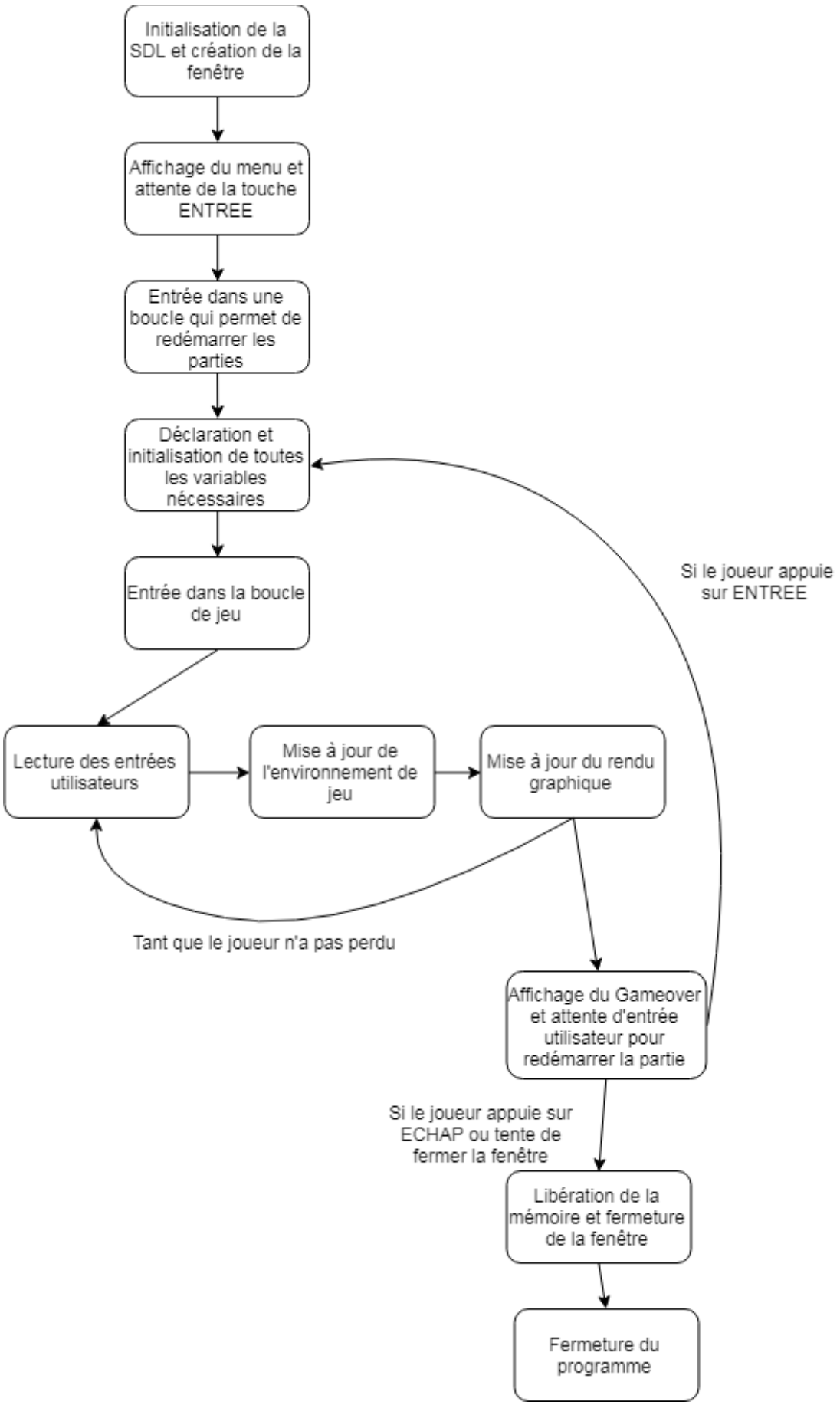


Figure 5: Déroulement du programme

Il y a au tout début une première déclaration de variables, lors de l'initialisation de la SDL et de la création de la fenêtre. Ces variables sont de types SDL_Renderer* et SDL_Window*. On emploie donc une méthode CreateWindow dans laquelle sont employés des fonctions SDL pour créer la fenêtre du programme.

