2023

Projeto Collision Course

EDUARDO ARAÚJO, 30008290
JOÃO LUCAS, 30008215
PEDRO AMARAL, 30008241

Projeto Paradigmas da Programação: Collision Course

Este projeto baseou-se na criação de uma aplicação gráfica, desenvolvida em Qt e C++, de simulação de tráfego de carros num cruzamento.

Requisitos:

Visto que o projeto foi desenvolvido por 3 elementos, o requisito foi a existência de duas ou mais faixas por sentido e carros gerados em faixas e velocidades aleatórias. No entanto, foi também desenvolvida da funcionalidade opcional, do projeto de 4 alunos, de controlo de tráfego com semáforos.

Funcionalidades:

Com base no requisito do projeto, que se encontra aplicada, foram também desenvolvidas funcionalidades que complementam esse requisito. O conjunto de *features* do programa são as seguintes:

- Seleção do número de carros a apresentar no ecrã;
- Carros gerados, aleatoriamente, em uma de quatro localizações possíveis;
- Velocidades aleatórias para cada carro gerado;
- Controlo de tráfego por semáforos;

Design e Código:

De forma a ir ao encontro dos requisitos do programa, foi verificado que pelo menos duas classes teriam de ser desenvolvidas para os objetos 'Car' e 'TrafficLight'. Ao longo do desenvolvimento, foi também verificada a necessidade de criação de uma terceira classe, 'TrafficLightLane', de apoio à classe 'TrafficLight'. No final a estrutura da nossa aplicação ficou a seguinte:

- Headers:
  - car.h
  - trafficlights.h
  - trafficlightlanes.h
  - mainwindow.h

- CPPs:

  - car.ccp

  - trafficlights.cpp

  - trafficlightlanes.cpp

  - mainwindow.cpp

  - main.cpp

- UI:

  - mainwindow.ui

main.ccp

Contém o código fonte que inicia a aplicação. Aqui é criada e apresentada a janela da GUI, que irá conter e mostrar ao utilizador, todos os objetos que irão ser criados.

