Rapport du Projet Bibliothèque de Développement Multimédia
The Endless Labyrinth

Page d'accueil de notre jeu - The Endless Labyrinth

Git : https://github.com/Richard-Drogo/Labyrinthe3D
Groupe : FISE 2 - TD C

Développeurs : **DROGO Richard** et **HOUDELET Lilian**
Graphiste : PARIZY Cédric
Sound Designer : LANCELOT Maxime

I - Présentation de l'interface utilisateur

Nous avons conçu notre jeu, de telle sorte qu'en plus de satisfaire le cahier des charges, il soit doté d'une interface plaisante à la vue, ergonomique et user-friendly.

L'utilisateur pourra naviguer à travers une interface composée de 4 fenêtres : fenêtre d'accueil ; fenêtre de paramétrage du labyrinthe ; fenêtre de calibrage ; fenêtre du labyrinthe.

Grâce aux fonctions de redimensionnements spécifiques que nous avons créées, il peut adapter la taille de la fenêtre de l'application à sa convenance sans que son contenu soit déformé.

La deuxième fenêtre de notre interface permet à l'utilisateur de configurer la taille du labyrinthe avec une taille de 10 * 6 par défaut sachant que chacune de ces valeurs peuvent être comprises dans l'intervalle [5;20]. Cette fenêtre affiche également les crédits, et le record de vitesse au cours de la session de jeu.

En appuyant sur le bouton jouer, il accède à la fenêtre de calibrage qui lui permet de définir en appuyant sur "Entrée" la position de sa tête avec laquelle il souhaite faire la partie (sachant qu'il pourra tout de même utiliser les touches au clavier "ZQSD"). Un message de confirmation est affiché pour s'assurer que le calibrage lui convient.

Après confirmation il accède à l'écran de jeu avec le Labyrinthe prenant la presque totalité de la fenêtre de l'application et la caméra et le chronomètre superposés à la fenêtre de jeu sans que ce cela ne devienne gênant. Pour ne pas surcharger l'interface. La carte 2D du labyrinthe lui est affichée directement sur la scène lorsqu'il est en position neutre.

Le joueur devra donc trouver la sortie le plus rapidement possible en ayant au préalable trouvé la sphère texturée avec le logo de TSE. Les murs, la porte et le plafond sont également texturés. Il disposera également d'une lampe torche en guise d'éclairage. Certains bugs sont tout de même présents (voir partie III).

II - Conception

1 - Présentation succincte des classes

A - Classes conçues

Classes graphiques

- Labyrinthe3D : Classe principale du projet permettant de naviguer à travers toutes les fenêtres de l'application. (QMainWindow)
- Labyrinthe : Classe permettant de représenter graphiquement le labyrinthe. (QOpenGLWidget)

Classes de données

- RGBColor : Classe permettant de regrouper toutes les informations de couleurs.
- GLColor : Identique à RGBColor mais retourne les informations de couleurs sous le format utilisé par OpenGL.
- Vertex : Permet de regrouper en une classe toutes les informations d'un sommet.
- Object3D : Classe mère des classes "Mur" et "Porte". Permet de créer un objet 3D simple comme un cube. Cette classe contient tous les attributs nécessaires à l'éclairage et à l'affichage de texture qui seront donc hérités par les classes filles.
- Mur : Classe fille d'Object3D et permet de créer un Mur du labyrinthe.
- Porte : Classe fille d'Object3D et permet de créer la porte du labyrinthe.
- Item : Classe permettant de créer la sphère texturée, clef du labyrinthe.
- Chronomètre : Classe modèle du chronomètre utilisé pour l'affichage du temps dans le labyrinthe.

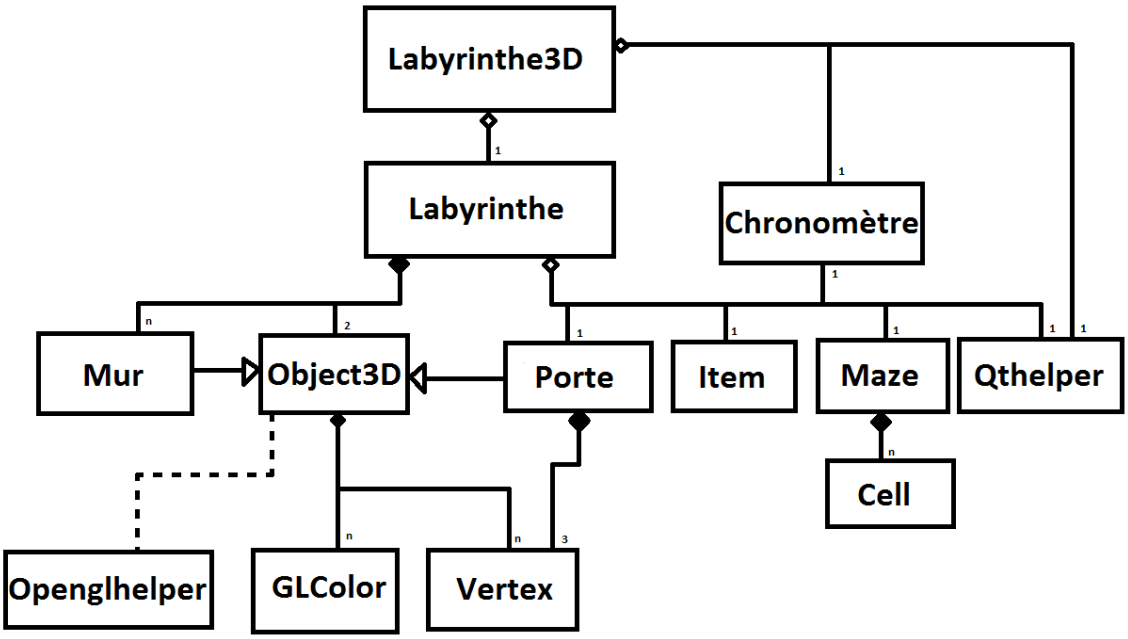
Classes d'aide

- OpenGLHelper : Classe d'aide permettant entre autres d'afficher un Object3D.
- QTHelper : Classe d'aide utilisée statiquement pour changer l'image de fond du QWidget passé en paramètre et utilisée en étant instanciée pour gérer la musique et les sons de l'application.

