M2103 : Rapport du projet de POO

Hex Light Bot

Cornuau Brice -- Lapicardise Romain
Groupe C''

Première année

Table des matières

1. Introduction

Ce rapport présente la réalisation d'un programme : le jeu Hex Light Bot.

Livrables
- ce rapport au format PDF
- le projet en lui même

Technologies
Ce programme a été réalisé en C++11, et utilise la bibliothèque SFML. Le développement a été effectué avec l'IDE Visual Studio puis porté sur QTCreator.

2. Le projet réalisé

2.1. Le Light Bot

Lightbot est un jeu vidéo éducatif pour apprendre le concept de la programmation, développé par Danny Yaroslavski. Lightbot a été joué plus de 7 millions de fois et est hautement noté sur iTunes et GooglePlay. Le jeu Lightbot est disponibles en jeu Flash en ligne et en application pour Androïd et iOS. Lightbot a été conçu en Flash et en OpenFL.

Dans ce jeu, les enfants apprennent les concepts de la programmation, ainsi ils apprennent à utiliser des conditions et des boucles sans taper un seul morceau de code. Ils acquièrent des compétences pour la résolution de problèmes algorithmiques plus ou moins complexes!
Le déroulement du jeu est simple; il faut donner une suite d'ordres (programme), à un robot et ce dernier va exécuter les actions demandées. Le but étant de lui faire allumer toutes les cases atteintes de la grille où il se situe. Plusieurs niveaux se succèdent avec pour chacun une grille avec des cases à allumer différente.

La version de ce projet est légèrement différente mais reprend les me^mes bases. Le robot est en faite une tondeuse, et il n'y a pas de cases à allumer mais de l'herbe à tondre.

2.2. Rôle du programme

Le programme présenté ici permet de

- donner des instructions(actions) à la tondeuse en les plaçant dans le bandeau de droite (dans le main, Proc1 et Proc2).
- lancer l'exécution du programme qui montre, étape par étape, la tondeuse exécuter les ordres.
- créer ou modifier un niveau en insérant et modifiant des cases et en choisissant les fonctions disponibles (mode édition)

Remarques :

- Il est possible de stopper le robot en pleine action, grâce au bouton "stop" et le robot revient ainsi à sa position d'origine.
- Il est possible de faire revenir le robot à sa position d'origine mais aussi supprimer toutes les actions positionnées dans le main, proc1 et proc2 grâce au bouton "reset".

2.3. Mode d'emploi

Le programme de "jeu" démarre avec un menu où il faut cliquer sur "Start Game" puis :

- pour ajouter / enlever des instructions (actions) dans le main ou proc1 ou proc2 : glisser - déposer.
- pour supprimer une action déjà placée : cliquer dessus.
- pour lancer l'animation ou en sortir : cliquer sur play ou stop.
- pour revenir à la position initiale du robot : cliquer sur reset.

