Projekt PROI 18L

Symulator ruchu miejskiego

Dokumentacja

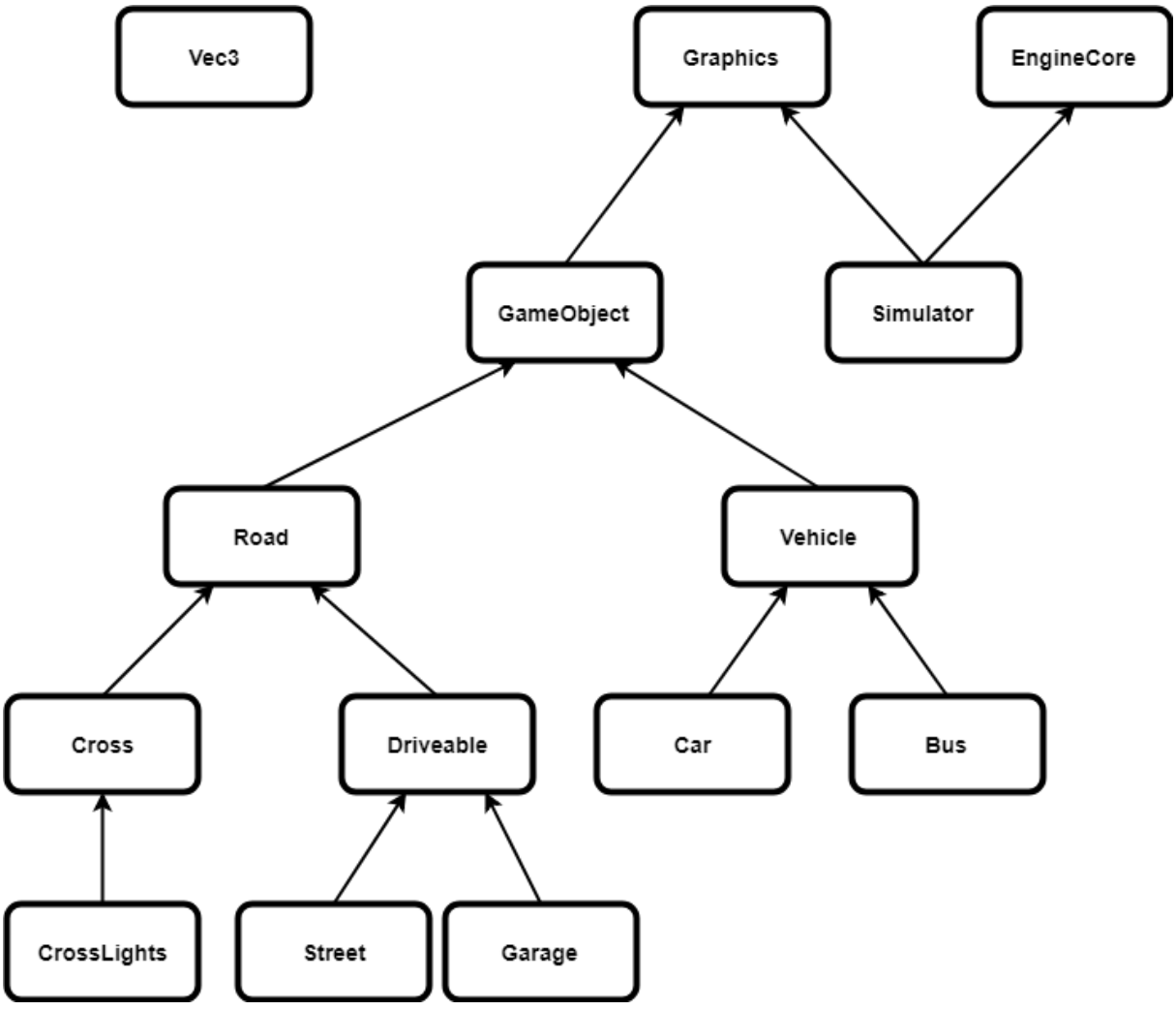
Robert Dudziński

OPIS ZADANIA

Program ma za zadanie symulować ruch pojazdów poruszających się po drogach i skrzyżowaniach. Obsługuje on dwa typy pojazdów: samochody oraz autobusy przegubowe. Jeżdżą one po drogach, które łączą skrzyżowania. Są dwa rodzaje skrzyżowań: ze światłami i bez. Do każdego skrzyżowania powinny dochodzić od dwóch do czterech dróg. Ulice dochodzące do skrzyżowania ze światłami dzielą się na dwie grupy, które na zmianę dostają pozwolenie na wjazd na to skrzyżowanie. Generowaniem pojazdów zajmują się garaże. Każdy z nich ma ustalony typ generowanych pojazdów i parametry pozwalające ustalić, ile i jak często ma tworzyć nowy pojazd. Każdy garaż może usuwać każdy rodzaj pojazdów, które do niego wjadą. Dodatkowo każdy garaż zawiera w sobie drogę dojazdową, którą trzeba połączyć z którymś skrzyżowaniem.

Program przed uruchomieniem symulacji wczytuje pliki wskazane przez użytkownika (poprzez wywołanie odpowiednich metod), zawierające sposób konfiguracji tej symulacji. Sama symulacja przedstawiona jest przez renderowanie wszystkich obiektów na scenie dzięki bibliotece OpenGL. Użytkownik jest w stanie poruszać kamerą po generowanej scenie. Tworzenie okna jak również interakcja z użytkownikiem jest zaimplementowana dzięki X11.

KLASY UŻYWANE W PROJEKCIE



KONFIGUROWANIE SYMULACJI

PLIK OPISUJĄCY OBIEKTY

Plik zawiera informacje o wszystkich obiektach (oprócz pojazdów), które mają wystąpić w symulacji. Każda linijka opisuje jedne obiekt (można zostawiać puste linie dla lepszej czytelności). Pierwszy wyraz linijki to typ tworzonego obiektu, a drugi to jego nazwa w programie. Nazwy elementów nie mogą się powtarzać. Plik ładuje się za pomocą metody loadRoad(nazwa_pliku).