B - Classes fournies

- Cell : Classe utilisée dans Maze, pour la génération du labyrinthe
- Maze : Classe générant aléatoirement le labyrinthe et définit la position des différents éléments (position joueur, sphère, sortie)

2 - Diagramme de classe simplifié



3 - Explication des relations inter classes

Comme dit précédemment, la classe Labyrinthe3D est la classe permettant de gérer toute l'UI. Dès le début, elle crée dynamiquement un objet QTHelper pour jouer de la musique. Lorsque le joueur aura fini la configuration désirée du labyrinthe et le calibrage, elle crée dynamiquement les objets Chronomètre et Labyrinthe. L'objet de la classe Labyrinthe3D s'occupe de l'affichage des données de l'objet Chronomètre dans un QLabel.

Lors de la création de l'objet Labyrinthe, une référence vers le modèle Chronomètre lui est transmise et comme ce modèle contient une référence vers sa zone d'affichage, cela permet depuis la classe Labyrinthe de pouvoir démarrer le chronomètre et ainsi actualiser l'affichage grâce au QTimer présent dans l'objet Chronomètre. La référence à l'objet QTHelper est également transmise à l'objet Labyrinthe pour que ce dernier puisse changer la musique et jouer les bruits de pas.

Lors de sa construction, l'objet Labyrinthe s'occupe tout d'abord de créer dynamiquement l'objet Maze puisqu'il lui permet de pouvoir générer le Labyrinthe. Puis il s'occuper de la création du sol, du plafond, des murs qui sont ses composants. Puis il crée dynamiquement la Porte et la sphère.

III - État de finalisation

1 - Fonctionnalités du cahier des charges

1. Terminée : la scène 3D du Labyrinthe est affichée dans la 4ème fenêtre de l'application et généré aléatoirement grâce aux classes données sur Mootse. Sa taille est déterminée selon la configuration indiquée dans la 2ème fenêtre de l'application. La taille par défaut est 10 par 6 et chaque valeur peut varier dans l'intervalle [5;20].

PS : Lors de la présentation nous avons présenté des petits labyrinthes (5 par 5 et 5 par 7) car le jeu devenait très lent lorsque nous mettions des tailles plus importantes. Ce problème fut réglé le soir même (comme peut en témoigner la version sur notre git) et était dû aux textures qui étaient chargées de manières continuelles. (Un changement de position de certaines lignes de codes a permis de régler le problème).

2. Terminée et Adaptée : Pour coller à l'ambiance de notre jeu (ambiance assez sombre) nous avons décidé de remplacer une lumière ambiante par une lumière directive (pour donner l'impression que le joueur dispose d'une lampe torche et explore le labyrinthe).

PS : Quelques bogues d'éclairage sont présents sur le plafond et le sol et sont dus à la non-définition des normales des primitives par manque de temps.

3. Terminée : Lorsque le joueur commence une partie, ce que filme sa caméra est retransmis dans le coin supérieur droit avec la zone de détection en superposition
   Lorsque le joueur fait un mouvement de la tête, en fonction de ce mouvement, une action est choisie.
   S'il se heurte à un mur (cela peut arriver), il sera dans l'incapacité de le traverser

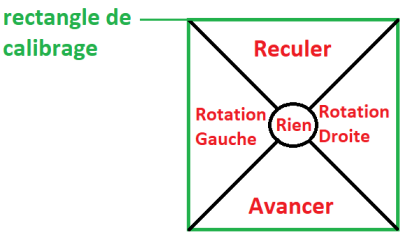
PS : Dans certains cas, la détection peut avoir quelques problèmes, en effet des points de lumière assez fort peuvent perturber cette détection.

4. Terminée : Position neutre quand : (aucune entrée clavier pour les déplacements avec "ZQSD") et (la tête reste dans la position neutre selon un seuil).
5. Terminée : Lorsque le joueur est dans une position neutre, la carte du labyrinthe est dessinée sur la fenêtre de jeu. (le joueur et la direction regardée sont représentées en vert, les murs en blancs et la porte en bleu. La sphère n'est pas représentée pour ne pas rendre le jeu trop simple.)
6. Terminée : Une sphère est générée à une position définie à la génération du labyrinthe. Elle disparaît quand le joueur l'atteint et permet d'ouvrir la porte.
7. Terminée : Inspiré du "design pattern MVC", un modèle de chronomètre a été créé et lié à son affichage dans un QLabel superposé à l'écran de jeu en bas à droite.
8. Terminée : Avec les coordonnées de la sortie, une zone du labyrinthe est définie comme étant à atteindre pour sortir du labyrinthe, le labyrinthe est détruit puis le message indiquant le temps apparaît et le jeu propose de commencer une nouvelle partie, puis renvoie vers le menu principal.

2 - Fonctionnalités “Bonus”

Nous n’avons pas eu le temps de faire des fonctionnalités comme celles indiquées dans les consignes bien que pour les indices, nous avions prévus comme suggéré par la deuxième image du menu d’accueil de donner la possibilité au joueur de changer de mode d’éclairage permettant de rendre visible les flèches bleues. Néanmoins, nous avons pu implémenter les fonctionnalités suivantes :

- Ajout de 2 musiques, une pour l’interface et une pour le labyrinthe avec en plus un son représentant les déplacements du joueur lorsque ce dernier se déplace.
- Contrôle de la saisie des paramètres du labyrinthe pour avoir des valeurs dans l’intervalle [5;20].
- Possibilité de se déplacer au clavier avec les touches “ZQSD”.
- Ajout de textures 2D pour les murs, le plafond et la porte.
- Tous les objets visibles possèdent une certaine épaisseur.
- Un système de collision empêchant de passer à travers les murs et la porte.
- Interface Graphique Design avec redimensionnement automatique si changement de la taille de la fenêtre.
- Fenêtre dédiée au calibrage de la détection des mouvements.
- Amélioration des mouvements à la caméra (voir figure).