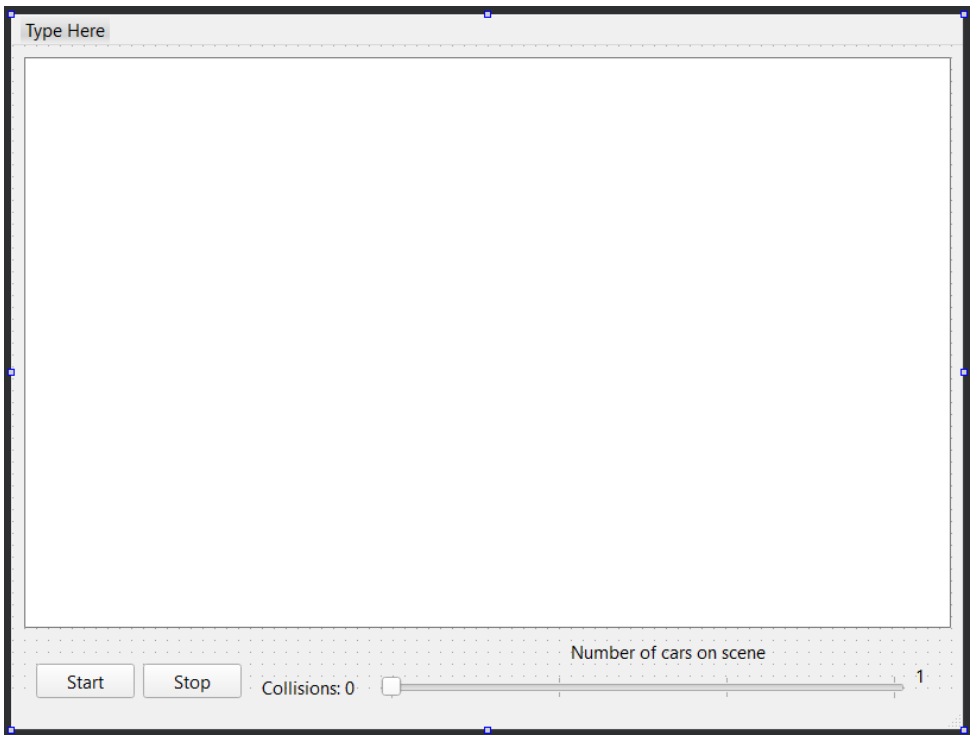
```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

main.cpp – Bibliotecas e Headers

Neste ficheiro é efetuado o *include* da biblioteca 'QApplication' para a declaração da variável 'a' e o *header* 'mainwindow.h' para a variável w, ambas utilizadas para a criação da janela da GUI

mainwindow.ui

Ficheiro com conteúdo xml onde se encontram os objetos criados com a utilização do 'Editor' de Qt. Neste caso não foi editado qualquer código diretamente no ficheiro, mas sim utilizada a opção 'Editor' do Qt para adicionarmos e posicionarmos os objetos na GUI, os valores iniciais das propriedades dos objetos criados no 'Editor' foram também definidos nesta interface.



mainwindow

Classe que contém o código base da aplicação. Aqui são editadas as propriedades dos objetos criados no 'Editor' e criados objetos do tipo 'Car', 'TrafficLights' e 'TrafficLightLanes' para depois serem adicionados à janela da GUI. Os métodos dos objetos da 'mainwindow.ui' são também definidos nesta classe.

mainwindow – Header – Bibliotecas

Para que fosse possível ter acesso a certas classes e métodos gráficos de Qt, foram importadas as bibliotecas 'QMainWindow', 'QGraphicsScene', 'QList' e 'QRandomGenerator'. Com a biblioteca 'QMainWindow' foi possível criar a janela da GUI que é apresentada ao utilizador e com a 'QGraphicsScene' podemos indicar que objetos mostrar ao utilizador.

```
#include <QMainWindow>
#include <QGraphicsScene>
#include <QList>
#include <QRandomGenerator>
```

mainwindow – Header – Variáveis e Métodos

No header de 'mainwindow' foram declarados métodos e variáveis de dois tipos, *public* e *private*. Variáveis e métodos que foram necessários aceder através de outras classes utilizadas no programa, como a variável 'scene' ou o método construtor, foram colocados em *public*, enquanto os restantes foram colocados em *private* pois apenas deverão ser acedidos pela própria classe. Para variáveis *private*, foram desenvolvidos *getters* e *setters.*

```
public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QGraphicsScene *scene;
    QList<QList<double>> possibleStarts;
    QList<QList<double>> trafficLightPosition;
    QList<QList<double>> lanePosition;
    int collisionCounter;

    int getCarQuantity();
    void setCarQuantity(int newQuantity);
```

```
private:
    int carQuantity = 0;
    Ui::MainWindow *ui;
    void spawnCar();
};
```

mainwindow – Header – Signals e Slots

Para a classe 'mainwindow' não foram necessários *Signals,* visto que esta classe não inicia comunicações com outras classes, mas foi necessário desenvolver 5 *Slots*, 1 *public* e 4 private.

```
public slots:
    void increaseCollisionCounter();

private slots:
    void on_carSlider_valueChanged(int value);

    void decreaseCarQuantity(int n);

    void on_stopButton_clicked();          ⚠ Slot

    void on_startButton_clicked();         ⚠ Slot
```

Aqui apenas o *Slot* 'increaseCollisionCounter()' é acedido por uma classe externa, os restantes *Slots* recebem sinais de elementos da Ui, que é uma variável da classe 'MainWindow'.

mainwindow – Source – Bibliotecas

No ficheiro source foi efetuado o *include* de bibliotecas diferentes das importadas no ficheiro header devido a diferentes necessidades no código. 'QImage' é utilizado em conjunto com 'QBrush' para definirmos uma imagem de fundo no objeto 'QGraphicsView', que se encontra implementado na Ui.

O *header* 'ui_mainwindow' também é incluído no ficheiro *source*. 'ui_mainwindow' é gerado automaticamente pelo Qt Creator e contém a classe 'Ui::MainWindow', que por sua vez contém todos os *widgets* e *layouts* definidos no Qt Editor. Os *headers* das nossas classes também estão incluídos, visto que é aqui que os objetos são criados.

