Rapport du projet Starlight

Florian Knop (39310) - Gatien Bovyn (39189)

18 avril 2015

Table des matières

Introduction

Ce document vise à présenter le travail d'analyse et de programmation effectué lors de la réalisation du projet du laboratoire Langage C++ : Starlight.

Ce projet a été réalisé en binôme par Florian Knop, matricule 39310 groupe 2G13, et Gatien Bovyn, matricule 39189 groupe 2G11.

Le programme à concevoir consiste en une implémentation du modèle et d'une interface graphique du jeu baptisé Starlight, puzzle à 2 dimensions basé sur la lumière.

Ce projet a été compilé principalement avec g++ (version 4.8.2 ou supérieure) sous la distribution Linux Ubuntu (ou une de ses dérivées). La version du framework Qt utilisée est la 5.0.2 ou supérieure. Ce projet a été fait sous QtCreator, IDE OpenSource en version 2.8.1 ou supérieure.

Sections

Conventions de nommages utilisées

Dans cette section, nous présenterons les différentes conventions utilisées lors de ce projet. Nous avons décidé de reprendre certaines conventions utilisées par la STL (Librairie Standard) tout en utilisant d'autres conventions.

De manière globale, tous les noms de variables, classes, fichiers, etc. sont en anglais.

Nommage des fichiers

Les noms de fichiers sont entièrement en minuscules et possèdent les extensions .h pour les headers et .cpp pour les fichiers sources.

- `nomfichier.h`
- `nomfichier.cpp`

Classes

Les noms des classes commencent par une majuscule à chaque mot. Cette convention est également valable pour les noms d'énumérations et structures.

- `Classe`
- `NomClasse`
- `NomStruct`
- `NomEnumeration`

Variables

Variables de classe

Les noms des variables de classes sont en minuscules, les mots sont séparés par un underscore et le nom est suffixé par un underscore.

- `variable_`
- `nom_variable_`

Cette convention est également valable pour les variables d'une structure.

Variables locales

Les noms des variables locales respectent les mêmes conventions que les variables de classe, à la seule exception qu'ils ne sont pas suffixés par un underscore.

- `variable`
- `nom_variable`

Constantes

Les noms des constantes sont entièrement en majuscules et les mots sont séparés par des underscores.

- `CONSTANTE`
- `NOM_CONSTANTE`

Méthodes

D'une manière générale, les noms de méthodes sont en minuscules et séparés par des underscores.

Getters

Le nom d'un getter (accesseur en lecture de variable de classe/structure) est égal au nom de la variable de classe sans l'underscore final.

- Variable : `variable_` - Getter : `variable()`
- Variable : `nom_var_` - Getter : `nom_var()`

Setters

Le nom d'un setter (accesseur en écriture de variable de classe/structure) est égal au nom de la variable sans l'underscore final préfixé de `set`.

- Variable : `variable_` - Setter : `set_variable()`
- Variable : `nom_var_` - Setter : `set_nom_var()`

Autres méthodes

Les autres méthodes possèdent les mêmes conventions que celles énoncées ci-dessus.

- `methode()`
- `nom_methode()`

Éléments d'une énumération

Les éléments d'une énumération suivent les mêmes conventions que celles des constantes énoncées ci-dessus.

Présentation générale du projet

Cette section décrit les objectifs principaux et secondaires effectués lors de la réalisation de ce projet. Toutes les classes utilisées sont décrites dans la section suivante.

Le projet fini ouvre sur un menu principal permettant de jouer à Starlight, de voir les règles du jeu, d'accéder à l'éditeur de carte ou de simplement quitter.

Le jeu

Le projet permet de jouer à Starlight en important sa propre carte de jeu au format `.lvl` qu'il est possible de créer soi-même grâce à l'éditeur de carte (cf. "Editeur de cartes").

Il est nécessaire de préciser que le jeu part du principe que la carte fournie est sans erreurs. Une carte fournie avec erreur produira donc un arrêt immédiat de l'application.