TWORZENIE SKRZYŻOWANIA BEZ ŚWIATEŁ

1 wyraz – typ – CR lub CROSS

2 wyraz – nazwa – dowolna

3, 4, 5 wyraz – współrzędne w trójwymiarze – odpowiednio x, y, z (gdzie y jest składową pionową)

Przykład: CR S1 -2 0 2

TWORZENIE SKRZYŻOWANIA ZE ŚWIATŁAMI

1 wyraz – typ – CL lub CROSSLIGHTS

2 wyraz – nazwa – dowolna

3, 4, 5 wyraz – współrzędne w trójwymiarze – odpowiednio x, y, z (gdzie y jest składową pionową)

Przykład: CL L1 -6 0 -6

TWORZENIE ULICY

1 wyraz – typ – ST lub STREET

2 wyraz – nazwa – dowolna

3, 4 wyraz – nazwy skrzyżowań, które ma połączyć tworzona ulica; odpowiednio początek i koniec

Przykład: ST D1 S1 S2

TWORZENIE GARAŻU

1 wyraz – typ – GA lub GARAGE

2 wyraz – nazwa – dowolna

3 wyraz – typ tworzonych pojazdów – C dla samochodów; B dla autobusów

4 wyraz – nazwa skrzyżowania, z którym ma być połączony garaż

5, 6, 7 wyraz – współrzędne w trójwymiarze – odpowiednio x, y, z (gdzie y jest składową pionową)

8 wyraz – czas w sekundach – wskazuje jak często starać się tworzyć nowy pojazd

9 wyraz – maksymalna liczba aktualnie jeżdżących pojazdów utworzonych przez ten garaż

Przykład: GA G1 C S1 -2 0 4 5 25

UWAGA

Podczas tworzenia obiektów odwołujących się do innych po ich nazwach trzeba uwzględnić, że muszą one być już wcześniej zdefiniowane. Dlatego sugerowana kolejność jest następująca: najpierw skrzyżowania, a dopiero potem ulice i garaże.

PLIK OPISUJĄCY PIERWSZEŃSTWO NA SKRZYŻOWANIU

W pliku podaje się dla każdego skrzyżowania kolejność dróg wchodzących w je w kolejności przeciwnej do ruchu wskazówek zegara. Jeżeli nie zdefiniuje się pierwszeństwa dla któregoś skrzyżowania to zostanie ono ustalone w sposób losowy. Każda linijka to jeden konfigurowanie jednego skrzyżowania (można zrobić puste linie). Plik ładuje się za pomocą metody loadPriority(nazwa_pliku).