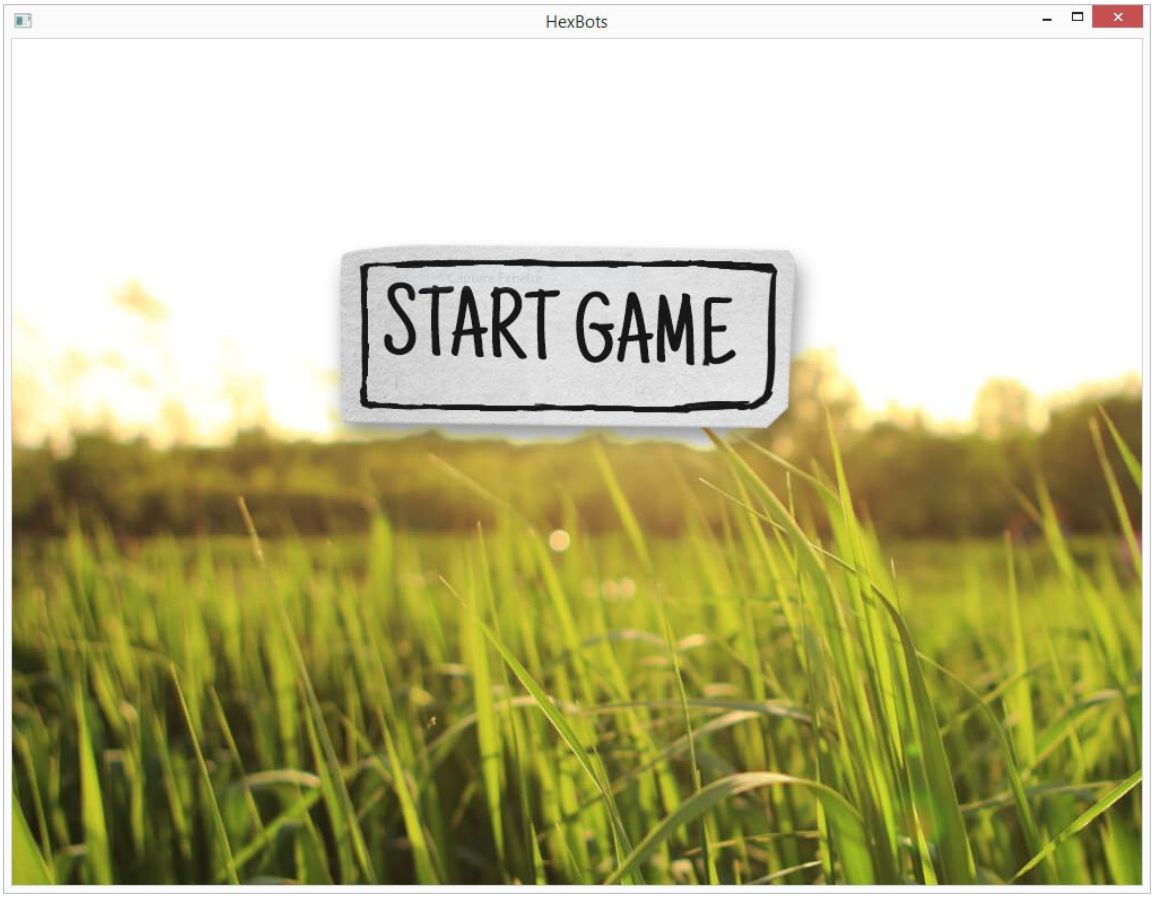
Le programme "d'édition" :

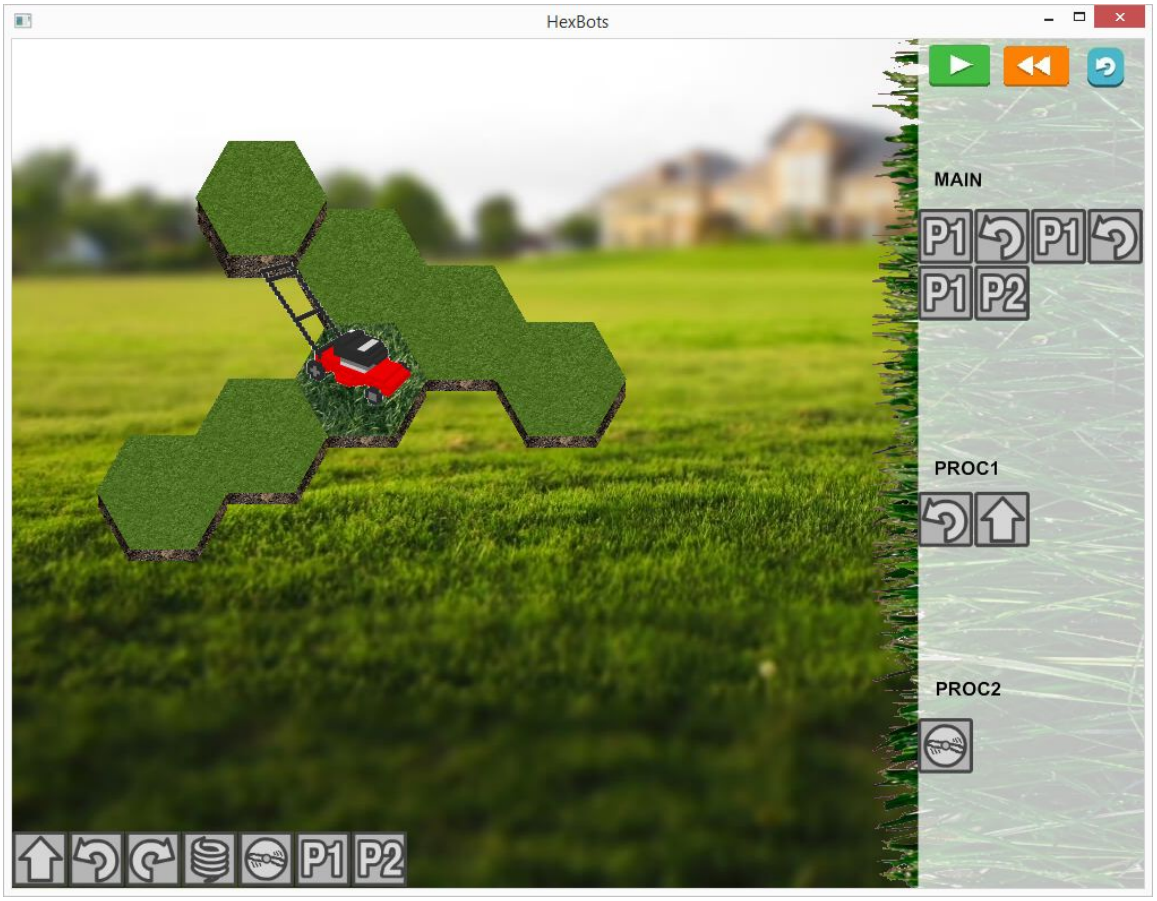
- pour ajouter / enlever des cases : cliquer dessus.
- pour ajouter / enlever des actions disponibles : cliquer dessus.