On affiche ensuite avec une méthode DisplayMenu une image et on rentre dans la boucle du menu principal dans laquelle le programme attend que le joueur appuie sur ENTREE pour continuer dans le jeu.

A cette étape du programme, il va falloir déclarer les variables de jeu, cependant on souhaite pouvoir placer tout le déroulement dans une boucle afin de pouvoir redémarrer la partie en cas d'échec. Pour cela on utilise une variable de type booléen qui pourra être mis sur false en cas de gameover.

On déclare et définit donc différentes variables dans la boucle de jeu. Certaines servent simplement aux graphismes tandis que d'autres correspondent à l'état du jeu et sont utilisées à la fois pour déterminer la suite du jeu (mort du joueur, génération d'un nouveau monstre…) que pour positionner des sprites à l'écran.

Variables relatives aux entrées utilisateurs :

-input : int

(Un entier qui varie selon les entrées utilisateurs et qui est utilisé pour lancer un sort ou quitter le jeu)

Variables relatives à l'environnement de jeu :

-enemyPosition : float

(La position de l'ennemi qui justifie ou non la défaite)

-gameover : bool

(Un booléen qui permet de déterminer si la boucle de jeu doit continuer ou non)

-enemyId : int

(Le type de l'ennemi qui justifie si celui-ci doit mourir selon le sort utilisé)

-spellId : int

(Un identifiant du sort qui justifie si l'ennemi meurt ou non)

-canCast : bool

(Un booléen permettant de savoir si le joueur à le droit de lancer un sort ou non)

-enemyWeaknesses : int[3]

(Un tableau qui donne a chaque identifiant d'ennemi l'identifiant de sort qui le tue)

-enemyAlive : bool

(Un booléen qui détermine si l'ennemi est vivant, et si non en génère un nouveau)

-gameSpeed : float

(La vitesse du jeu qui influe sur la vitesse de l'ennemi)

-score : int

(Un entier représentant le nombre d'ennemis vaincus à la suite)

Variables relatives au rendu graphique du jeu :

-heroSpritePositionY : int

(Un entier servant à donner une impression que le personnage lévite)

-heroSpriteGoingUp : bool

(Un booléen permettant d'alterner entre faire monter et descendre le sprite du hero)

-enemySpritePositionX : int

(Un entier qui donne la position du sprite ennemi dans l'écran selon enemyPosition)
-wall1SpritePositionX : int
(Un entier qui permet de faire défiler un mur de fond)
-wall2SpritePositionX : int
(Un autre entier utilisé pour faire défiler un mur de fond)
-scoreFont : TTF_Font*
(Une police utilisée pour l'affichage du score)
-black : SDL_Color
(Une couleur employée pour le score)

Il est important de cerner le fait que les variables relatives au rendu graphique sont mises à jour en fonction des variables relatives à l'environnement de jeu et que les variables relatives à l'environnement de jeu sont mises à jour en fonction de la variable input.

Ainsi, on retrouve donc dans la boucle de jeu trois méthodes qui sont appelées afin de s'accorder avec le schéma d'exécution du programme :
+UpdateInput
+UpdateEnvironment
+UpdateGraphics
Bien souvent, ces fonctions utilisent en arguments des références à certaines variables afin de les mettre à jour dans le programme principal en coordination avec la boucle de jeu.

Les entrées utilisateurs et l'environnement de jeu

Il faut donc en premier lieu parvenir à créer un système dans lequel l'environnement de jeu peut évoluer à partir des entrées utilisateurs. Les méthodes UpdateInput et UpdateEnvironment fonctionnent donc de la sorte :

