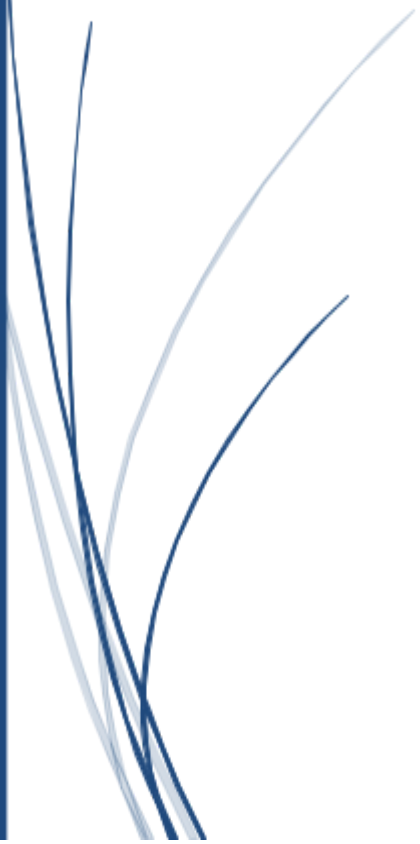
POLYTECH SORBONNE

SORBONNE UNIVERSITÉ

Rapport C++

Année universitaire
2021/2022

There is no planet B



Youssef LAKHDHAR
Constantinos SAVVIDIS

Objectif de notre projet:

Pour notre projet, nous avons souhaité concevoir un jeu mettant en avant l'environnement et sa protection. Notre objectif était de créer un jeu simple et amusant pour qu'il soit populaire auprès des enfants, tout en restant un minimum éducatif afin de les familiariser avec le principe du tri sélectif, en les aidant à mémoriser les couleurs de celui-ci et ainsi pouvoir l'utiliser dans leur vie courante.

Description du jeu:

Notre jeu a lieu dans un futur apocalyptique de notre planète, où la pollution a atteint un niveau extrêmement élevé au point qu'une partie de la population s'est transformée une espèce de « zombies ».
Afin d'aider peu à peu à reconstruire la société, votre objectif est de ramasser tous les déchets avant que le temps s'écoule, tout en respectant les couleurs des poubelles afin de correctement recycler tous les déchets.

Mode d'emploi:

Une fois le jeu lancé, vous pouvez diriger votre personnage avec les flèches du clavier afin de naviguer sur la map. Il faut par ailleurs éviter les zombies à tout pris, sinon vous vous retrouvez à l'autre bout de la carte et perdez du temps, le but étant de ramasser un nombre minimal de déchets dans le temps imparti.
Pour collecter les déchets, il suffit de de passer dessus… Une fois le déchet ramassé, il faut naviguer jusqu'aux poubelles pour les jeter. En choisissant la bonne couleur de poubelle, vous gagnez des points.
Ainsi, la partie s'achève lorsque le temps est écoulé (défaite) ou que vous avez collecté un nombre suffisant de déchets (victoire).

Installation:

L'installation de la librairie graphique SFML est nécessaire pour pouvoir lancer ce jeu.
La procédure d'installation de la librairie se trouve à l'adresse suivante
https://www.sfml-dev.org/tutorials/2.5/start-linux-fr.php#:~:text=Diff%C3%A9rentes%20approches%20sont%20possibles%20pour,et%20copier%20les%20fichiers%20manuellement