BUDOWA REKORDU

1 wyraz – nazwa konfigurowanego skrzyżowania

2 wyraz – liczba dróg wchodzących do tego skrzyżowania

Następne wyrazy – kolejne nazwy dróg (ulic lub garażów) w odpowiedniej kolejności

Jeżeli podczas konfiguracji symulacji program napotka błędy w plikach użytkownika to zostanie wyrzucony stosowny wyjątek i zostanie wyświetlono stosowna informacja na ekranie.

KLASA Vec3

OPIS

Klasa reprezentuje matematyczny wektor i pozwala wykonywać na nim operacje

DEKLARACJA KLASY

```
class Vec3
{
public:

    Vec3();
    Vec3(const float a, const float b, const float c);

    float x,y,z;
    static float dst(const Vec3 b, const Vec3 e);
    static float length(const Vec3 a);
    static Vec3 lerp(Vec3 b, Vec3 e, float s);
    static Vec3 cross(const Vec3 u, const Vec3 v);
    static float angleDiff(float b, float e);

    float angleXZ() const;

    void normalize();

    Vec3& operator += (const Vec3& right);
    Vec3& operator -= (const Vec3& right);
    Vec3& operator *= (const float right);
    Vec3& operator /= (const float right);

    Vec3 operator - ();
};
```

DEKLARACJE FUNKCJI

```
Vec3 operator + (Vec3 left, const Vec3& right);
Vec3 operator - (Vec3 left, const Vec3& right);
Vec3 operator * (Vec3 left, const float right);
Vec3 operator / (Vec3 left, const float right);

std::ostream& operator << (std::ostream& out, const Vec3& right);
```

METODY STATYCZNE

`static float dst(const Vec3 b, const Vec3 e)`

zwraca odległość między punktami b oraz e

`static float length(const Vec3 a)`

zwraca długość wektora a

`static Vec3 lerp(Vec3 b, Vec3 e, float s)`

Zwraca punkt leżący na prostej łączącej punkty b i e; gdy s=0 funkcja zwraca b; gdy s=1 funkcja zwraca e; dla innych s zwraca wartości pośrednie

`static Vec3 cross(const Vec3 u, const Vec3 v)`

Zwraca produkt iloczynu wektorowego wektorów u oraz v

`static float angleDiff(float b, float e)`

oblicza różnicę w stopniach między kątami b oraz e

METODY

```
float angleXZ() const
```

zwraca kąt w stopniach, podający kierunek wektora w poziomie

```
void normalize()
```

normalizuje wektor

FUNKCJE

```
Vec3 operator + (Vec3 left, const Vec3& right);
Vec3 operator - (Vec3 left, const Vec3& right);
Vec3 operator * (Vec3 left, const float right);
Vec3 operator / (Vec3 left, const float right);
```

Operatory do wykonywania operacji arytmetycznych na obiektach

```
std::ostream& operator << (std::ostream& out, const Vec3& right)
```

operator do wyświetlania składowych wektora w postaci: (x, y, z)

ZMIENNE

```
float x,y,z;
```

współrzędne wektora

KLASA Graphics

OPIS

Klasa zawiera metody związane z rysowaniem. Jej metody są oparte na OpenGL.

DEKLARACJA KLASY

```
class Graphics
{
protected:
    virtual void draw();
    void drawCube(float a) const;
    void drawCube(float x, float y, float z) const;
    void drawLine(const Vec3 begP, const Vec3 endP) const;
    void drawTile(float a) const;
    void setColor(const float r, const float g, const float b);
    void setColor(const Vec3 c);

    void drawVertex(const Vec3 a) const;
    void setNormal(const Vec3 a);
    void setNormal(const float x, const float y, const float z);
    void drawQuad(const Vec3 a1, const Vec3 a2, const Vec3 a3, const Vec3 a4) const;

    float lerp(float a, float b, float s) const;
    float lerpAngle(float a, float b, float s) const;
    int rotateDirection(float a, float b) const;

    static const unsigned int QUADS;
    static const unsigned int TRIANGLES;
    static const unsigned int LINES;
    static const unsigned int POLYGON;

    void beginDraw(const int mode);
    void endDraw();
    void drawTriangle(const Vec3 a1, const Vec3 a2, const Vec3 a3) const;
    void pushMatrix();
    void popMatrix();
    void rotateX(const float x);
    void rotateY(const float y);
    void rotateZ(const float z);
    void translate(const float x, const float y, const float z);
    void translate(const Vec3 t);
    void scale(const float x, const float y, const float z);
    void scale(const Vec3 s);
};
```