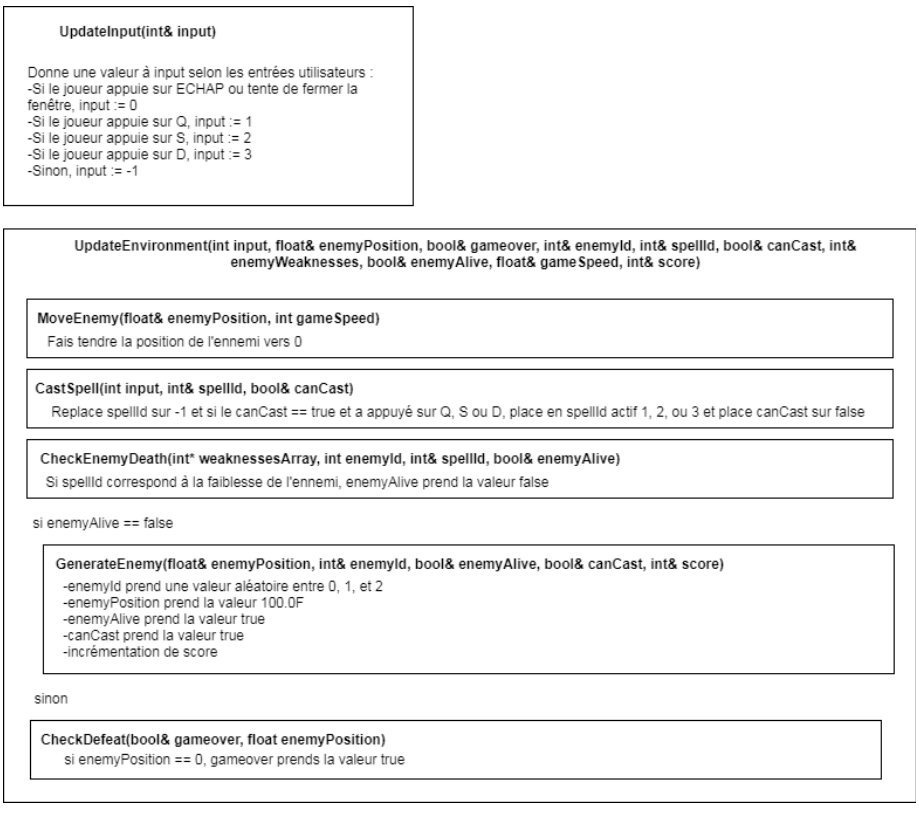


Figure 6: Méthodes UdpateInput et UpdateEnvironment

Graphismes

En ce qui concerne les graphismes, il fallait tout d'abord établir une liste d'éléments à afficher à l'écran :

- un menu
- le personnage joueur
- l'ennemi
- un élément de HUD qui permet au joueur de savoir s'il peut lancer un sortilège
- le score du joueur
- le sortilège lancé
- un arrière plan

```
253     //Nettoyage de l'écran
254     SDL_RenderClear(renderer);
255     //Choix de ce qui va être afficher et où
256     DisplayWall(renderer, sprWall1Position, sprWall2Position);
257     DisplayScore(renderer, score, scoreFont, white);
258     DisplayBook(renderer, canCast);
259     DisplayHero(renderer, sprHeroPosition, goingUp);
260     DisplaySpell(renderer, sprEnemyPosition, spellId);
261     DisplayEnemy(renderer, sprEnemyPosition, enemyPosition, enemyId);
262     DisplayBook(renderer, canCast);
263     //Affichage de l'écran
264     SDL_RenderPresent(renderer);
265     //Légère pause si le joueur lance un sort
266     if(spellId != -1 && spellId != 0)
267     {
268       SDL_Delay(500);
269     }
```

Figure 7: Déroulement de la fonction UpdateGraphics

Pour chacun de ces éléments, nous utilisons une méthode qui est employé dans la méthode UpdateGraphics. Chacune de ces méthodes utilise des variables qui lui sont propres selon ce qu'elle doit accomplir. Par exemple, la méthode responsable de l'affichage de l'ennemi utilise les variables enemyId, enemyPosition et enemySpritePositionX. Cependant, il est important de remarquer que toutes ces méthodes reçoivent en argument l'adresse de la variable renderer déclarée en début de programme. Plus de détail sur le renderer seront donnée dans la suite de ce rapport.

Pour pouvoir effectuer des rendus graphiques, ceux-ci doivent se dérouler dans un fenêtre. Dès le début, du programme est lancé une fonction CreateWindow. Celle-ci prend en argument deux variables, déclarées juste avant l'exécution de la fonction, de types propres à la SDL :
SDL_Window* window ;
SDL_Renderer* renderer ;

Afin de créer une fenêtre avec la SDL, il faut utiliser la fonction SDL_CreateWindow en lui donnant en argument des informations sur la fenêtre (nom, résolution). Ensuite, afin de pouvoir « dessiner » dans cette fenêtre, nous devons créer un renderer qui peut afficher des éléments visuels dans la fenêtre qu'on lui associe.