- Le joueur peut voir son record de rapidité (commun à toutes les tailles de labyrinthe) sur la fenêtre de configuration du labyrinthe (A chaque fois qu’il battra son record il sera mis-à-jour).
- Ergonomie de jeu : Le labyrinthe 3D, la carte 2D, la caméra et le chronomètre sont superposées sur la fenêtre de jeu.
- La programmation de la carte 2D est faite de telle sorte qu’elle occupe un espace fixe quelle que soit la taille de labyrinthe configurée (elle ne masquera jamais la caméra ou le chronomètre).
- La sortie est générée dès la création du labyrinthe, de ce fait des fonctions permettent de mettre la sphère suffisamment loin de la sortie et ainsi mettre le joueur loin de la sphère.

PS : Dans le cas de labyrinthe trop grand, la méthode met plus de temps à converger vers une solution, ce qui peut être très long. Critère corrigé à posteriori.

- Complexification des murs : Un mur n’est pas un gros cube occupant tout l’espace de sa case, mais les murs ont une certaine épaisseur et en fonction de leur position ils ne sont pas composés de la même manière : Un mur d’angle est composé de deux pavés; les murs du contours ne sont composés que d’un seul pavés… Ce qui donne beaucoup plus d’espace pour se déplacer.

IV - En-têtes des classes (sauf la classe Cell non modifiée)

```cpp
// Auteur : Richard

// Includes et autres instructions omises pour la lisibilité...

/* Classe Labyrinthe3D
Rôle : Classe d'entrée du programme et permettant de gérer la navigation entre toutes les UI.
*/
class Labyrinthe3D : public QMainWindow
{
    Q_OBJECT

public:
    /* Labyrinthe3D(QWidget *parent = nullptr)
    Rôle : Constructeur de la classe Labyrinthe3D
    Entrée : QWidget * qui sera égal à nullptr étant que c'est le point d'entrée graphique du programme.
    */
    Labyrinthe3D(QWidget *parent = nullptr);
    /* ~Labyrinthe3D()
    Rôle : Destructeur de la classe Labyrinthe3D
    */
    ~Labyrinthe3D();

    // Partie des constantes publiques omise pour la lisibilité...

private slots:
    /* on_pushButton_jouer_clicked()
    Rôle : Permet de mettre en place l'UI de configuration du labyrinthe.
    */
    void on_pushButton_jouer_clicked();

    /* on_pushButton_retour_accueil_clicked()
    Rôle : Permet de reset l'UI de configuration du labyrinthe et mettre en place l'UI d'accueil.
    */
    void on_pushButton_retour_accueil_clicked();

    /* updateVideo()
    Rôle : Appelée en réponse au SLOT(timeout()) du timer_video, elle permet de mettre-à-jouer l'image de la caméra.
    */
    void updateVideo();

    /* finAnimationAccueil()
    Rôle : Appelée en réponse au SLOT(timeout()) du timer_accueil_ et réalise l'animation de l'accueil.
    */
    void finAnimationAccueil();

    /* on_lineEdit_largeur_editingFinished()
    Rôle : Permet de vérifier la bonne saisie d'une valeur de longueur pour le labyrinthe.
    */
    void on_lineEdit_largeur_editingFinished();

    /* on_lineEdit_largeur_editingFinished()
    Rôle : Permet de vérifier la bonne saisie d'une valeur de largeur pour le labyrinthe.
    */
    void on_lineEdit_longueur_editingFinished();

private:
    Ui::Labyrinthe3D *ui; // Permet d'accéder aux éléments conçus via QT UI Designer.

    QTHelper * qthelper_ = Q_NULLPTR; // Objet QTHelper pour la gestion de la musique.
    QTimer * timer_video = nullptr; // Timer permettant de m-à-j les images de la webcam grâce au SLOT(timeout()) qui appelle le SIGNAL(updateVideo()).
    QTimer * timer_accueil_ = Q_NULLPTR; // Timer permettant de temporiser avant de passer à l'UI de paramétrage. SIGNAL(finAnimationAccueil()).
    QMediaPlaylist * playlist_ = Q_NULLPTR; // Non utilisé (oubli de déletion)
    QMediaPlayer * player_ = Q_NULLPTR; // Non utilisé (oubli de déletion)
    quint32 record_ = 0; // Valeur du record en secondes et permet de comparer avec une nouvelle valeur.
    QString recordS_ = ""; // String représentant le record actuel.

    // Début : Attributs permettant d'indiquer si l'UI n'a jamais été configurée.
    bool isUIParametrageLabyrintheInitialized_ = false;
    bool isUICalibrageInitialized_ = false;
    bool isUILabyrintheInitialized_ = false;
    // Fin : Attributs permettant d'indiquer si l'UI n'a jamais été configurée.

    bool partie_en_cours_ = false; // Non utilisé (oubli de déletion)
    VideoCapture * webcam = nullptr; // La webcam
    Chronometre * chronometre_ = Q_NULLPTR; // Modèle de chronomètre.

    // Début : Attributs nécessaires pour la partie détection de visage
    CascadeClassifier CascadeClassifier_visages;
    Rect calibrageRect;
    Rect templateRect;
    cv::Point workingCenter;
    Mat frameReference, resultImage;
    // Fin : Attributs nécessaires pour la partie détection de visage.

    // Début : UI Labyrinthe
    QLabel * label_video_labyrinthe_ = Q_NULLPTR; // QLabel où est affiché la caméra durant le labyrinthe.