3. La programmation

Le programme est organisé de manière à ce que toutes les classes (assez peu nombreuses) gèrent les attributs qui lui sont propres.

- La classe Application lance le jeu et charge tous les sprites.
- La classe Case gère les cases hexagonales.
- Le classe Robot gère le robot et son comportement en fonction des instructions qui lui sont données.
- La classe Application
- La classe Niveau lit les fichiers où sont inscrits le contenus des niveaux, lance et affiche ces derniers, s'occupe de l'IHM, affiche les programmes créés et vérifie s'ils sont réussis pour passer au niveau suivant.
- Le programme main.cpp est minimaliste; il crée un objet Application et le lance.




HexBots
MAIN
P1
P1
P1
P2
PROC1
PROC2
P1
P2

3.1. Classe Application

3.1.1. Membres publics

Dans la classe Application il y a essentiellement:

Application() un constructeur;
~Application() un destructeur
void run() une fonction qui permet de lancer l'application.

3.1.2. Membres privés

void loop() est une boucle dans lequel on va faire appel à la fonction event_listenet() gère les évènements.
void init() charge les sprites.
void stop() qui est appelé pour fermer la fenêtre ou quand la fenêtre est fermée par l'utilisateur.
void display() qui permet d'afficher les informations sur la fenêtre
void event_listener() qui va permettre de lister les événements et d'y réagir.
void setMouseCoord(int x, int y) qui permet de récupérer les coordonnés de la souris sur la fenêtre pour la mettre sur les coordonnées des données.

void key_pressed(sf::Keyboard::Key code) qu'on fait appel quand une touche est appuyée et liste les évènement à faire dans les cas où on appuie sur une touche.
void mouse_pressed(sf::Mouse::Button) pour gérer les événements lorsque qu'on appuie sur un bouton de la souris.
void mouse_released(sf::Mouse::Button) quand on relache un bouton de la souris.
void mouse_moved() quand on bouge la souris.

Texture menu, new_game, jeu;
Sprite Menu, New_game, Jeu;

3.2. Classe case

3.2.1. Membres privés

Les membres privés de la classe Case sont :

- sf::Sprite grass_tondu,grass_pas_tondu,dirt_gauche,dirt_droite,dirt_bas;
- sf::Texture textureGrass;

- sf::Vector2f m_position permettant de stocker la position de la case.
- int m_hauteur = 0 qui permet de savoir la hauteur de la case.
- std::array<Case*, 6> m_voisines est un tableau de pointeurs de cases qui permet de pointer les voisines de la case
- enum Etat_Case{tondu, a_tondre} m_etat=a_tondre qui permet de savoir dans quel état est la case.