Pour pouvoir accéder à cette fenêtre et à ce renderer, il est nécessaire qu'ils soient attribués à des variables.

```
//Generating window
window = SDL_CreateWindow("MageRunner", 0, 0, 16*80, 9*80, 0);
renderer = SDL_CreateRenderer(window, -1, 0);
```

Figure 8 : Attribution d'une fenêtre créée et d'un renderer aux variables

Une fois la fenêtre créée, nous devons être capable de la détruire (à la fin du programme). Pour cela nous utilisons une fonction DestroyWindow(..) qui prend en argument la fenêtre et le renderer que nous avons déclarés dans le main et qui utilise deux fonctions de la SDL pour les détruire et libérer la mémoire. Mais avoir une fenêtre ne suffit pas, il faut aussi pouvoir dessiner dedans : c'est à ce moment qu'entre en scène le renderer et la fonction UpdateGraphics.

La fonction UpdateGraphics(..) est placée dans la boucle de jeu qui consiste en la lecture des entrées utilisateur, de la mise à jour de l'environnement de jeu et finalement de la mise à jour des graphismes. Elle prend donc en argument les variables window et renderer et les variables graphiques évoquées précédemment et se déroule en 3 temps :
-le nettoyage du renderer
-l'écriture dans le renderer
-l'affichage du renderer

Le nettoyage se fait grâce à une fonction de la SDL : SDL_RenderClear(renderer) ;

L'écriture dans le renderer est de loin l'étape la plus compliqué, nous avons choisi de lancer une fonction par élément à afficher, il s'agit donc du héro et de l'ennemi. Cela correspond à deux fonctions, la fonction DisplayHero qui prend en argument le renderer et en référence la variable heroSpritePositionY et le booléen heroSpriteGoingUp, et la fonction DisplayEnnemi qui prend en arguments les variables renderer, enemySpritePositionX, enemyPosition et enemyId.

La fonction DisplayHero fonctionne en créant deux variables, une de type SDL_Surface* qui va permettre de recopier les informations visuelles d'un fichier et une autre de type SDL_Texture* qui recopie la surface pour être ensuite affichée dans le renderer à l'aide d'une fonction SDL_RenderCopy qui prend en argument la texture à écrire, le renderer dans lequel on dessine la texture, et des infomations sur celle-ci (taille et position dans l'écran). Le positionnement se fait donc à partir de la variable heroSpritePosition.

La fonction DisplayEnemy fonctionne de manière similaire, sauf que la surface recopie un fichier différent selon le type de l'ennemi. En ce qui concerne la position de la texture dans l'écran, celle-ci se fait selon la variable enemyPosition

.

```
42  void DisplayHero(SDL_Renderer* renderer, int& heroSpritePositionY, bool& heroSpriteGoingUp)
43  {
44    SDL_Surface* heroSurface;
45    SDL_Texture* heroTexture;
46
47    heroSurface = IMG_Load("hero.png");
48    heroTexture = SDL_CreateTextureFromSurface(renderer, heroSurface);
49    SDL_FreeSurface(heroSurface);
50
51    if(heroSpriteGoingUp)
52    {
53      ++heroSpritePositionY;
54    }
55    else
56    {
57      --heroSpritePositionY;
58    }
59
60    if(heroSpritePositionY == 20 || heroSpritePositionY == 0)
61    {
62      heroSpriteGoingUp = !heroSpriteGoingUp;
63    }
64
65    SDL_Rect heroTransform;
66    heroTransform.x = -20;
67    heroTransform.y = 460 + heroSpritePositionY;
68    heroTransform.w = 320;
69    heroTransform.h = 240;
70
71    SDL_RenderCopy(renderer, heroTexture, NULL, &heroTransform);
72    SDL_DestroyTexture(heroTexture);
73  }
```