    QLabel * label_chronometre_ = Q_NULLPTR; // QLabe où est affiché le chronomètre durant le labyrinthe.
    Labyrinthe * labyrinthe_ = Q_NULLPTR; // Objet Labyrinthe créé dynamiquement.
    // Fin : UI Labyrinthe

    // Partie des constantes privées omise pour la lisibilité...

    /* retournerIndiceDuRectanglePlusGrand(std::vector<Rect> rectangles)
    Rôle : Permet de retourner l'indice du plus grand rectangle d'un vector<Rect> passé en paramètre.
           Utile pour la reconnaissance de visages dans la vidéo. (On veut un seul visage !)
    Entrée : vector<Rect>, vecteur contenant tous les rectangles correspondants aux visages détectés.
    Sortie : L'indice du rectangle le plus grand dans la liste
    */
    unsigned long long retournerIndiceDuRectanglePlusGrand(std::vector<Rect> rectangles);

    /* verifierConfiguration()
    Rôle : Effectue d'autres vérifications sur les saisies des paramètres du labyrinthe.
    */
    void verifierConfiguration();

    /* qint8 getActionCamera(cv::Point vecteur_translation, double norme)
    Rôle : Permet de déterminer l'action voulue par le joueur via la caméra. qint8 est une constante d'action définie dans Labyrinthe.h
    Entrées :   * cv::Point vecteur_translation : Vecteur de translation déterminé
                * double norme : Norme du vecteur translation (Si la norme n'est pas suffisante => aucune action)
    Sortie : qint8 correspondant à une constante d'action définie dans Labyrinthe.h
    */
    qint8 getActionCamera(cv::Point vecteur_translation, double norme);

    /* demarrerVideo()
    Rôle : Après avoir cliqué sur le bouton "Jouer" permet d'initialiser les variables pour le traitement d'image et lancer la vidéo.
    */
    void demarrerVideo();


    // Début : Méthodes permettant de configurer la première fois les widgets et composants de l'UI en question
    void initialiserUIParametrageLabyrinthe();

    void initialiserUICalibrage();

    void initialiserUILabyrinthe();
    // Fin : Méthodes permettant de configurer la première fois les widgets et composants de l'UI en question


    // Début : Méthodes permettant de mettre en place les UI en question
    void setupUIAccueil();

    void setupUIParametrageLabyrinthe();

    void setupUICalibrage();

    void setupUILabyrinthe();
    // Fin : Méthodes permettant de mettre en place les UI en question


    // Début : Méthodes permettant de redimensionner tous les composants de l'UI en question
    void resizeUIParametrageLabyrinthe(QResizeEvent* event = Q_NULLPTR);

    void resizeUICalibrage(QResizeEvent* event = Q_NULLPTR);

    void resizeUILabyrinthe(QResizeEvent* event = Q_NULLPTR);
    // Fin : Méthodes permettant de redimensionner tous les composants de l'UI en question


    // Début : Méthodes permettant de réinitialiser les UI en question (quand on quitte une UI)
    void razUIAccueil();

    void razUICalibrage();

    void razUILabyrinthe();
    // Fin : Méthodes permettant de réinitialiser les UI en question (quand on quitte une UI)

private slots:
    /* partieTerminee()
    Informations : SIGNAL appelé par la classe Labyrinthe lorsque le joueur sort du labyrinthe.
    Rôle : Permet de mettre à jour le QLabel record, de rénitialiser l'UI Labyrinthe et de mettre en place l'UI de paramétrage.
    */
    void partieTerminee();

protected:
    void keyPressEvent(QKeyEvent * event); // Méthode étant appelée à chaque fois que l'utilisateur appuye sur une touche.

    void resizeEvent(QResizeEvent* event); // Méthode étant appelée à chaque fois que l'utilisateur redimensionne la fenêtre principale

};
```

```cpp
// Auteurs : Lilian et Richard

// Includes et autres instructions omises pour la lisibilité...

/* Classe Labyrinthe
Rôle : Cette classe est le QOpenGLWidget permettant l'affichage du Labyrinthe en 3D et de la carte en 2D. La génération informatique du labyrinthe est effec
*/
class Labyrinthe : public QOpenGLWidget
{
    Q_OBJECT

public:
    /* Labyrinthe(QWidget * parent, QTHelper * qthelper, qint8 longueur, qint8 largeur, Chronometre * chronometre);
    Rôle : Constructeur de la classe.
    Entrées :
                - QWidget * parent : Représentant un object Labyrinthe3D.
                - QTHelper * qthelper : Utilisé dans cette classe pour géré la musique et les sons.
                - quint8 longueur : Longueur du labyrinthe paramétrée [5;20] (Ce n'est donc pas la longueur effective)
                - quint8 largeur : Largeur du labyrinthe paramétrée [5;20] (Ce n'est donc pas la largeur effective)
                - Chronometre * chronometre : Pointeur vers le modèle chronomètre. (Utilisé pour le démarrer)
    */
    Labyrinthe(QWidget * parent, QTHelper * qthelper, qint8 longueur, qint8 largeur, Chronometre * chronometre);

    /* ~Labyrinthe()
    Rôle : Destructeur de la classe.
    */
    ~Labyrinthe();