Le fonctionnement du jeu

Le but du jeu est de déplacer un rayon provenant d'une source lumineuse pouvant être allumée ou éteinte vers une destination à l'aide de miroirs plans amovibles réfléchissant la lumière. La lumière possède une longueur d'onde comprise dans le spectre de lumière visible.

En addition avec leur déplacement, les miroirs peuvent tourner autour d'un point de pivot. Ce déplacement et cette rotation se font tout en restant dans certaines limites si celles-ci ont été définies à la création du miroir.

Une gestion des collisions a également été implémentée pour éviter que les miroirs déplacés ne heurtent les autres éléments de la carte. De ce fait, la carte de jeu fournie par Mr. Absil a légèrement été modifiée pour éviter que les miroirs ne se trouvent dans les murs à la création du niveau et qu'il soit donc impossible de les déplacer par la suite.

La carte peut aussi posséder des cristaux qui modifient la longueur d'onde du rayon les traversant, ainsi que des lentilles laissant passer la lumière d'une certaine intervalle de longueur d'onde. La couleur de la lumière est donc modifiée selon la longueur d'onde du rayon lumineux.

Et pour finir, la carte peut être munie de bombes qui terminent et font perdre instantanément la partie si celles-ci sont touchées.

L'interface

L'interface graphique du jeu Starlight a été réalisée sous Qt (en version 5.0.2 ou supérieure). Il s'agit ici d'une interface simple et minimaliste permettant d'effectuer la fonction de base demandée : jouer.

La fenêtre de jeu possède également des menus permettants de :

- `Quitter`
- `Revenir au menu principal`
- `Charger une carte`
- `Quitter la carte`
- `Voir les règles du jeu`

Les raccourcis clavier permettant d'accéder aux fonctions citées ci-dessus sont dépendants du système d'exploitation utilisé.

Les miroirs sont sélectionnés en utilisant un double clic gauche. Si un troisième clic s'ensuit, les limites de déplacement du pivot s'affichent sous la forme d'un rectangle bleu.

Les miroirs peuvent être déplacés et tournés au clavier en utilisant les touches :

- `Z` pour déplacer vers le haut
- `S` pour déplacer vers le bas
- `Q` pour déplacer vers la gauche
- `D` pour déplacer vers la droite
- `Flèche directionnelle gauche` pour tourner dans le sens anti-horloger
- `Flèche directionnelle droite` pour tourner dans le sens horloger
- `Shift` + une des touches citées ci-dessus pour se déplacer / tourner plus vite

Des sons ont été ajoutés lorsque la source est allumée ou éteinte, lorsqu'une bombe a été touchée et lorsque la destination est atteinte.

Faiblesses de l'interface L'interface ne permet pas de déplacer les miroirs à la souris, mais seulement au clavier. Le rectangle bleu affichant les limites du miroir s'affiche après un 3e clic plutôt que lors de la sélection du miroir.

Les sons ajoutés qui ont été cités ci-dessus ne s'activent pas toujours. Par exemple lorsque la source est allumée et éteinte très vite. Il se peut également que la source n'émette simplement pas de son.

Éditeur de cartes

L'éditeur de carte a été conçu sur base du modèle et des vues existantes. Il reprend tous les éléments disponibles dans un niveau du jeu Starlight.

Possibilités de l'éditeur

Au lancement de l'éditeur, il est possible de charger un niveau existant ou d'en créer un nouveau en personnalisant sa taille. Celle-ci ne peut être changée par la suite.

Une fois chargé/créé, il est alors possible d'ajouter de nouveaux éléments dans le niveau (mur, miroir, lentille, cristal, bombe). Comme prévu par l'énoncé, une seule source et une seule destination peuvent être incluses. Dès lors, celles-ci sont créées en même temps que le niveau et il n'est pas possible de les supprimer. Il n'est également pas possible de supprimer les murs extérieurs au niveau.