Figure 9: Fonction DisplayHero qui permet d'afficher le héro à l'écran

En réalité, toutes les fonctions Display fonctionnent sur les mêmes principes, il y a juste des cas qui rendent certaines uniques (comme le personnage se déplaçant de haut en bas par exemple)

Après avoir programmé cela, il y avait malgré tout un problème : le programme voyait sa consommation en mémoire augmenter sans cesse jusqu'à faire planter la machine sur laquelle il fonctionnait. Cela était dû au fait que les variables déclarées de type SDL_Texture* et SDL_Surface* continuait d'occuper de la place dans la mémoire. Pour régler ce problème, il a suffit d'ajouter dans les fonctions Display des fonctions de la SDL libérant les espaces mémoires occupés, respectivement SDL_FreeSurface et SDL_DestroyTexture.

Développement

Découpage du code

Les différentes fonctions employées dans le programme sont en réalité réparties entre plusieurs fichiers dont chacun à un rôle clair et distinct :

- graphics.cpp/graphics.hpp
- environment.cpp/environment.hpp
- input.cpp/input.hpp
- main.cpp (dans lequel se déroule tout le programme)

Le début du programme

Au début du programme sont déclarées et parfois définies des variables utilisées tout le long du programme. Nous voulions agit différemment en déclarant les variables des les autres fichiers (environment.cpp…) mais cela causait des problèmes lors de l'utilisation des fonctions.

```
    //Initialisation de la SDL et création de la fenêtre
    SDL_Init(SDL_INIT_VIDEO);
    IMG_Init(IMG_INIT_PNG);
    TTF_Init();
    SDL_Window* window;
    SDL_Renderer* renderer;
    CreateWindow(window, renderer);

    //Initialisation de l'aléatoire
    srand(time(NULL));

    //menu variables
    bool quitMenu;
    quitMenu = false;

    DisplayMenu(renderer);
    while (quitMenu == false) {
      MenuInput(quitMenu);
    }

    bool replay;
    replay = true;

    //Initializing input
    int input;

    while(replay && input != 0)
    {
      input = -1;

      //Initializing game environment
      float enemyPosition;
      enemyPosition = 100.0F;
      bool gameover;
      gameover = false;
      int enemyId;
      enemyId = 0;
      int spellId;
      spellId = -1;
      bool canCast;
      canCast = true;
      int enemyWeaknesses[3];
      enemyWeaknesses[0] = 1;
      enemyWeaknesses[1] = 2;
      enemyWeaknesses[2] = 3;
      bool enemyAlive;
      enemyAlive = true;
      float gameSpeed;
      gameSpeed = 9.0F;
      int score;
      score = 0;
```

```

      //Initializing graphics variables
      int heroSpritePositionY;
      heroSpritePositionY = 0;
      bool heroSpriteGoingUp;
      heroSpriteGoingUp = true;
      int enemySpritePositionX;
      enemySpritePositionX = 0;
      int wall1SpritePositionX;
      wall1SpritePositionX = 0;
      int wall2SpritePositionX;
      wall2SpritePositionX = 1280;
      //Variables graphiques pour le score
      TTF_Font* scoreFont;
      scoreFont = TTF_OpenFont("arial.ttf", 64);
      SDL_Color white;
      white.a = 0;
      white.r = 255;
      white.g = 255;
      white.b = 255;

      //gameloop
```

Figure 10: Début du programme et entrée dans la première boucle

La boucle de jeu

Dans la boucle de jeu sont donc envoyé en arguments références les différentes variables instanciées précédemment dans 3 fonctions afin de faire persister et évoluer l'état du jeu.

```
//Boucle de jeu
while(!gameover && input != 0)
{
  UpdateInput(input);
  UpdateEnvironment(input, enemyPosition, gameover, enemyId, spellId, canCast, enemyWeaknesses, enemyAlive, gameSpeed, score);
  UpdateGraphics(window, renderer, heroSpritePositionY, heroSpriteGoingUp, enemySpritePositionX, enemyPosition, enemyId, wall1SpritePositionX, wall2SpritePositionX, score, scoreFont, white, spellId, canCast);
}
```

Figure 11: Déroulement de la boucle de jeu

Lecture des entrées utilisateurs

La première chose à faire a été de trouver un moyen de lire les entrées utilisateurs pour pouvoir les utiliser dans le programme. Nous avons donc décidé de déclarer une variable input dans la boucle de partie (pas celle de jeu!) qui serait passée en argument référence de la méthode UpdateInput.