    /* void actionCamera(qint8 action)
    Rôle : Cette méthode est appelée par le parent Labyrinthe3D dans la méthode updateVideo(). A chaque fois qu'une nouvelle acquisition est faite. L'actior
    Entrée : qint8 action : L'action déterminée par la méthode "getAction() de Labyrinthe3D" et étant une des valeurs ci-dessous.
    */
    void actionCamera(qint8 action);

    // Début : Constantes utilisées par la Caméra
    static const qint8 SEUIL_NORME_ACTION_CAMERA = 30; // DOIT TOUJOURS ÊTRE != 0
    static const qint8 ACTION_CAMERA_AUCUNE = 0;
    static const qint8 ACTION_CAMERA_AVANCER = 1;
    static const qint8 ACTION_CAMERA_TOURNER_CAMERA_A_GAUCHE = 2;
    static const qint8 ACTION_CAMERA_RECULER = 3;
    static const qint8 ACTION_CAMERA_TOURNER_CAMERA_A_DROITE = 4;
    // Fin : Constantes utilisées par la Caméra

    // Début : Constantes dimensionnelles publiques
    static constexpr double LONGUEUR_CASE = 2;
    // Fin : Constantes dimensionnelles publiques

signals:
    void sortieAtteinte(); // Non utilisé (oubli de déletion)

private:
    // Début : Attributs
    // Début : Attributs du Constructeur
    QWidget * parent_;
    QTHelper * qthelper_; // Classe QTHelper permettant de gérer la musique et les sons.
    double longueur_; // Longueur indiquée sur l'UI.
    double largeur_; // Largeur indiquée sur l'UI.
    Chronometre * chronometre_ = Q_NULLPTR; // Modèle Chronomètre.
    GLuint* texturesId; // Textures
    // Fin : Attributs du Constructeurs

    // Début : Attributs de la physique
    Vertex positionJoueur_;
    Vertex direction_;
    double angle_direction_ = -1; // Angle compris entre 0 et 360 durant l'exécution du jeu.
    qint8 action_camera_actuelle_ = ACTION_CAMERA_AUCUNE;
    bool mode_indice_active_ = false; // Non utilisé (oubli de déletion)
    // Fin : Attributs de la physique

    // Début : Objets 3D dessinés
    Object3D sol_;
    Object3D plafond_;
    QVector<Mur> murs_;
    Item* item_; // Sphère texturée avec le logo de TSE (clef pour ouvrir la porte)
    Porte* porte_ = nullptr;
    // Fin : Objets 3D dessinés

    // Début : Information sphère
    double itemPosX_;
    double itemPosY_;
    bool itemGet_ = false;
    const QString logo_ = ":/textures/Ressources/Textures/logo_telecom.png";
    // Fin : Information sphère

    // Début : Information sortie
    double exitPosX_;
    double exitPosY_;
    bool exitReached_ = false;
    // Fin : Information sphère


    // Début : Attributs du labyrinthe
    QVector<QVector<qint8>> matrice_labyrinthe_; // Matrice retournée par la fonction spécifique créée dans la classe Maze.
    Maze * maze_ = nullptr; // Instance de Maze générateur de labyrinthe.
    QTimer * timer_carte_du_labyrinthe_ = Q_NULLPTR; // Timer permettant la temporisation avant l'affichage de la carte 2D.
    bool afficher_carte_= false; // Utilisé dans le paintGL permet d'indiquer d'afficher la carte.
    // Fin : Attributs du labyrinthe
    // Fin : Attributs


    // Constantes privées omises pour la lisibilité ...


    // Début : Méthodes privées
    void display(); // Permet d'afficher dans le paintGL le labyrinthe
    void genererMur(); // Boucle permettant de créer les murs.
    void genererPorte(); // Permet de créer la porte.
    void definirTypeMur(qint8 x, qint8 y); // Création d'un mur correspondant à la case localisé en x et en y.
    qint8 compterCombienDeCasesNonDefinies(qint8 x, qint8 y); // Permet de compter le nombre de cases non définies autour du mur en x et y.
    void avancer(); // Effectue les changements de position pour avancer si aucune collision.
    void reculer(); // Effectue les changements de position pour reculer si aucune collision.
    void tournerCameraAGauche();
    void tournerCameraADroite();
    void dessinerCarteLabyrinthe(QPainter & painter);
    void arreterTimerCarteDuLabyrinthe(); // Lors d'un déplacement, permet d'arrêter l'affichage de la carte.
    // Fin : Méthodes privées


private slots:
    void timerCarteDuLabyrintheFini(); // Lors d'une position neutre, timer introduisant une temporisation avant affichage.

protected:
    void initializeGL(); // Initialisation de la scène OpenGL
    void resizeGL(int width, int height); // Méthode appelée lors du redimensionnement de la fenêtre
    void paintGL(); // Methode permettant de dessiner la scène.
    void keyPressEvent(QKeyEvent * event); // Appelée lorsque le joueur appuye sur une touche (déplacements)
    void keyReleaseEvent(QKeyEvent * event); // Appelée lorsque le joueuru relâche une touche.
    void touchTheBall(); // Permet de vérifier si le joueur est en contact avec la sphère.
    bool touchTheWall(double X, double Y); // Permet de vérifier si le joueur entre en collision avec le mur ou la porte.
    void ReachExit(); // Permet d'indiquer au programme que le joueur a atteint la sortie.
};
#endif // LABYRINTHE_H
```