Tous les éléments ont la possibilité d'être modifiés à tous niveaux, les setters appropriés ayant été ajoutés dans les classes du modèle. Il est au minimum possible de déplacer tous les éléments, soit par le panel droit qui offre la possibilité de modifier un objet, soit par l'utilisation du clavier (Z pour le monter, S pour descendre, Q pour le translater sur la gauche et D pour une translation vers la droite).

Une fois le niveau adapté, il est possible de le sauvegarder au format `.lvl`. La classe `MapWriter` a été écrite pour l'occasion, qui permet de sauver dans un fichier un niveau, comme `MapReader` permet de le lire.

Certaines options ont été désactivées dans l'éditeur pour rendre l'édition plus agréable, notamment la gestion des collisions, les bruits et autres événements spéciaux.

Faiblesses de l'éditeur

La gestion des erreurs est totalement absente, ce qui signifie que l'utilisateur est responsable des données qu'il entre. Il est tout à fait possible de créer des éléments dont les caractéristiques ne permettront pas de jouer une partie (une source en dehors des limites par exemple).

La gestion dynamique des déplacements ne se fait que dans un sens, si l'on déplace un objet à l'aide du clavier ses propriétés dans le panel de droite seront automatiquement modifiées. Cependant, il faudra utiliser le bouton « Appliquer » du panel de droite afin que les changements effectués à cet endroit entrent en application.

Présentation des différentes classes

Dans cette section, nous allons décrire les différentes classes composant ce projet. L'implémentation du projet est divisée entre la partie modèle et la partie vue ainsi qu'une partie classes utilitaires. Elle est également basée sur le design pattern « Observeur / Observé » comme demandé dans les consignes.

Les classes/namespaces utilitaires

MapReader

La classe `MapReader` est la classe qui lit un fichier `.lvl` et crée le niveau (cf. "Level").

Cette classe considère que le fichier est sans erreur. Une instantiation d'objets erronés produira donc un arrêt de l'application.

Cette classe est basée sur le `Singleton Pattern`. On ne peut qu'instancier un niveau à la fois.

MapWriter

La classe `MapWriter` est la classe qui va écrire un niveau (cf. "Level") dans un fichier texte. Il s'agit donc du processus inverse de la classe `MapReader`. Cette classe sert à l'éditeur lors de la sauvegarde de la carte en cours d'édition.

Constants

`constants.h` est un header reprenant toutes les constantes utilisées dans ce projet.

- `INF` correspond à l'infini.
- `EPSILON` correspond à `0.00001`. EPSILON permet d'éviter les imprécisions entre deux nombres réels (cf. "umath - méthode equals()").
- `PI` correspond à la valeur de PI `3.14159...` avec autant de décimales possibles qu'un `double` peut contenir.
- `PI_2` correspond à la valeur de PI divisé par deux.
- `PI_4` correspond à la valeur de PI divisé par quatre.
- `PI_2_3` correspond à la valeur de PI_2 multiplié par trois.

Le namespace umath

Le namespace `umath` possède toutes les méthodes utilitaires mathématiques et géométriques servant au projet. Comme par exemple des méthodes permettant de trouver les intersections entre droites, segments, ellipses et rectangles.

`umath` reprend des méthodes d'égalité de nombre réels. Deux nombres réels sont égaux si la valeur absolue de la soustraction de ceux-ci est plus petite que la valeur de EPSILON (cf. "Constants").

Une autre méthode d'égalité permet de vérifier si un nombre équivaut à l'infini (INF ou -INF). Il y a également des méthodes de conversions entre pentes, radians, degrés, etc.

Les méthodes d'intersections sont décrites plus en détails dans la section Annexe - Démarches mathématiques.

Modèle

Dans cette section, nous allons décrire les différentes classes du modèle (classes métiers). Un squelette de classes a été fourni par Monsieur Absil. Ce squelette contenait les fichiers suivants : 'point.h, source.h, dest.h, nuke.h, wall.h, crystal.h, lens.h, mirror.h, ray.h, level.h' ainsi que leurs sources correspondantes. Nous avons décidé de modifier ce squelette tout en gardant la structure générale.