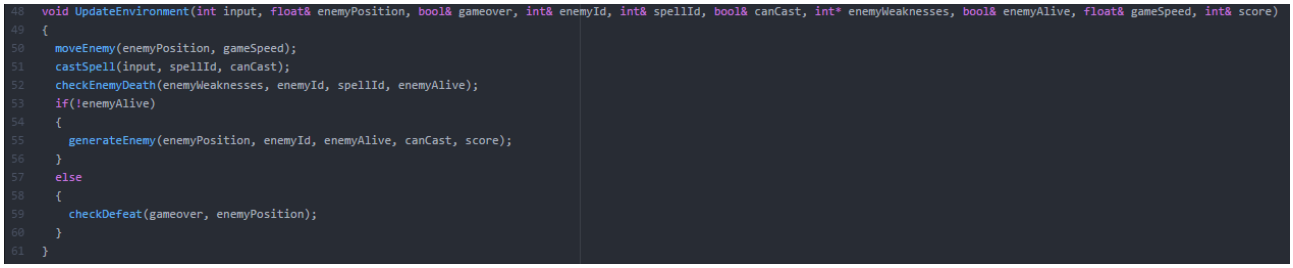
```
void UpdateInput(int& input)
{
    SDL_Event event;

    input = -1;

    SDL_Delay(17);
    SDL_PollEvent(&event);

    switch (event.type)
    {
      case SDL_QUIT:
        input = 0;
        break;

      case SDL_KEYDOWN:
        switch(event.key.keysym.sym)
        {
          case SDLK_ESCAPE:
            input = 0;
            cout << "PRESSED ESCAPE" << endl;
            break;

          case SDLK_q:
            input = 1;
            cout << "PRESSED Q" << endl;
            break;

          case SDLK_s:
            input = 2;
            cout << "PRESSED S" << endl;
            break;

          case SDLK_d:
            input = 3;
            cout << "PRESSED D  " << endl;
            break;
        }
      }

}
```

Figure 12: Fonction de lecture des entrées utilisateurs utilisée dans la boucle de jeu

Nous ne couvrons pas la totalité des fonctions de ce type pour éviter de rendre le rapport rébarbatif.

La mise à jour de l’environnement

La mise à jour de l’environnement se fait comme évoqué précédemment en utilisant la méthode UpdateEnvironment. Cette solution fut assez simple à implémenter car il suffisait simplement de se servir des fonctionnalités du C++ pour le passage en référence.



```
void UpdateEnvironment(int input, float& enemyPosition, bool& gameover, int& enemyId, int& spellId, bool& canCast, int* enemyWeaknesses, bool& enemyAlive, float& gameSpeed, int& score)
{
  moveEnemy(enemyPosition, gameSpeed);
  castSpell(input, spellId, canCast);
  checkEnemyDeath(enemyWeaknesses, enemyId, spellId, enemyAlive);
  if(!enemyAlive)
  {
    generateEnemy(enemyPosition, enemyId, enemyAlive, canCast, score);
  }
  else
  {
    checkDefeat(gameover, enemyPosition);
  }
}
```

Figure 13: La fonction UpdateEnvironment

Le rendu graphique

La partie programmation graphique a été, et de loin, la tâche plus complexe à mettre en œuvre. En effet, savoir établir de bons algorithmes ne suffisait pas : il fallait faire l’effort de prendre en main la SDL pour pouvoir y parvenir. Et cela ne nous a pas épargné d’autres soucis ! En effet, l’emploi de certaines images nous posait un énorme problème de performances sans être capable de comprendre pourquoi. Nous avons heureusement bénéficié de l’aide d’un des professeurs qui nous a suggéré de réduire la résolution du fichier PNG utilisé, ce qui était bien la source du problème.