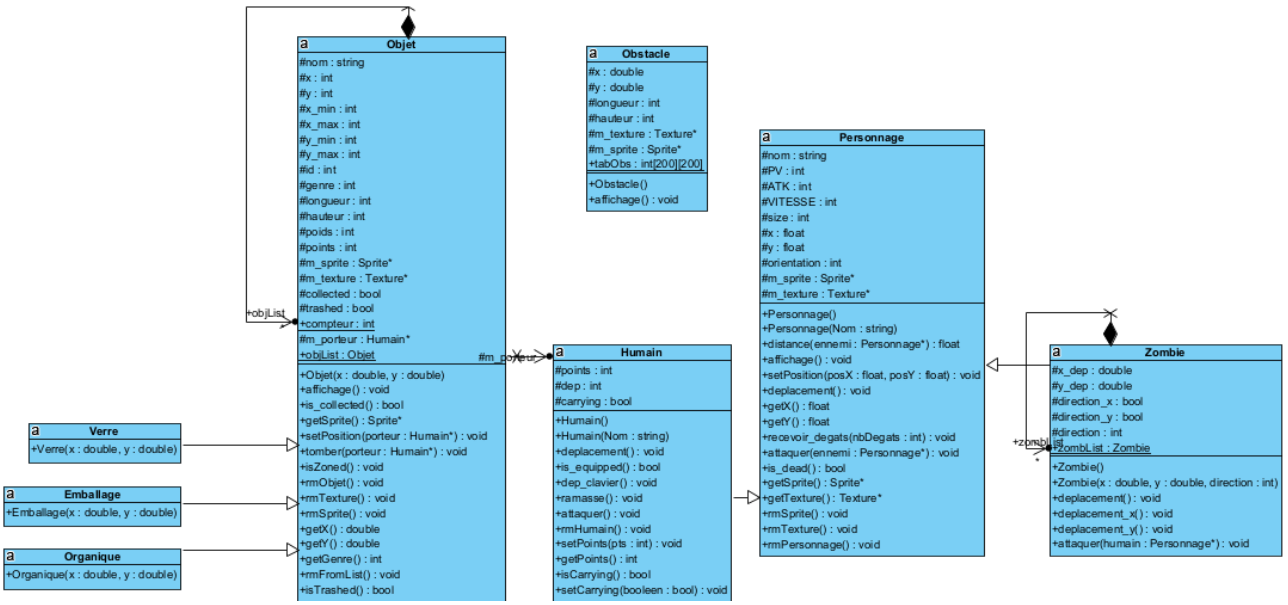
(Une fois la librairie installée, il suffit simplement d'effectuer la commande make dans un terminal afin de compiler le projet puis de rentrer la commande ./miam afin de lancer le jeu.)

SFML est une librairie graphique permettant l'affichage de sprites.
On retrouve régulièrement les mêmes bouts de code à l'intérieur au sein de notre programme afin de charger une image dans une texture et d'affecter cette texture à un sprite pour ensuite l'afficher.
Ci-dessous un des nombreux exemples d'affichage de sprites revenant dans ce code.

```
15    m_texture = new sf::Texture;
16    m_texture->loadFromFile("sprites_link.png", sf::IntRect(24, 64, 24, 32));
17    m_sprite = new sf::Sprite;
18    m_sprite->setTexture(*m_texture);
19    m_sprite->setPosition(sf::Vector2f(x,y)); //DEPLACEMENT DU SPRITE
```

La ligne 19 (pas toujours présente) sert à donner sa position au sprite.

Diagramme UML:



Architecture du code:

On peut visualiser ci-dessus un diagramme UML de notre code, avec les différentes classes. On peut constater que notre modélisation met en avant 4 grandes classes interagissant avec les autres: *Objet*, *Personnage*, *Humain* et *Zombie*.
De la classe *Objet* héritent les 3 types de déchets: Emballage, *Organique* et *Verre*.
Les classes *Humain* et *Zombie* héritent quant à elles de *Personnage*.
Enfin, la classe Obstacle est seule car elle n'a aucun point commun avec les autres.
Une classe *Spritable* regroupant les différents attributs et méthodes nécessaires à l'affichage de toutes ces entités et dont toutes les classes *Personnage* et *Objet* hériteraient a été envisagée et expérimentée. Cependant, de trop nombreuses erreurs de segmentation dont la source étant connues mais insolvables a empêché l'implémentation de cette solution.

La classe *Personnage* regroupe les attributs généraux de nos personnages.