```
#include <QImage>
#include "mainwindow.h"
#include "./ui_mainwindow.h"
#include "car.h"
#include "trafficlights.h"
#include "trafficlightlanes.h"
```

mainwindow – Source – Construtor

No código do método construtor, começamos por atribuir valores às variáveis definidas no ficheiro *header* da classe. Aqui definimos a variável 'scene', que irá conter todos os objetos a serem apresentados ao utilizador no objeto 'QGraphicsView'. Definimos também os valores das coordenadas onde os objetos das nossas classes deverão aparecer na 'scene'. Em todas as QLists, a ordem dos vetores que contêm as coordenadas é sempre Norte -> Sul -> Este -> Oeste.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    collisionCounter = 0;
    scene = new QGraphicsScene(this);
    scene->setSceneRect(0, 0, 770, 470);

    //possibleStarts' order: North, South, East, West
    possibleStarts = {{scene->width()/2 - 90, 0}, {scene->width()/2, scene->height()}, {scene->width(),scene->height()/2 - 45}, {0, scene->height()/2 + 10}};

    //Traffic lights' positions: North, South, East, West
    trafficLightPosition = {{230, 90} ,{470, 290}, {470, 90} ,{230, 290}};
    lanePosition = {{70, -50} ,{scene->width() / 2 + 10, scene->height() - 120},{scene->width() / 2 + 150, 50} ,{-60, scene->height() / 2 + 10}};
```

A seguir, é corrido um ciclo 'for' que irá criar os objetos 'TrafficLights' e TrafficLightLanes'. Os objetos são imediatamente adicionados à 'scene' pois devem ser apresentados ao utilizador ao iniciar a aplicação. É também criada uma ligação entre o *Signal* 'TrafficLight::trafficLightChanged()' e o *Slot* TrafficLightLanes::trafficLightChanged()'. Esta ligação irá permitir a comunicação entre as duas classes indicadas.

```
for(int i = 0; i < 4; i++){
    TrafficLights *trafficLight = new TrafficLights(trafficLightPosition[i]);
    scene->addItem(trafficLight);

    //Create traffic light lanes
    TrafficLightLanes *trafficLightLane = new TrafficLightLanes(lanePosition[i]);
    trafficLightLane->setPen(Qt::NoPen);
    scene->addItem(trafficLightLane);

    QObject::connect(trafficLight, SIGNAL(trafficLightChanged(int)), trafficLightLane, SLOT(trafficLightChanged(int)));
}
```

No final, são alteradas propriedades de 'QGraphicsView', é chamado o método 'spawnCar()' e apresentamos o conteúdo em 'QGraphicsView'.

```
ui->setupUi(this);
ui->view->setScene(scene);
ui->view->setSceneRect(0, 0, 770, 470);
ui->view->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
ui->view->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
ui->view->setBackgroundBrush(QBrush(QImage(":/images/resources/images/bg_02_resized.png")));

spawnCar();

ui->view->show();
```

mainwindow – Source – *Deconstructor*

Neste método é efetuado o *delete* da variável 'ui', que irá apagar da memória o conteúdo desta variável e todos os objetos que têm 'ui' como 'parent'.

```
MainWindow::~MainWindow()
{
    delete ui;
}
```

mainwindow – Source – getCarQuantity()

Getter da variável privada carQuantity. Apenas efetuado o return do valor da variável.

```
int MainWindow::getCarQuantity()
{
    return carQuantity;
}
```

mainwindow – Source – increaseCollisionCounter()