Point

La classe `Point` représente une position dans un espace à deux dimensions. Elle est munie des coordonnées `x` et `y` qui sont tous deux des nombres réels (`double`).

La classe `Point` fourni par Mr. Absil comprenait des coordonnées entières. Nous avons décidé de passer les coordonnées en nombres réels par souci de précision.

La classe permet également de calculer la distance entre deux points grâce à la formule :

$$\sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Line

La classe `Line` représente une droite de la forme : $D \equiv ax + by + c = 0$

Elle possède les trois paramètres a, b et c ainsi que l'angle que forme la droite pour éviter de devoir reconvertir la pente calculée par $\frac{-a}{b}$ en angle à chaque fois.

La classe `Line` possède des méthodes permettant de savoir si celle-ci est verticale ($b = 0$) ou horizontale ($a = 0$), si elle est perpendiculaire ou parallèle à une autre droite. Des méthodes permettant d'obtenir x selon une valeur de y donné et inversément sont également présentes. Cependant ces méthodes peuvent renvoyer une valeur infinie dans le cas où la droite est soit verticale soit horizontale.

Cette classe sert principalement à modéliser un rayon de lumière (cf. "Ray") pour trouver les intersections entre le rayon et les éléments du jeu.

LineSegment

La classe `LineSegment` représente un segment de droite possédant deux points (cf. "Point") qui sont les extrémités du segment.

Nous avons fait le choix qu'un segment puisse posséder deux fois le même point. Bien que cela n'ait pas vraiment de sens purement mathématique, un segment ayant deux fois le même point est tout simplement un point.

Le segment peut être transformé en droite grâce aux deux points le constituant et ainsi former l'équation de droite.

Le segment peut également être décalé d'un certain x et y ainsi que tourné d'un certain angle [1].

Ellipse

Cette classe représente une conique de forme elliptique. C'est-à-dire une ellipse ou un cercle.

Rectangle

Cette classe représente une forme géométrique rectangulaire. Elle possède un point supérieur gauche ainsi qu'une longueur et hauteur nous permettant de retrouver facilement les autres extrémités de la forme.

Cette classe sert à représenter les objets Source et Destination.

Element

La classe Element est la super-classe de tous les éléments pouvant se trouver sur une carte de jeu. Il s'agit de :

- `Source`
- `Dest`
- `Wall`
- `Mirror`
- `Lens`
- `Crystal`
- `Nuke`

La classe Ray (cf. "Ray") n'est pas un élément, les éléments concernent les objets pouvant interagir avec un rayon. Il s'agit principalement d'une classe `tag` pouvant donner le type de l'élément. Son rôle est de pouvoir retrouver le type d'un objet lors d'une intersection (cf. "Level").

1. "Rotation de segment"

Une énumération fortement typée est donc présente. Elle s'appelle `Type` et reprend les noms des éléments cités ci-dessus.

Source

Dest

Wall

Mirror

Lens

Crystal

Nuke

Ray

Level

Cette classe représente la classe principale du jeu, elle gère toute la logique métier du jeu. Elle permet de calculer la trajectoire du rayon lumineux.

Vue

L'interface graphique a été réalisée en 'Qt' à la main. Chaque élément visuel dispose d'un pointeur vers son équivalent dans le modèle, sur base duquel il est construit. Il observe également cet élément afin de se mettre à jour automatiquement.

Chaque élément a également été prévu pour être utilisé dans l'éditeur, et dispose ainsi de booléens pour les rendre sélectionnables quand cela est nécessaire.

Les classes composant la partie vue de l'application sont :

CrystalView

Classe modélisant un cristal, élément oval qui modifie la longueur d'onde d'un rayon si celui-ci traverse ledit cristal.

DestinationView

Classe modélisant la destination à atteindre par le rayon émis depuis la source pour gagner la partie.

ElementView

Classe servant de super-classe à toutes les éléments présents sur un `MapView`. Il s'agit de :

- `SourceView`
- `DestinationView`
- `WallView`
- `MirrorView`
- `LensView`
- `CrystalView`
- `NukeView`

LensView

Classe modélisant une lentille à travers laquelle un rayon peut passer si sa longueur d'onde est comprise entre les valeurs de la lentille. Sinon la lentille se comporte comme un mur et le rayon est stoppé.