La classe *Humain* gère tout ce qui est en lien avec notre personnage principal (Roger), que ce soit sa visualisation (via l'utilisation des sprites), ses attributs, ou même son déplacement.
On a ci-dessous le code la fonction de déplacement du personnage, fonction essentiel permettant de gérer ses déplacements, les interactions avec son environnement et surtout son affichage.

```
27  void Humain::deplacement()
28  {
29    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) && y > 0)
30    {
31      m_texture->loadFromFile("sprites_link.png",sf::IntRect(0,0,32,32));
32      if (Obstacle::tabObs[int(x)/5][int(y/5)-1] == 0)
33      {
34        setPosition(x,y - dep);
35        m_sprite->setPosition(sf::Vector2f(x, y - dep));
36      }
37    }
38    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down) && y < SIZE_Y - H_SPRITE)
39    {
40      m_texture->loadFromFile("sprites_link.png",sf::IntRect(24,64,24,32));
41      if (Obstacle::tabObs[int(x)/5][int(y/5)+1] == 0)
42      {
43        setPosition(x,y + dep);
44        m_sprite->setPosition(sf::Vector2f(x, y + dep));
45      }
46    }
47    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right) && x < SIZE_X - W_SPRITE)
48    {
49      m_texture->loadFromFile("sprites_link.png",sf::IntRect(0,32,32,32));
50      if (Obstacle::tabObs[int(x/5)+1][int(y/5)] == 0)
51      {
52        setPosition(x + dep,y);
53        m_sprite->setPosition(sf::Vector2f(x + dep, y));
54      }
55    }
56    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left) && x > 0)
57    {
58      m_texture->loadFromFile("sprites_link.png",sf::IntRect(0,96,32,32));
59      if (Obstacle::tabObs[int(x/5)-1][int(y/5)] == 0)
60      {
61        setPosition(x - dep,y);
62        m_sprite->setPosition(sf::Vector2f(x - dep, y));
63      }
64    }
65  }
```

Le code peut se résumer à une scrutation d'événements. En effet, les entrées clavier sont scrutées et dès lors qu'une flèche directionnelle est pressée (lignes 29,38,47,56), alors une suite d'instructions est exécutée:

- On modifie l'orientation du personnage en changeant de personnage sur la feuille de sprites (lignes 31,40,49,58).
- On vérifie que la prochaine case est libre, c'est-à-dire sans obstacles ni limites de la fenêtre(lignes 32,41,50,59).
    - Si oui, on y déplace le personnage et on met à jour ses coordonnées (lignes 34,43,52,61). On déplace ensuite son sprite pour le rendu visuel (lignes 35,44,53,62).
    - Sinon, la seule action performée aura été le changement d'orientation.

Le personnage pouvant à présent se déplacer et interagir avec son environnement, il faut maintenant gérer le comportement des zombies. ainsi que leur affichage.

La classe Zombie contient une fonction clé pour les zombies, à savoir leur déplacement. Ci-dessous le code de cette fonction.

```
19  void Zombie::deplacement()
20  {
21    if (direction == 0)
22      deplacement_x();
23    else
24      deplacement_y();
25  }
26
27  void Zombie::deplacement_x()
28  {
29    if((x_dep - x == -250)||(x_dep - x == 0)){
30      direction_x = !(direction_x);
31    }
32    if((x_dep - x > -250)&&(direction_x == 1)){
33      m_texture->loadFromFile("sprites_zombie.png", sf::IntRect(0, 128, 32, 64));
34      x = x + 5;
35    }
36    if((x_dep - x < 0)&&(direction_x == 0)){
37      m_texture->loadFromFile("sprites_zombie.png", sf::IntRect(0, 64, 32, 64));
38      x = x - 5;
39    }
40    m_sprite->setPosition(sf::Vector2f(x,y));
41  }
42
43  void Zombie::deplacement_y()
44  {
45    if((y_dep - y == -250)||(y_dep - y == 0)){
46      direction_y = !(direction_y);
47    }
48    if((y_dep - y > -250)&&(direction_y == 1)){
49      m_texture->loadFromFile("sprites_zombie.png", sf::IntRect(0, 0, 32, 64));
50      y = y + 5;
51    }
52    if((y_dep - y < 0)&&(direction_y == 0)){
53      m_texture->loadFromFile("sprites_zombie.png", sf::IntRect(0, 192, 32, 64));
54      y = y - 5;
55    }
56    m_sprite->setPosition(sf::Vector2f(x,y));
57  }
```