Método que corre assim que é emitido um sinal do método 'Car::collisionHappened()'. Quando ocorre uma colisão entre objetos 'Car', aumentamos 1 em 'collisionLabel' e chamamos o método 'MainWindow::decreaseCarQuantity()'. O 'int' enviado como parâmetro será a quantidade de carros a reduzir da variável.

```
void MainWindow::increaseCollisionCounter()
{
    //increase UI collision counter and call MainWindow.spawnCar() to add new cars in scene
    ui->collisionLabel->setText("Collisions: " + QString::number(++collisionCounter));

    //Reduce 2 since we deleted both colliding cars
    decreaseCarQuantity(2);
}
```

mainwindow – Source – on_carSlider_valueChanged()

Slot para o objeto 'carSlider' que se encontra em 'Ui'. Sempre que o slider for colocado num novo valor, iremos alterar o valor apresentado na label 'carQuantityLabel' de forma a indicar ao utilizador qual o valor selecionado. Como o utilizador poderá ter aumentado o número de carros apresentar no ecrã, é chamado o método 'spawnCar()' que irá confirmar se é necessário colocar mais objetos 'Car' na 'scene'.

```
void MainWindow::on_carSlider_valueChanged(int value)
{
    ui->carQuantityLabel->setText(QString::number(ui->carSlider->value()));
    spawnCar();
}
```

mainwindow – Source – on_stopButton_clicked()

Slot para o objeto 'stopButton' que se encontra em 'Ui'. Quando o utilizador clica no botão 'stop', é efetuado um ciclo 'for' que irá verificar o tipo de todos os objetos que se encontram na 'scene' e irá fazer *delete* a qualquer um que seja da classe 'Car'. Por cada 'Car' apagado, retiramos 1 ao valor da variável 'carQuantity'.

```
void MainWindow::on_stopButton_clicked()
{
    foreach (QGraphicsItem *item, scene->items()) {
        if(typeid(*(item)) == typeid(Car)){
            Car *car = dynamic_cast<Car*>(item);

            scene->removeItem(car);
            delete car;

            carQuantity--;
        }
    }
}
```

mainwindow – Source – on_startButton_clicked()

Slot para sinais enviados do objeto 'startButton'. Aqui é apenas chamado o método 'spawnCar()' para que novos carros sejam colocados na 'scene'.

```
void MainWindow::on_startButton_clicked()
{
    spawnCar();
}
```

Car

Classe com todas as propriedades dos carros que irão ser apresentados ao utilizador. Informação com as coordenadas iniciais do objeto e timers são guardadas para cada objeto 'Car' individualmente. Foram também desenvolvidas funcionalidades nesta classe para o movimento do objeto na 'scene' e deteção de colisões com outros objetos.

car – Header – Bibliotecas

Como a classe 'Car' será um objeto gráfico a apresentar na 'view', foi necessário efetuar o *include* de 'QGraphicsItem' e 'QGraphicsPixmapItem' para que ficassem disponíveis funcionalidades, como a possibilidade de definir uma imagem para o objeto, no nosso projeto. Sendo 'Car' um 'QObject', foi também efetuado o *include* de 'QObject' e 'QTimer' que irá ajudar com o movimento do objeto pela 'scene'.

```
#include <QGraphicsPixmapItem>
#include <QGraphicsItem>
#include <QTimer>
#include <QObject>
```

car – Header – Variáveis e Métodos

Para 'Car' foram definidas 3 variáveis privadas, que irão conter a informação individual de cada carro, e 5 métodos, incluindo o método construtor, que serão as funcionalidades do objeto.

```
class Car : public QObject, public QGraphicsPixmapItem
{
    Q_OBJECT
public:
    explicit Car(QList<double> startCoordinates,int carOrientation, QGraphicsItem *parent = nullptr);
    QList<double> getStartPosition();
    int getOriginalTimer();
    void setOriginalTimer(int newTimer);

private:
    int originalTimer;
    QTimer *timer;
    QList<double> startPosition;
    void deleteCar();
```

car – Header – *Signals* e *Slots*

Na classe 'Car' foi verificada a necessidade de desenvolvimento de 3 *slots* públicos e 2 *signals* para se chegar aos resultados pretendidos para a classe. Aqui, os *slots* são baseados no movimento do objeto, enquanto os *signals* dão indicação de quando ocorre uma colisão ou quando um carro é apagado da 'scene'.

```
public slots:
    void move();
    void stopTimer();
    void startTimer();

signals:
    void collisionHappened();
    void carDeleted(int);
```