effectuée dedans

L'action désirée est déterminée (Voir valeurs ci-dessous) puis passée paramètre. Cette méthode s'occupe ensuite de commander la réalisation des actions.

```cpp
// Auteur : Richard

#ifndef RGBCOLOR_H
#define RGBCOLOR_H

#include <GL/glu.h>


class RGBColor
{
public:
    /* RGBColor(GLclampf red_color_, GLclampf green_color_, GLclampf blue_color_)
    Rôle : Constructeur permettant de définir la couleur selon les 3 valeurs RGB.
    */
    RGBColor(GLclampf red_color_, GLclampf green_color_, GLclampf blue_color_);

    // Début : Getters
    unsigned short int getRed() {return red_;};
    unsigned short int getGreen() {return green_;};
    unsigned short int getBlue() {return blue_;};
    // Fin : Getters

    // Quelques couleurs prédéfinies.
    static const RGBColor red_color_, green_color_, blue_color_, yellow_color_, pink_color_, cyan_color_, black_color_, white_color_, silver_col

private:
    unsigned short int red_;
    unsigned short int green_;
    unsigned short int blue_;
};

#endif // RGBCOLOR_H
```

```cpp
// Auteur : Richard

#ifndef GLCOLOR_H
#define GLCOLOR_H

#include <GL/glu.h>

#include <rgbcolor.h>

/* Classe GLColor
Rôle : Classe permettant de regrouper les informations de couleurs.
*/
class GLColor
{
public:
    /* GLColor(GLclampf red, GLclampf green, GLclampf blue)
    Rôle : Constructeur permettant de définir la couleur selon les trois canaux RGB
    */
    GLColor(GLclampf red, GLclampf green, GLclampf blue);

    /* GLColor(RGBColor color)
    Rôle : Constructeur selon un RGBColor
    */
    GLColor(RGBColor color);
    GLColor();

    // Début : Getters
    GLclampf getRed() const {return red_;};
    GLclampf getGreen() const {return green_;};
    GLclampf getBlue() const {return blue_;};
    // Fin : Getters
private:
    // champs pour les couleurs
    GLclampf red_;
    GLclampf green_;
    GLclampf blue_;
};
#endif // GLCOLOR_H
```

```cpp
// Auteur : Richard

#ifndef VERTEX_H
#define VERTEX_H

/* Classe Vertex
Rôle : Classe permettant de regrouper les coordonnées.
*/
class Vertex
{
public:
    Vertex();
    Vertex(float x, float y, float z);

    // GETTERS
    float getX(){return x_;};
    float getY(){return y_;};
    float getZ(){return z_;};

    // SETTERS
    void setX(float x){x_ = x;};
    void setY(float y){y_ = y;};
    void setZ(float z){z_ = z;};

    /* Vertex xContrary()
    Rôle : NON UTILISE Mais permet de retourner les vertex symétriques à l'axe Y.
    */
    Vertex xContrary(){return Vertex(-x_,y_,z_);};

private:
    float x_;
    float y_;
    float z_;
};

#endif // VERTEX_H
```

```cpp
// Auteurs : Lilian et Richard

#ifndef OBJECT3D_H
#define OBJECT3D_H

#include <vector>
#include <QString>
#include <QVector>
#include <QImage>

#include "vertex.h"
#include "glcolor.h"

using namespace std;

class Object3D
{
public:
    // Constructeurs
    Object3D();
    /* Object3D(QString name, QVector<QVector<Vertex>> vertices, QVector<GLColor> colors, GLfloat brillance = 0, const QImage * image = Q_NULL
    Rôle : Construire un objet en 3D selon les paramètres
    Entrées :   * QString name : le nom de notre objet (utile en débuggage)
                * QVector<QVector<Vertex>> vertices : Les vertex de notre objet
                * QVector<GLColor> colors : Les couleurs de chaque sommet ou chaque face.
                * GLfloat brillance : Brillance pour l'éclairage (les autres paramètres d'éclairage sont définies avec la couleur)
                * const QImage * image : Texture
    */
    Object3D(QString name, QVector<QVector<Vertex>> vertices, QVector<GLColor> colors, GLfloat brillance = 0, const QImage * image = Q_NULLPTR)
    virtual qint8 display(); // Affichage de l'objet 3D

    // Getters
    QVector<QVector<Vertex>> getVertices(){return vertices_;};
    QVector<GLColor> getColors() {return colors_;};

    // Setters
    void setName(QString name) {name_ = name;};
    void setVertices(QVector<QVector<Vertex>> vertices) {vertices_ = vertices;};
    void setColors(QVector<GLColor> colors) {colors_ = colors;};

protected:
    QVector<QVector<Vertex>> vertices_;
    QVector<Vertex> normales_; // Normales pour l'éclairage
    QVector<GLColor> colors_;
    QString name_;

    // Début : Composantes pour la définition du matériau face à l'éclairage.
    QVector<GLfloat> couleur_ambiente_;
    QVector<GLfloat> couleur_diffuse_;
    QVector<GLfloat> couleur_speculaire_;
    QVector<GLfloat> couleur_emission_;
    GLfloat brillance_ = 0;
    // Fin : Composantes pour la définition du matériau face à l'éclairage.

    // Début : Gestion des Textures
    QImage const* image_ = Q_NULLPTR;
    GLuint texture_;
    // Fin : Gestion des Textures


};
#endif // OBJECT3D_H
```

```cpp
// Lilian et Richard

#ifndef MUR_H
#define MUR_H

#include <QPainter>

#include "object3d.h"

class Mur : public Object3D
{
public:
    /* Mur(double x, double y, qint8 type, qint8 orientation, double epaisseur, double hauteur, double longueur,
                                    QVector<GLColor> colors, GLfloat brillance = 0, const QImage * image = Q_NULLPTR)
    Rôle : Construit un mur selon les paramètre indiqués :
    Entrées :   * double x, y : Coordonnées graphiques.
                * qint8 type : Type de mur (ANGLE, CONTOUR_T1...),
                * qint8 orientation : NE,NW...
                * double epaisseur, hauteur, longueur : dimensions physiques
                * QVector<GLColor> colors : Couleur du matériau
                * GLfloat brillance : Brillance sous éclairage.
                * const QImage * image : Texture.
    */
    Mur(double x, double y, qint8 type, qint8 orientation, double epaisseur, double hauteur, double longueur, QVector<GLColor> colors, GLfloat
    qint8 display() override; // Affichage dans la scène du mur.

    /* draw(QPainter & painter, qreal longueur_case_carte, qreal largeur_case_carte)