Lors de l'instanciation d'un zombie, on renseigne un paramètre *direction* permettant de choisir si le zombie se déplacera de manière horizontale (0) ou verticale (1). On peut ainsi lire que l'ennemi aura une amplitude de mouvements de 250 pixels. Enfin, à

chaque retournement, le sprite de l'ennemi est également modifié de manière à ce qu'il regarde toujours devant lui, un peu comme notre cher Roger.
Cependant, les zombies n'étant pas des entités uniques mais bien un groupe, il a été choisi de les regrouper au sein d'un même vecteur statique afin de faciliter les fonctions d'affichage et notamment la gestion des interactions avec Roger.
Ci-dessous les fonctions de création du vecteur de zombies (ligne 64) et de leur affichage (ligne 73).

```
63
64  void remplirListe()
65  {
66    Zombie::zombList.push_back(Zombie(200,200,0));
67    Zombie::zombList.push_back(Zombie(500,300,0));
68    Zombie::zombList.push_back(Zombie(200,100,1));
69    Zombie::zombList.push_back(Zombie(150,400,0));
70    Zombie::zombList.push_back(Zombie(535,100,1));
71  }
72
73  void afficherZombies(sf::RenderWindow *window)
74  {
75    for (int i = 0; i < Zombie::zombList.size(); i++)
76    {
77      Zombie::zombList[i].deplacement();
78      if (Zombie::zombList[i].getSprite())
79      {
80        if(Zombie::zombList[i].getSprite())
81          window->draw(*(Zombie::zombList[i].getSprite()));
82      }
83    }
84  }
```

Roger et les zombies pouvant à présent être affichés et se déplacer, il faut gérer leurs interactions. Celles-ci sont gérées via la méthode *attaquer()* de *Humain*, même si il serait plus cohérent de la mettre dans la classe *Zombie*; en effet, le zombie attaque Roger et non pas l'inverse. Mais cela nous confrontait à un problème de dépendance circulaire, c'est pourquoi nous avons adopté la solution ci-dessous.

```
67  void Humain::attaquer()
68  {
69      for (int i = 0; i < Zombie::zombList.size(); i++)
70      {
71        if (this->distance(&Zombie::zombList[i]) < 30)
72          setPosition(50,50);
73      }
74  }
```

Ainsi, à chaque passage de boucle, on vérifie si Roger est très près d'un zombie (à moins de 30 pixels). Si c'est le cas, alors Roger est jeté à l'endroit le plus éloigné de la zone de tri. Ainsi, le joueur perd du temps pour collecter les objets, sur lesquels nous allons à présent nous pencher, et notamment sur leur implémentation au sein du jeu.

Les objets sont gérés via la classe éponyme *Objet*. Cette classe contient plusieurs attributs tels que les coordonnées du personnage ainsi que les différents objets relatifs à son affichage et sa manipulation (texture, sprite).
Il avait été en premier lieu de réaliser des déchets (objets à collecter) et des bonus (tels que des boosts de vitesse ou de l'invulnérabilité aux zombies) mais cette idée aurait demandé un travail trop conséquent et infaisable au vu des dates de rendu.

Au sein de la classe *Objet* se trouve la méthode assurant la fonction fondamentale d'un objet: pouvoir être collecté.
Ci-dessous son code.

```
26  void Objet::setPosition(Humain *porteur)
27  {
28    if (!this->collected)
29    {
30      if ((x == porteur->getX() && y == porteur->getY()) && !porteur->isCarrying())
31      {
32        this->collected = true;
33        m_porteur = porteur;
34        porteur->setCarrying(true);
35      }
36    }
37    else
38    {
39      m_sprite->setPosition(sf::Vector2f(porteur->getX(),porteur->getY()+16));
40      x = porteur->getX();
41      y = porteur->getY();
42      if (sf::Keyboard::isKeyPressed(sf::Keyboard::X))
43      {
44        this->tomber(porteur);
45      }
46    }
47  }
```