Car – Source – Bibliotecas

Para o ficheiro de código fonte da classe 'Car', mantemos a necessidade de incluir a biblioteca 'QGraphicsItem' com a adição de 'QGraphicsScene' para que seja possível retirar objetos desta classe da 'scene'. O ficheiro *header* 'trafficlightlanes.h' foi também incluído, de forma a acedermos aos objetos do tipo 'TrafficLightLanes' que irão estar na 'QList colliding_Items'.

```
#include <QGraphicsItem>
#include <QGraphicsScene>
#include <QRandomGenerator>
#include <QList>
#include "car.h"
#include "trafficlightlanes.h"
```

Car – Source – *Constructor*

No método construtor, começamos por gerar um número gerado aleatoriamente entre 5 e 95, que será guardado na variável 'randomNumber' e 'originalTimer'. Este valor será utilizado em conjunto com o timer para definir a velocidade a que o método 'move()' é chamado. Quanto mais rápido o método for chamado, mais rápido o objeto se movimenta pela 'scene'.

As coordenadas iniciais do objeto, recebidas como parâmetros de entrada do *constructor*, são também guardadas na 'QList startPosition'.

```
Car::Car(QList<double> startCoordinates, int carOrientation, QGraphicsItem *parent)
    : QObject(), QGraphicsPixmapItem(parent)
{
    // set random timer
    int randomNumber = (QRandomGenerator::global()->generate() % 95) + 5;
    originalTimer = randomNumber;
    timer = new QTimer(this);

    // retrive car starting position from constructor parameters
    // we will save the start position within this class
    startPosition.append(startCoordinates[0]); //append x coordinate
    startPosition.append(startCoordinates[1]); //append y coordinate

    // send signal to move() everytime timer ends.
    // the faster the timer, the faster the car will move
    QObject::connect(timer, SIGNAL(timeout()), this, SLOT(move()));
    timer->start(randomNumber);
```

A seguir, é definida a imagem que será utilizada para apresentar o objeto ao utilizador, a posição inicial do objeto na 'scene' e o ponto central do objeto baseado nas suas dimensões. Como o objeto pode mover-se em diferentes sentidos, é efetuada a rotação do objeto com origem no seu ponto central. A rotação do objeto irá depender da sua posição inicial.

```
// set car picture and starting position
setPixmap(QPixmap(":/images/resources/images/f1_white_01_resized.png"));
setPos(startPosition[0], startPosition[1]);
setTransformOriginPoint(pixmap().width()/2, pixmap().height()/2);

//set car orientation based on start position
if(carOrientation == 0){
    setRotation(90);
}
else if(carOrientation == 1){
    setRotation(270);
}
else if(carOrientation == 2){
    setRotation(180);
}
```

Car – Source – getStartPosition()

Getter da variável privada 'startPosition', retorna o valor dessa variável.

```
QList<double> Car::getStartPosition()
{
    return startPosition;
}
```

Car – Source – getOriginalTimer()

Getter da variável privada 'OriginalTimer, retorna o valor dessa variável.

```
int Car::getOriginalTimer()
{
    return originalTimer;
}
```

Car – Source – setOriginalTimer()

Setter da variável privada 'originalTimer', necessária para atribuir novos valores à variável.

```
void Car::setOriginalTimer(int newTimer)
{
    originalTimer = newTimer;
    startTimer();

}
```

Car – Source – deleteCar()