MapView

Classe représentant le plateau de jeu, où tous les éléments sont disposés et affichés. Elle s'occupe de gérer les raccourcis clavier utilisés pour déplacer/pivoter les miroirs dans le jeu et de déplacer les éléments en général dans l'éditeur de carte.

MirrorView

Classe représentant un miroir sur lequel un rayon peut être réfléchi.

NukeView

Classe représentant une bombe, élément explosif du plateau qui fait perdre la partie si touché par un rayon lumineux (`RayView`). Dès le moment où elle est illuminée, une bombe change de couleur (passe du noir au rouge) et produit un son et un message visuel indiquant la fin de partie.

RayView

Classe modélisant un rayon lumineux, émis depuis la source (`SourceView`) et destiné à atteindre la destination (`DestView`) en se reflétant sur des miroirs et en passant à travers des cristaux si nécessaire. Un `RayView` a une couleur différente selon sa longueur d'onde comme proposé dans les objectifs secondaires, cette

couleur reflétant la couleur réelle qu'aurait un rayon lumineux de cette longueur d'onde.

SourceView

Classe représentant une source lumineuse sur le plateau de jeu. Bien que dans les consignes, la source lumineuse soit un carré, nous avons ensuite utilisé des images pour la représenter. La source dispose de 2 états, allumée ou éteinte. Passer d'un état à l'autre change l'image la représentant et produit un son d'interrupteur.

WallView

Classe représentant un mur, élément visuel sur lequel le rayon est stoppé.

Conclusion

Bibliographie

Les sons utilisés ont été produits par Mike Koenig et sont sous licence Attribution 3.0.

Bruit de bombe : http ://soundbible.com/106-Car-Explosion.html
Bruit de victoire : http ://soundbible.com/1003-Ta-Da.html
Bruit d'interrupteur : http ://soundbible.com/761-Switch.html

Annexes

Annexe A : Démarches mathématiques

Rotation de segment

La méthode de rotation de segment utilisée se base sur les formules de coordonnées polaires [2] ainsi que les formules trigonométriques d'additions [3].

Pour tourner un segment, il faut un point de pivot. Ce point de pivot doit ensuite être déplacé sur l'origine du repère $(0, 0)$.

2. Wikipedia : Coordonnées polaires
3. Wikipedia : Formules trigonométriques

Les formules de coordonnées polaires pour x et y sont :

$$x = r * \cos q$$

$$y = r * \sin q$$

r correspond à la distance entre l'origine et le point et q correspond à l'angle entre l'axe des abscisses et la droite formée avec l'origine et le point.

Lorsque l'on souhaite tourner un segment et trouver x' et y', il suffit d'augmenter ou diminuer l'angle q par un angle f comme ceci :

$$x' = r * \cos q + f$$
$$y' = r * \sin q + f$$

Grâce aux formules trigonométriques d'additions, on peut transformer ce résultat en :

$$x' = r * \cos q * \cos f - r * \sin q * \sin f$$
$$y' = r * \sin q * \cos f - r * \cos q * \sin f$$

Et pour finir, en remplaçant les deux premières égalités dans cette dernière on obtient :

$$x' = x * \cos f - y * \sin f$$
$$y' = x * \sin f + y * \cos f$$

Il faut ensuite redéplacer le pivot à son point de départ.

Trouver l'intersection entre deux droites

$$D \equiv y = ax + b \tag{1}$$
$$D \equiv x = k \tag{2}$$

(1) représente une droite non verticale. (2) représente une droite verticale où k est une valeur quelquonque sur l'axe des x.

Dans un premier temps, il faut tester que la pente a est différente pour les deux droites. Si les pentes des deux droites sont identiques, il n'y a pas d'intersection. En effet, si les pentes sont égales, cela signifie que les droites sont parallèles. Dans le cas de droites confondues, on considère qu'il n'y a pas d'intersection, bien qu'en réalité il y en ait une infinité.