Si l'objet n'a pas encore été collecté et que Roger passe dessus en n'étant pas chargé, alors l'objet est ramassé et ses coordonnées deviennent celles de Roger.
Sinon, cela signifie que l'objet a été ramassé et que Roger le porte actuellement. Les coordonnées du sprite sont alors à peu près celles de la main de Roger pour un meilleur rendu visuel. On scrute alors l'appui sur la touche X, qui correspond au lâcher de l'objet par Roger.
Ci-dessous le code correspondant à cette action (la méthode *tomber()*).

```
49  void Objet::tomber(Humain *porteur)
50  {
51    x = porteur->getX();
52    y = porteur->getY();
53    this->isZoned();
54    m_porteur = NULL;
55    this->collected = false;
56    porteur->setCarrying(false);
57  }
```

Les coordonnées de l'objet deviennent celles du lieu du lâcher. On met alors à jour l'attribut *collected* de l'objet d'une part pour dire qu'il est par terre et l'attribut *carrying* de Roger d'autre part pour signifier qu'il est prêt à collecter de nouveaux objets.
Enfin, la méthode *isZoned()* est également appelée: c'est elle qui gère les points.
Ci-dessous le code de la méthode *isZoned()*.

```
59  void Objet::isZoned()
60  {
61    if ((x_min < x && x_max > x) && (y_min < y && y_max >y))
62    {
63      x = 900;
64      y = 900;
65      m_porteur->setPoints(m_porteur->getPoints() + points);
66      this->rmObjet();
67    }
68  }
```

Dans cette méthode on s'assure tout d'abord que l'objet est dans la bonne poubelle, c'est à dire entre x_min et x_max d'une part et y_min et y_max d'autre part. Ces 4 variables sont réglées dans le constructeur de chacune des classes décrivant un type de déchet: *Emballage*, *Organique* et *Verre*.
On y sort l'objet de la carte puis on incrémente les points du joueur et on finit par supprimer l'objet de la mémoire via la méthode *rmObjet()*.

On a maintenant Roger qui peut se déplacer, être affiché et interagir avec les zombies et les déchets.
Il ne reste qu'à rendre le jeu plus réaliste en y ajoutant des obstacles.
Ci-dessous la carte de jeu.

On peut y voir de nombreux objets susceptibles de jouer le rôle d'obstacles tels que la voiture ou les troncs d'arbre.
Pour modéliser ces obstacles, la carte a été transformée en un tableau statique *tabObs[200][200]* de la classe Obstacle dont les cases ont été mises à 1 lorsque un obstacle s'y trouvait (case inaccessible) et à 0 sinon (case accessible). La figure ci-dessus nous permet de visualiser la fidélité de la reconstitution de la carte.

Une fois ces classes mises bout à bout dans le fichier *main.cpp*, on obtient un jeu parfaitement fonctionnel.

Fiertés:

Mener à bien un projet de la sorte est bien différent des séances de travaux pratiques que nous avons eu au cours de notre scolarité et parvenir à créer un jeu en partant de quasiment zéro et sans aucune aide extérieure a été pour nous quelque chose de très formateur.
Une fierté de ce projet a été de réussir à travailler en équipe, à se répartir les tâches en respectant les deadlines établies entre nous et à réussir à produire un code fonctionnel.
Gérer toute la partie graphique a été de loin la partie la plus satisfaisante pour nous. Les premiers pas de Roger à l'aide du pavé directionnel ont été pour nous un grand moment de joie car cela représentait pour nous la pose de la première brique de l'énorme chantier de programmation qu'a été ce projet. Se consacrer au dessin des sprites fut long mais finalement très gratifiant lorsque l'on lance le jeu et que l'on peut voir le magnifique rendu du décor réalisé par nos soins.
Malgré quelques regrets comme celui de ne pas avoir pu créer la classe *Spritable* ou la subsistance de certains bugs, nous garderons un très bon souvenir de ce projet.