Método chamado quando um carro deve ser apagado da 'scene' por ter ultrapassado os limites da mesma. É emitido um sinal de 'carDeleted' para o *Slot* 'MainWindow::decreaseCarQuantity()' seguido da remoção do objeto de 'scene' e o *delete* do objeto da memória.

```
void Car::deleteCar()
{
    //remove car from scene and delete object
    //emit signal to MainWindow.decreaseCarQuantity()
    emit carDeleted(1);
    scene()->removeItem(this);
    delete this;
}
```

Car – Source – move()

O método principal da classe 'Car'. Antes de iniciar o movimento do objeto, é primeiro verificado se o mesmo se encontra a colidir com um outro objeto do mesmo tipo. Caso isso se verifique, é emitido um sinal de 'collisionHappened()' para 'MainWindow::increaseCollisionCounter()' e ambos os objetos que colidiram são removidos da 'scene' e apagados da memória.

```
void Car::move()
{
    //register collisions, remove collided objects from scene and delete those same objects
    QList<QGraphicsItem *> colliding_items = collidingItems();
    for(int i = 0; i < colliding_items.size(); i++){
        if(typeid(*(colliding_items[i])) == typeid(Car)){

            //increase collision counter by emitting a signal to MainWindow.increaseCollisionCounter()
            emit collisionHappened();

            //remove collided cars from scene and delete them
            scene()->removeItem(colliding_items[i]);
            scene()->removeItem(this);

            //delete objects
            delete colliding_items[i];
            delete this;

            return;
        }
```

Caso a colisão não tenha sido entre dois objetos do tipo 'Car', é verificado se a colisão foi com um objeto do tipo 'TrafficLightLane' e, caso o objeto 'TrafficLightLane' tenha a variável 'currentLight' com o valor "red", será chamado o método 'stopTimer()' que irá parar o *timer* do carro e, por consequência, irá parar o movimento do objeto.

```
    else if(typeid(*(colliding_items[i])) == typeid(TrafficLightLanes)){

        TrafficLightLanes *trafficLightLane = dynamic_cast<TrafficLightLanes*>(colliding_items.at(i));

        if(trafficLightLane->currentLight == "red"){
            stopTimer();
        }
    }
}
```

No caso de nenhuma das verificações ser verdadeira, é efetuado o movimento do objeto. Para onde o objeto será movido, irá depender da sua posição inicial. Caso o carro ultrapasse os limites da 'scene', é chamado o método 'deleteCar()' para que o objeto seja apagado.

```
    //move car based on it's starting position and check if it went beyond scene boundaries
    //if car went over scene boundaries, call Car.deleteCar()
    if(startPosition[0] == scene()->width()/2 - 90 && startPosition[1] == 0){
        //Started north, going south
        setPos(x(), y() + 5);
        if(pos().y() > scene()->height()){
            deleteCar();
            return;
        }
    }
    else if(startPosition[0] == scene()->width()/2 && startPosition[1] == scene()->height()){
        //Started south, going north
        setPos(x(), y() - 5);
        if(pos().y() + pixmap().height() < 0){
            deleteCar();
            return;
        }
    }
    else if(startPosition[0] == scene()->width() && startPosition[1] == scene()->height()/2 - 45){
        //Started east, going west
        setPos(x() - 5, y());
        if(pos().x() + pixmap().width() < 0){
            deleteCar();
            return;
        }
    }
    else if(startPosition[0] == 0 && startPosition[1] == scene()->height()/2 + 10){
        //Started west, going east
        setPos(x() + 5, y());
        if(pos().x() > scene()->width()){
            deleteCar();
            return;
        }
    }

}
```

Car – Source – stopTimer()

Método para paragem do temporizador do carro. Irá fazer com que o objeto pare de se mover.

```
void Car::stopTimer()
{
    timer->stop();
};
```

Car – Source – startTimer()

Método para iniciar o temporizador do carro, com o valor definido na variável 'originalTimer'.

```
void Car::startTimer()
{
    timer->start(originalTimer);
}
```

TrafficLights

Classe com todas as propriedades dos semáforos. Esta classe tem como principal função, ser o ponto de controlo do utilizador para o transito. Sempre que o semáforo se encontrar com a luz vermelha ativa, todos os carros na faixa controlada pelo semáforo deverão estar parados.