3.2.2. Membres publics

Les membres publics de la classe Case sont :

Case(int taille); un constructeur.
Case(sf::Vector2f position,int taille); un autre constructeur (avec la position de la case et sa taille).
~Case() un destructeur de case.
void init() qui permet d'initialiser les différentes textures et sprites.
void add_neighbour(int direction,Case* voisine) permet de remplir le tableau de m_voisine grace aux nouvelles cases quand on charge le fichier.
void paint() qui permet de passer la l'état pas tondu à tondu quand la tondeuse tond.
sf::Vector2f get_position();
bool have_neighbour(int direction) renvoie un booléen selon si il y a un voisin dans cette direction
bool have_neighbour(int direction, int taille) même chose mais en plus pour savoir si il existe un voisin de la même hauteur
Case* neighbour(int direction) renvoi un élément du tableau de m-voisine, renvoie en réalité le pointeur de la voisine.
void display(sf::RenderWindow &window, int taille) permet à la case de s'afficher sur le renderwindow à la taille voulue.

int high() const renvoie la hauteur de la case.
bool valid() renvoie si la case est tondue ou non.
float center_distance(sf::Vector2f point) renvoie un flottant qui correspond à la distance entre le centre de la case et la point qui a été donné en paramètre.
int direction(sf::Vector2f point) renvoie la direction dans laquelle se trouve le point qui a été mis en paramètre.

3.3. Classe Programme

3.3.1. Membres privés

Les membres privés de la classe Programme sont :

sf::Texture actions;
sf::Sprite avancer, tourner_droite, tourner_gauche, saut, tondre, Proc1, Proc2;
std::vector<int> actions_en_cours est un vecteur qui permet de connaître l'action qu'on est en train d'effectuer selon l'itération à laquelle on est. Les programme s'appelle les uns les autres grâce à P1 et P2. Pour que les programmes s'appellent ainsi, on a besoin d'un vecteur dont l'itérateur indique combien de fois on a appelé ce programme et ce vecteur va donc indiqué à quelle action on est à chaque fois qu'on l'a appellé.
int iteration=0 permet de savoir combien de fois on a appelé le programme et donc d'aller voir dans le vecteur actions_en_cours de façon correspondante.
Robot *m_robot est un pointeur vers le robot qui permet au programme d'envoyer directement au robot les actions qu'il doit effectuer.
std::vector<Action>m_actions stocke les actions stockées dans le programme.
int m_id correspond au numéro du programme (main → 0, P1-->1, P2-->2).

3.3.2. Membres publics

Les membres publics de la classe Programme sont :

Programme(int id,Robot *robot) un constructeur.
~Programme() un destructeur.
bool execute(Niveau * niveau) qui renvoie un booléen pour savoir si le programme est fini. Cela permet d'exécuter une action à chaque fois qu'on l'appelle. Elle prend en paramètre un pointeur de niveau pour qu'elle puisse demander au niveau de lancer un autre programme enregistré comme P1 ou P2.
void clear() qui permet de supprimer toutes les actions stockées dans le programme
void retry() qui ne vide pas le vecteur m_actions mais vide le vecteur d'action_en_cours pour qu'on puisse relancer le programme sans avoir à réécrire le programme.
void add_action(Action action,int position) indique à quel endroit on souhaite ajouter l'action dans le programme dans le vecteur m_actions. Gère aussi si la position est trop loin et si on peut insérer à un endroite. Décale aussi les actions si il y a besoin.
void display(sf::RenderWindow &window) qui permet au programme de s'afficher sur le RenderWindow donné en paramètre.

Action* erase_action(int position) supprime l'action présente et renvoie un pointeur vers elle pour qu'on puisse la déplacer dans le niveau.

3.4. Classe Niveau

3.4.1. Membres privés

Les membres publics de la classe Niveau sont :

bool souris_click= false permet de savoir si on a cliqué.
bool vic = false permet de savoir si on a fini un niveau grâce à un programme qui a gagné.
bool fini = false permet de savoir si on a fini d'exécuter le programme en cours et donc de la remettre à zéro.

sf::Texture actions, bar, victoire, fin, boutons;
sf::Sprite avancer, tourner_droite, tourner_gauche, saut, tondre, Proc1, Proc2, Selection, Bar, Victoire, Fin, Start, Replay, Reset, Stop;

std::vector<Case*>m_cases est un vecteur de pointeur de case qui permet de stocker la grille.
std::vector<Action>m_action_disponible stocke les actions disponibles pour faire le niveau.
std::array <Programme*,3> m_programmes est un tableau de pointeur de programme (main P1 et P2).

Robot *m_robot pointe vers le robot du niveau.
int m_direction_robot;
Case *m_position_initial;

void load_fichier(int level) charge le fichier d'un niveau.

Action *selection = nullptr est un pointeur d'actions qui correspond à l'action qu'on a sélectionné à la souris et qu'on est en train de déplacer.
int m_numero qui correspond au numéro du niveau actuel.
bool programme_en_cours qui permet de savoir si on est en train d'exécuter le programme.
void addCase(sf::Vector2f position, int taille, int etat) crée de nouvelles cases au chargement du fichier en connaissant la position la taille et l'état.