Si la pente est différente, l'intersection entre deux droites est assez simple, il suffit d'injecter une variable d'une des deux équations dans l'autre. Dans le cadre du projet, il faut evidemment faire attention aux droites verticales.

Le cas d'une droite verticale :

On injecte (2) dans (1).

$$D \equiv y = ak + b \tag{3}$$

Avec (3) on obtient y du point dont le $x = k$ pour une droite verticale (2).

Pour deux droites non verticales on injecte le y d'une équation de type (1) dans le y d'une autre équation de type (1).

$$aremplir \tag{4}$$

$x = \frac{(y-b)}{a}$ pour une droite non verticale dont le y est celui obtenu plus tôt (1).

Trouver l'intersection entre une droite et un segment

Trouver le(s) intersections entre une ellipse et une droite

La formule d'une ellipse est :

$$E \equiv \frac{(x - x1)^2}{a^2} + \frac{(y - y1)^2}{b^2} = 1$$

où $x1$ et $y1$ sont respectivement les coordonnées x et y du centre de l'ellipse.

où a et b sont respectivement les rayons de l'axe x et y.

Pour trouver une intersection entre une ellipse et une droite, il faut égaler deux variables identiques :

On doit donc remplacer la variable x ou y de la droite dans l'équation de l'ellipse.

Le cas de la droite verticale :

Dans ce cas-là, il n'y a pas de choix, il faut remplacer x dans l'équation de l'ellipse. Nous avons également décidé de refactoriser l'équation en prenant le PPCM (Plus Petit Commun Multiple) de $a^2 * b^2$ que nous appellerons ici lcm pour Least Commun Multiple. Ce choix a été fait pour éviter les overflows lorsque de nombres trop grands sont mis au carré et multipliés. Bien que dans notre cas, nous avons rarement des nombres pouvant fournir de tels résultats.

$$E \equiv (lcmy \cdot (k - x1)^2) + ((y - y1)^2 \cdot lcmx) = lcm$$

où $lcmy$ est le facteur par lequel il faut multiplier b^2 (rayon y au carré) pour obtenir lcm,
où $lcmx$ est le facteur par lequel il faut multiplier a^2 (rayon x au carré) pour obtenir lcm.

$$E \equiv lcmy \cdot (k - x1)^2 + (y^2 + y1^2 - 2 \cdot y1 \cdot y) \cdot lcmx = lcm$$
$$E \equiv lcmy \cdot (k - x1)^2 + lcmx \cdot y^2 + lcmx \cdot y1^2 - 2 \cdot lcmx \cdot y1 \cdot y - lcm = 0$$

Avec ceci, il reste plus qu'à résoudre l'équation du second degré avec

$$\rho = b^2 - 4ac$$

où

$$a = lcmx$$
$$b = 2 \cdot lcmx \cdot y1 \cdot y$$
$$c = (lcmy \cdot (k - x1)^2) + (lcmx \cdot y1^2) - lcm$$

Le nombre d'intersections est différent selon la valeur de ρ.

$$n = \begin{cases} 0 & \text{si } \rho < 0 \\ 1 & \text{si } \rho = 0 \\ 2 & \text{si } \rho > 0 \end{cases}$$

$$y = \frac{-b}{2a}$$

$$y1 = \frac{(-b + \sqrt{\rho})}{2a}$$
$$y2 = \frac{(-b - \sqrt{\rho})}{2a}$$

On a donc le(s) y du/des point(s) d'intersection, et le x vaut k (de l'équation de départ).

Le cas de la droite non verticale :

C'est le même principe que le cas de la droite verticale sauf que pour trouver la deuxième variable finale il faudra remplacer la variable trouvée dans l'équation de la droite.

Trouver le(s) intersections entre une ellipse et un segment

Il s'agit du même principe que les intersections droite/segment. Il faut vérifier les intersections entre les ellipses et un segment transformé en droite et ensuite vérifier si les points d'intersections se trouvent sur le segment.