TrafficLights – Header – Bibliotecas

Sendo esta classe um objeto gráfico como a classe 'Car', foi também necessário fazer *include* das bibliotecas 'QObject', QGraphicsItem' e 'QGraphicsPixmapItem' mas, ao contrário dos carros, terá de ser possível controlar os semáforos a parir de cliques do rato. Para isso foi feito o *include* da biblioteca 'Qt3DInput/QMouseEvent'.

```
#include <QObject>
#include <QGraphicsItem>
#include <QGraphicsPixmapItem>
#include <Qt3DInput/QMouseEvent>
```

TrafficLights – Header – Variáveis e Métodos

Para esta classe foram definidas 3 variáveis, duas delas constantes. Estas variáveis irão guardar o caminho das imagens a utilizar para a luzes do semáforo e guardar a informação da luz que se encontra, de momento, ligada. Em termos de métodos, a classe possuí apenas dois, o método construtor e o método 'mousePressEvent()' que irá correr quando o utilizador clicar no rato.

```
class TrafficLights : public QObject, public QGraphicsPixmapItem
{
    Q_OBJECT
public:
    explicit TrafficLights(QList<double> trafficLightPosition, QGraphicsItem *parent = nullptr);
    void mousePressEvent(QGraphicsSceneMouseEvent *event);
    const QPixmap redLight = QPixmap(":images/resources/images/traffic_red.png");
    const QPixmap greenLight = QPixmap(":images/resources/images/traffic_green.png");
    QString currentLight = "green";
```

TrafficLights – Header – *Signals* e *Slots*

A classe 'TrafficLights' tem apenas um *Signal*, que irá informar o método 'TrafficLightLanes:: trafficLightChanged()' da mudança da cor do semáforo.

```
signals:
    void trafficLightChanged(int);
```

TrafficLights – Source – Bibliotecas

Para o ficheiro *source* da classe, foi apenas incluída a biblioteca 'QGraphicsItem' que contém a classe que será *parent* de objetos da classe 'TrafficLights' e o seu ficheiro *header*.

```
#include <QGraphicsItem>
#include "trafficlights.h"
```

TrafficLights – Source – *Constructor*

No método construtor da classe, é apenas definida a posição do objeto na 'scene' e a sua imagem.

```
TrafficLights::TrafficLights(QList<double> trafficLightPosition, QGraphicsItem *parent)
    : QObject(), QGraphicsPixmapItem(parent)
{

    setPos(trafficLightPosition[0], trafficLightPosition[1]);
    setPixmap(greenLight);

}
```

TrafficLights – Source – mousePressEvent()

Método chamado sempre que é efetuado clique do rato no objeto. Aqui é feita a verificação da cor ativa no semáforo, caso seja verde a imagem é substituída para a imagem do semáforo vermelho, o valor de 'currentLight' é alterado para "red" e por fim é emitido um sinal para 'TrafficLightLanes::trafficLightChanged()' com a indicação da nova cor do semáforo. O mesmo acontece caso a cor do semáforo seja vermelha.

```
void TrafficLights::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    //set new traffic light
    if(currentLight == "green"){
        setPixmap(redLight);
        currentLight = "red";
        emit trafficLightChanged(0);
    }
    else if(currentLight == "red"){
        setPixmap(greenLight);
        currentLight = "green";
        emit trafficLightChanged(1);
    }
}
```

TrafficLightLanes

Classe de apoio ao controlo de tráfego, em conjunto com a classe 'TrafficLights'. Os objetos criados com esta classe são colocados sobre as faixas controladas pelos respetivos semáforos e têm como objetivo parar qualquer objeto 'Car' que esteja a colidir com ela, caso esteja o semáforo vermelho ativo.