3.4.2. Membres publics

Les membres publics de la classe Niveau sont :

Niveau() un constructeur.
~Niveau() un destructeur.

void launch_level(int level) permet de lancer le niveau et donc de charger toutes les textures et sprites nécessaires.
bool launch_programme(int programme) lance une action du programme qui est mis en paramètre et retourne vrai si le programme a fini son exécution.
void click(sf::Vector2f position, sf::Mouse::Button code) est appellé par Application quand on a cliqué dans la fenêtre.
void move(sf::Vector2f position) donne la position de la souris quand cette dernière est en déplacement.
void drop(sf::Vector2f position) quand on relâche le bouton.
void key(sf::Keyboard::Key code) quand on appuie sur une touche.
void display(sf::RenderWindow &window) affiche le niveau et fait appelle au autre éléments qui s'affichent.

3.5. Classe Robot

3.5.1. Membres publics

Les membres publics de la classe Robot sont :

Robot() un constructeur.
~Robot() un destructeur.
void depart(Case * position, int direction) décrit la position et la direction de départ du robot.
void init() permet de charger toutes les textures et sprites nécessaires au robot.
void action(Action action) qui permet au programme de demander au robot d'exécuter tel ou tel action.
bool display(sf::RenderWindow &window) permet au robot de s'afficher sur la fenêtre.

3.5.2. Membres privés

Les membres publics de la classe Robot sont :

sf::Sprite spriteTondeuseArriere, spriteTondeuseAvant, spriteTondeuseAvantDroite, spriteTondeuseAvantGauche, spriteTondeuseArriereDroite, spriteTondeuseArriereGauche;
sf::Texture textureTondeuse;

sf::SoundBuffer buffer;
sf::Sound sound;

Case *m_position un pointeur de case pour que le robot sache sur quel case il est.
int m_direction=0 indique dans quelle direction le robot est (compris entre 0 et 6).
int etape qui permet de savoir à quelle étape de l'animation le =robot est de son action.
bool bouge pour savoir si le robot bouge ou non.
Action actionencours qui permet de savoir quelle est l'action en cours.

3.5.3. Autre

enum Action{forward,turn_left,turn_right,jump,paint,P1,P2};

3.6. Etat du projet

Le code source fourni fonctionne. Nombre de lignes :
Application.h 38
Application.cpp 152
Case.h 35
Case.cpp 224
main.cpp 9
Niveau.h 49
Niveau.cpp 743
Programme.h 34
Programme.cpp 201
Robot.h 34
Robot.cpp 488

3.7. Le code source

3.8. Licence

Ce logiciel est sous licence libre.

Une licence libre est une licence s'appliquant à une œuvre par laquelle l'auteur concède tout ou partie des droits que lui confère le droit d'auteur, en laissant au minimum les possibilités de modification, de rediffusion et de réutilisation de l'œuvre dans des œuvres dérivées. Ces libertés peuvent être soumises à conditions, notamment l'application systématique de la même licence aux copies de l'œuvre et aux œuvres dérivées, principe nommé copyleft.

Un logiciel sous licence libre permet toujours ces différentes possibilités:

La possibilité d'utiliser l'œuvre, pour tous les usages ;
La possibilité d'étudier l'œuvre ;
La possibilité de redistribuer des copies de l'œuvre ;
La possibilité de modifier l'œuvre et de publier ses modifications.

Les images d’herbes sont libres de droit, la tondeuse en 3D a été créé sur le site UseCubes (https://usecubes.com/explore/337966) et les images de boutons d’actions ont été inspirés du jeu original. La musique de fond est Faidherbe square (instrumental) du groupe toulousain Proleter et le son de la tondeuse a été téléchargé sur le site internet lasonotheque.org de Joseph Sardin.

3.9. Utilisation

Pour jouer à HexBot ou éditer un niveau, lancer un IDE, de préférence QTCreator ou Visual Studio (qui est l'outil sur lequel le jeu a été développé). Compilez et exécutez et puis c'est à vous de jouer.

3.10. Améliorations / Extensions possibles

- Réduire les bugs de l'éditeur.
- Améliorer les animations.
- Optimiser la réactivité du programme en diminuant les temps de chargement notamment.
- Ajouter un bouton appelant le programme "éditeur" dans le programme "principale" et inversement.