    Rôle : Permet de dessiner ce mur sur la carte avec le painter et les longueurs et large d'une case sur la carte.
    */
    void draw(QPainter & painter, qreal longueur_case_carte, qreal largeur_case_carte);

    // Getters
    double getX() {return x_;};
    double getY() {return y_;};

    static const qint8 ANGLE = 0;

    static const qint8 CONTOUR_T1 = 1;
    static const qint8 CONTOUR_T2 = 2;

    static const qint8 CENTRE_T1 = 3;
    static const qint8 CENTRE_T2 = 4;
    static const qint8 CENTRE_T3 = 5;
    static const qint8 CENTRE_T4 = 6;
    static const qint8 CENTRE_T5 = 7;

    static const qint8 NE = 0;
    static const qint8 NW = 1;
    static const qint8 SW = 2;
    static const qint8 SE = 3;

    static const qint8 N = 0;
    static const qint8 W = 1;
    static const qint8 S = 2;
    static const qint8 E = 3;

    static const qint8 H = 0;
    static const qint8 V = 1;

    static const qint8 AUCUNE_ORIENTATION = -1;
private:
    double x_;
    double y_;
    qint8 type_;
    qint8 orientation_;
    double epaisseur_;
    double hauteur_;
    double longueur_;

    /* createVertices()
    Rôle : Méthode privée permettant de créer tous les Vertex relatifs au type de mur et à son orientation.
    */
    void createVertices();
};

#endif // MUR_H
```

```cpp
// Auteurs : Lilian et Richard

#ifndef PORTE_H
#define PORTE_H

#include "object3d.h"
#include "mur.h"

class Porte : public Object3D
{
public:
    // Début : Méthodes publiques
    /* Porte(double x, double y, qint8 type, qint8 orientation, double epaisseur, double hauteur, double longueur,
                                    QVector<GLColor> colors, GLfloat brillance = 0, const QImage * image = Q_NULLPTR)
    Rôle : Construit un mur selon les paramètre indiqués :
    Entrées :   * double x, y : Coordonnées graphiques.
                * qint8 type : Type de mur (ANGLE, CONTOUR_T1...),
                * qint8 orientation : NE,NW...
                * double epaisseur, hauteur, longueur : dimensions physiques
                * QVector<GLColor> couleurs_porte : Couleur du matériau
                * GLfloat brillance : Brillance sous éclairage.
                * const QImage * image : Texture.
    */
    Porte(double x, double y, qint8 position, double epaisseur, double hauteur, double longueur, QVector<GLColor> couleurs_porte, GLfloat bril
    qint8 display(); // Affiche la porte
    void ouvrir(); // Non utilisé (non fonctionnel)

    /* draw(QPainter & painter, qreal longueur_case_carte, qreal largeur_case_carte)
    Rôle : Permet de dessiner la porte sur la carte avec le painter et les longueurs et large d'une case sur la carte.
    */
    void draw(QPainter & painter, qreal longueur_case_carte, qreal largeur_case_carte);
    // Fin : Méthodes publiques


    // Début : Constantes des positions possibles
    static const qint8 N = 0;
    static const qint8 W = 1;
    static const qint8 S = 2;
    static const qint8 E = 3;
    // Fin : Constantes des positions possibles

    // Début : GETTERS
    bool isOuverte() {return ouverte_;}; // Non utilisée
    // FIN : GETTERS
private:
    // Début : Attributs du Constructeur
    double x_;
    double y_;
    qint8 position_;
    double epaisseur_;
    double hauteur_;
    double longueur_;
    // Fin : Attributs du Constructeur


    // Début : Attributs de physique (Non utilisés)
    double angle_NE = 0;
    double angle_SW = -90;
    double angle_SE;
    bool ouverte_ = false;
    Vertex centre_NE;
    Vertex centre_SW;
    Vertex centre_SE;
    // Fin : Attributs de physique


    // Début : Constantes de physique (Non utilisés)
    const double DEPLACEMENT_ANGULAIRE = 1;
    const qint8 LIMITE_OUVERTURE_ANGULAIRE = 90;
    // Fin : Constantes de physique


    // Début : Méthodes privées
    /* createVertices()
    Rôle : Méthode privée permettant de créer tous les Vertex relatifs de la porte.
    */
    void createVertices();
    // Fin : Méthodes privées
};
#endif // PORTE_H
```

```cpp
// Auteur : Lilian

#ifndef ITEM_H
#define ITEM_H

#include <qopengl.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <QColor>

/* Classe Item
Rôle : Classe permettant la gestion des objets dans le labyrinthe sous forme de quadrique (sphère). Utilisé pour la sphère avec le logo de TSE.
*/
class Item
{
private :
    unsigned int colorR; // Couleur du matériau Canal Rouge
    unsigned int colorG; // Couleur du matériau Canal Vert
    unsigned int colorB; // Couleur du matériau Canal Bleu
    float posX; // Position X de la sphère
    float posY; // Position Z de la sphère
    float rayon; // Rayon de la sphère
    GLUquadric* sphere; // Quadrique

public:
    /* Item(float posX, float rayon, float posY, unsigned int colorR, unsigned int colorG,  unsigned int colorB)
    Rôle : Permet de construire la sphère selon sa position X, Z, son rayon et sa couleur.
    */
    Item(float posX, float rayon, float posY, unsigned int colorR, unsigned int colorG,  unsigned int colorB);//, float revolution);

    // Destructeur
    virtual ~Item();

    // Methode d'affichage de la sphère.
    void Display() const;

};
#endif // ITEM_H
```

```cpp
// Auteur : Richard

#ifndef CHRONOMETRE_H
#define CHRONOMETRE_H

#include <QTimer>
#include <QObject>
#include <QLabel>

/* Classe Chronometre
Rôle : Cette classe est le modèle de chronomètre utilisé dans le Labyrinthe.
*/
class Chronometre : public QObject
{
    Q_OBJECT

public:
    /* Chronometre(QLabel * label_chronometre)
    Rôle : Constructeur de la classe Chronomètre (Modèle)
    Entrée : QLabel *, pointeur vers le QLabel ou sera affiché le temps écoulé.
    */
    Chronometre(QLabel * label_chronometre);

    // Début : Méthodes publiques
    void start() {timer_->start(INTERVALLE);};
    void stop() {timer_->stop();};
    /* QString getTempsEcoule()
    Rôle : Permet de retourner le temps écoulé sous forme de QString pour l'afficher dans le QLabel des Records notamment.