TrafficLightLanes – Header – Bibliotecas

A classe, sendo das mais simples (mas não menos importantes) do programa, necessitam apenas das bibliotecas 'QObject' e 'QGraphicsItem', para terem acesso a 'colliding_items'.

```
#include <QObject>
#include <QGraphicsItem>
```

TrafficLightLanes – Header – Variáveis e Métodos

Para esta classe apenas foi necessária uma variável 'QString' para guardar a cor da luz que se encontra ativa no semáforo, que será verde ao iniciar o programa. Como método, apenas o construtor se encontra definido e precisa somente de receber as coordenadas para se posicionar na 'scene'.

```
class TrafficLightLanes : public QObject, public QGraphicsRectItem
{
    Q_OBJECT
public:
    explicit TrafficLightLanes(QList<double> lanePosition, QObject *parent = nullptr);
    QString currentLight = "green";
```

TrafficLightLanes – Header – *Signals* e *Slots*

Nesta classe foi definido o slot público 'trafficLightChanged()' que irá ser utilizado para comunicar com a classe 'TrafficLights' sobre alterações do estado dos semáforos. Não foram definidos signals.

```
public slots:
    void trafficLightChanged(int currentLight);

};
```

TrafficLightLanes – Source – Bibliotecas

No ficheiro *source* foi feito o *include* dos ficheiros *header* da própria classe e da classe 'Car', que será necessário para o 'dynamic_cast' dos objetos que se encontram em 'colliding_items'.

```
#include "trafficlightlanes.h"
#include "car.h"
```

TrafficLightLanes – Source – *Constructor*

Aqui é definido que o nosso objeto será um retângulo e será colocado nas coordenadas recebidas como parâmetros de entrada.

```
TrafficLightLanes::TrafficLightLanes(QList<double> lanePosition, QObject *parent)
    : QObject{parent}
{
    setRect(0, 0, 300, 180);
    setPos(lanePosition[0], lanePosition[1]);
}
```

TrafficLightLanes – Source – trafficLightChanged()

Método do *slot* público da classe. Aqui começamos por definir uma varável com o conteúdo de 'collidingItems()', que irá conter os carros que estão a colidir com este objeto. A seguir é verificado o valor do parâmetro de entrada de 'TrafficLights::trafficLightChanged()' e é alterada a variável 'currentLight' de acordo com o valor recebido.

```
void TrafficLightLanes::trafficLightChanged(int trafficLight)
{
    //check which cars are on traffic light lane
    QList<QGraphicsItem *> colliding_items = collidingItems();

    if(trafficLight == 1){
        currentLight = "green";
    }
    else if(trafficLight == 0){
        currentLight = "red";
    }
```

Depois, é iniciado um ciclo 'for' que irá correr todos os elementos de 'collidingItems()'. Caso a luz do semáforo tenha sido alterada para vermelha e o objeto esteja a colidir com um objeto do tipo 'Car', é criado um apontador para o espaço de memória do objeto 'Car' e o seu *timer* é parado. Caso o semáforo tenha sido alterado para verde, o apontador é criado, mas o *timer* é iniciado ao invés de parado.

```
    for(int i = 0; i < colliding_items.size(); i++){
        if(typeid(*(colliding_items[i])) == typeid(Car) && currentLight == "red"){

            Car *car = dynamic_cast<Car*>(colliding_items.at(i));
            car->stopTimer();

        }
        else if(typeid(*(colliding_items[i])) == typeid(Car) && currentLight == "green"){

            Car *car = dynamic_cast<Car*>(colliding_items.at(i));
            car->startTimer();

        }
    }
}
```

Interface Gráfica - GUI

Para a GUI, foi utilizado como exemplo a interface do programa apresentado pelo professor no E-Learning. A mesma apresenta todos os objetos colocados na 'scene', os botões 'start' e 'stop' e o *slider* de controlo do número de carros a serem colocados na 'scene' em simultâneo.