Manuel d'utilisation

Déroulement

Le fonctionnement de l'application reste des plus simples. En effet, dès le lancement apparaît le menu qui invite le joueur à appuyer sur ENTREE pour pouvoir lancer la partie.
Une fois la partie lancée le jeu démarre et continue ainsi jusqu'à la défaite du joueur.
Quand le joueur perd la partie, le menu de Gameover l'invite à recommencer la partie ou à quitter le jeu en lui indiquant les touches sur lesquels appuyer.

Comment jouer ?

Tout jeu qui se respecte apporte avec lui son jeu de règles (jeu de mot (jeu de mot)) que le débutant doit connaître avant de se jeter dans le bain. Ici les règles sont très simples : les démons apparaissent et se dirigent vers le joueur. Le joueur pour se défendre doit se servir du bon sortilège pour vaincre le démon. S'il se trompe de sortilège face à un démon, il n'aura plus la chance d'en utiliser un et sera condamné à mourir ! Heureusement, quiconque à joué à Pokémon comprendra aisément quel sort détruit quel démon. En effet, le démon rouge est détruit par le sort bleu, le démon bleu par le sort vert, et le démon vert par le sort rouge.

Démon	Rouge	Bleu	Vert
Faiblesse	Bleu	Vert	Rouge

Il faut donc pour le joueur faire coïncider les faiblesses des monstres avec les bonnes touches pour les détruire.

Touche	Q	S	D
Sort	Bleu	Vert	Rouge

Le joueur n'a pas besoin au cours de sa partie d'employer d'autres boutons.
Il faut savoir qu'à tout moment de la partie, le joueur peut appuyer sur ECHAP afin de faire fermer le programme.

Conclusion

Arrivé au bout des deadlines le cahier des charges n'est pas achevé, et ce, seulement en ce qui concerne l'implémentation de sons et de musiques au jeu. Comment cela se fait ? Et bien il s'agit là du résultat d'un problème rencontré en employant les capacités audio de la SDL. En effet, si en suivant des tutoriels pas à pas nous avons réussi à placer dans des sons dans le jeu, notre programme était incapable de continuer à jouer du son au bout du 15ème joué. En tentant de résoudre en vain ce problème, nous nous sommes effectivement rendu compte que nous ne maîtrisions pas du tout leur emploi et qu'il s'agissait simplement de la copie d'un code sur Internet, chose que nous n'acceptons pas de faire.

En ce qui concerne le fonctionnement de l'application elle même, nous n'avons rien à redire, nous ne rencontrons pas de bugs, ni de gros problèmes de performances. Nous sommes plutôt fier d'être parvenu à mettre en place ce programme et d'avoir fourni le travail annexe à la programmation, tel que la rédaction de rapport, la gestion du projet, ou bien encore la production de ressources visuelles.

Même si nous ne sommes pas parvenu à remplir tout notre engagement, nous ne trouvons pas que la surcharge de travail était énorme. À vrai dire, avec l'expérience obtenue, que ce soit en programmation, l'utilisation d'un librairie comme la SDL, en travail d'équipe, en gestion de projet ou bien encore avec les logicielles de production visuelle, nous pourrions nous lancer dans un projet bien plus ambitieux afin de grandement s'améliorer.

Cependant nous nous sommes rendu compte de ce qui est un problème dans les travaux de groupes de ce style là, il s'agit bien du respect de ses engagements envers le groupe et de savoir prendre le temps pour travailler sur ce qui est prévu et nécessaire. En effet, il faut parfois se forcer à travailler sur le projet pour ne pas laisser notre groupe dans un terrible embarras et leur permettre d'avancer.

Pour conclure, nous avons trouvé cette expérience très enrichissante, car en effet, s'il ne s'agit pas nécessairement d'un exploit, ce projet aura su nous pousser à donner le meilleur possible, que ce soit pour fournir un programme digne de ce nom, que pour accomplir le but de cette UE : avoir une bonne expérience dans le travail de groupe.

Webographie

-Documentation de la SDL2 :
https://wiki.libsdl.org/CategoryAPI

-Tutoriel dans lequel nous avons puisé des aides :
https://openclassrooms.com/courses/apprenez-a-programmer-en-c/installation-de-la-sdl

-Répertoire GitHub du projet :
https://github.com/Shaimoogle/CMI-Runner-Game