    Sortie : Une QString représentant le temps écoulé sous la forme suivante : jour, heures, minutes, secondes.
    */
    QString getTempsEcoule();
    /* quint32 getTempsInt()
    Rôle : Retourne le temps écoulé en seconde sous forme d'un quint32. Pour être comparé avec le record précédent.
    Sorti : quint32, le temps écoulé en secondes.
    */
    quint32 getTempsInt() {return (secondes_ + minutes_ * 60 + heures_ * 60 * 60 + jours_ * 24 * 60 * 60);};
    // Fin : Méthodes publiques

private:
    QLabel * label_chronometre_ = Q_NULLPTR; // Pointeur vers le QLabel ou sera affiché le temps. (Celui du labyrinthe)

    // Début : Attributs de la gestion du temps
    QTimer * timer_ = Q_NULLPTR;
    quint8 secondes_ = 0;
    quint8 minutes_ = 0;
    quint8 heures_ = 0;
    quint16 jours_ = 0;
    const quint16 INTERVALLE = 1000; // Le timer déclenche le signal "timeout()" toutes les INTERVALLE ms.
    // Début : Attributs de la gestion du temps

private slots:
    /*
    Rôle : Slot activé par le signal "timeout()" du timer cette fonction permet d'incrémenter le temps écoulé.
    */
    void augmenter1Seconde();
};

#endif // CHRONOMETRE_H
```

```cpp
// Auteur : Richard

#ifndef OPENGLHELPER_H
#define OPENGLHELPER_H

#include "glcolor.h"
#include "vertex.h"

#include <GL/glu.h>
#include <vector>
#include <QGLWidget>

using namespace std;

/* Classe OpenGLHelper
Rôle : Classe d'aide utlisée pour afficher des cubes.
*/
class OpenGLHelper
{
public:
    OpenGLHelper();
    /* static qint8 drawQUAD3D(QVector<Vertex> vertices, GLColor color)
    Rôle : Affiche un Rectangle selon les vertex et les couleurs indiquées et retourne un qint8 si tout s'est bien passé.
    */
    static qint8 drawQUAD3D(QVector<Vertex> vertices, GLColor color);

    /* static qint8 drawCube(QVector<QVector<Vertex>> vertices, QVector<Vertex> normales, QVector<GLColor> colors, qint8 quantite = 1,
     *                                                                                   quint8 option_de_texture = AUCUNE_OPTION)
    Rôle : Permet d'afficher un Cube selon les paramètres donnés :
    Entrées :   * QVector<QVector<Vertex>> vertices, QVector<Vertex> normales : Les Vertex et les normales pour l'éclairage.
                * QVector<GLColor> colors : Les couleurs
                * qint8 quantite : Le nombre de cubes à dessiner.
                * quint8 option_de_texture = AUCUNE_OPTION : Paramètres spécifiques à notre application pour l'affichage de Texture.
    */
    static qint8 drawCube(QVector<QVector<Vertex>> vertices, QVector<Vertex> normales, QVector<GLColor> colors, qint8 quantite = 1, quint8 option_de_texture =

    static const quint8 AUCUNE_OPTION = 0;
    static const quint8 MUR = 1;
    static const quint8 PLAFOND = 2;
};
#endif // OPENGLHELPER_H
```

```cpp
// Auteur : Richard

#ifndef QTHELPER_H
#define QTHELPER_H

#include <QWidget>
#include <QString>
#include <QMediaPlaylist>
#include <QMediaPlayer>
#include <QSound>

/* Classe QTHelper
Rôle : Classe permettant de gérer la musique, les sons et les images de fond des widgets.
*/
class QTHelper
{
public:
    QTHelper();
    ~QTHelper();

    // Début : Méthodes nécessitant une instanciation
    void jouerMusique(QString path);
    void arreterMusique();
    void jouerSon(QString path);
    void arreterSon();
    // Fin : Méthodes nécessitant une instanciation

    static void setImageDeFond(QWidget * widget, QString chemin); // Méthode statique pouvant s'utiliser sans instanciation


private:
    // Début : Attributs pour la gestion de la musique de fond
    QMediaPlaylist * playlist_ = Q_NULLPTR;
    QMediaPlayer * player_ = Q_NULLPTR;
    // Fin : Attributs pour la gestion de la musique de fond

    // Début : Attributs pour la gestion des sons
    QString path_son_en_cours_ = "";
    QSound * son_ = Q_NULLPTR;
    // Fin : Attributs pour la gestion des sons
};
#endif // QTHELPER_H
```

```cpp
// Auteurs : Prof + Modifications de Lilian

#ifndef MAZE_H
/** An implementation of Prim's algorithm for generating mazes.
 * from <http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm>
 *
 * C++ implementation by C. Ducottet
**/

#define MAZE_H

#include "cell.h"
#include <vector>
#include <list>
#include <utility>
#include <QVector>

using namespace std;

using Point=pair<int,int>;

class Maze
{
    vector<vector<Cell>> grid_;
    // taille du labyrinthe
    int width_;
    int height_;
    // position de la sortie
    int exitx_;
    int exity_;
    // orientation de la sortie
    int pos;

    // coordonnées des éléments dans la matrice
    Point exit_;
    Point initPosPlayer_;
    Point gettableItem_;
    // Matrice du labyrinthe
    QVector<QVector<qint8>> grid_number_;
    // chemin entre l'objet et la position initiale du joueurs
    list<Point> pathPlayerToItem_;

    void addFrontier(Point p,list<Point> & frontier);
    void mark(Point p,list<Point> & frontier);
    list<Point> neighbors(Point p);
    Cell::Direction direction(Point f, Point t);

public:
    Maze(int width,int height); // Dimensions du labyrinthe souhaitée, définition la position de la sortie et lance la génération
    void reinit();
    void display(bool pause=false);
    void generate(bool show=false);
    list<Point> path(Point,Point); // renvoie le chemin entre deux points dans la metrice
    list<Point> MatrixNeighbors(Point p,QVector<QVector<qint8>> grid_number_copy);
    void generateInitialPosition(Point start, int addedPoint); // génère la position du point loin du point en paramètre start
    QVector<QVector<qint8>> getGridNumber() {return grid_number_;}; // renvoie la matrice
    Point getPlayerPos() { return initPosPlayer_;};
    Point getexitPos() { return exit_;};
    Point getItemPos() { return gettableItem_;};
    int getExitOrientation() {return pos;}; // Portion de mur sur lequel se trouve la sortie
};

#endif // MAZE_H
```