METODY WIRTUALNE

```
virtual void draw()
```

metoda rysująca obiekt

METODY

```
void drawCube(float a) const
void drawCube(float x, float y, float z) const
```
metody rysujące sześcian o podanym rozmiarze

```
void drawLine(const Vec3 begP, const Vec3 endP) const
```
metosa rysująca linię od begP fo endP

```
void drawTile(float a) const
```
metoda rysuje kwadrat w poziomie o boku długości a

```
void setColor(const float r, const float g, const float b)
void setColor(const Vec3 c)
```
metody ustawiają kolor w jakim będą rysowane następne obiekty

```
void drawVertex(const Vec3 a) const
```
metoda rysuje wierzchołek; musi być poprzedzona metodą beginDraw()

```
void setNormal(const Vec3 a)
void setNormal(const float x, const float y, const float z)
```
metody ustawiają wektor normalny do rysowanej płaszczyzny; wektor jest potrzebny do prawidłowego oświetlenia

```
void drawQuad(const Vec3 a1, const Vec3 a2, const Vec3 a3, const Vec3 a4) const
```
metoda rysuje czworokąt o podanych wierzchołkach

```
float lerp(float a, float b, float s) const
```
podaje wartość pośrednią między a oraz b w zależności od wartości s;
np. s=0 zwraca a; s=1 zwraca b; s=0.5 zwraca średnią arytmetyczną a i b

```
float lerpAngle(float a, float b, float s) const
```
działa tak samo jak metoda lerp() tylko, że dla wartości od 0 do 360 stopni

```
int rotateDirection(float a, float b) const
```
zwraca 0 gdy kąty podane w stopniach są równe;
zwraca -1 gdy trzeba się odwrócić w lewo aby z kata a osiągnąć b;
zwraca 1 gdy trzeba się odwrócić w prawo aby z kata a osiągnąć b;

```
void beginDraw(const int mode)
void endDraw()
```
metody rozpoczynające lub kończące rysowanie własnych prymitywnych figur

```
void drawTriangle(const Vec3 a1, const Vec3 a2, const Vec3 a3) const
```
rysuje trójkąt o wierzchołkach w podanych punktach

```
void pushMatrix()
void popMatrix()
```
metody odkładają na stos lub zabierają ze stosu aktualną macierz przekształceń (przesunięcie, obrót, skalowanie) w celu łatwiejszego rysowania obiektów

```
void rotateX(const float x)
void rotateY(const float y)
void rotateZ(const float z)
```
obraca grafikę względem wybranej osi

```
void translate(const float x, const float y, const float z)
void translate(const Vec3 t)
```
przesuwa grafikę o wybrany wektor

```
void scale(const float x, const float y, const float z)
void scale(const Vec3 s)
```
skaluje 3 osie grafiki o wybrane wartości

STAŁE

```
static const unsigned int QUADS;
static const unsigned int TRIANGLES;
static const unsigned int LINES;
static const unsigned int POLYGON;
```

stałe wykorzystywane przy wywoływaniu metody beginDraw(), aby określić w jakim trybie ma działać rysowanie

KLASA GameObject (dziedziczy po klasie Graphics)

OPIS

Klasa zawiera podstawowe informacje o obiekcie i umożliwia w łatwy sposób przechowywać różne typy obiektów występujących w symulatorze.

DEKLARACJA KLASY

```
class GameObject : public Graphics
{
public:
    void setPos(const Vec3 p);
    void setRot(const Vec3 r);
    Vec3 getPos() const;
    Vec3 getRot() const;
    std::string id;

    GameObject();
    virtual ~GameObject(){};

    void updateObject(const float delta);
    void drawObject();

protected:
    Vec3 pos;
    Vec3 rot;

    virtual void update(const float delta);
    static float randFloat(const float minV, const float maxV);
    static int randInt(const int minV, const int maxV);
};
```

METODY STATYCZNE

```
static float randFloat(const float minV, const float maxV)
static int randInt(const int minV, const int maxV)
```

metody zwracające losowa wartość z zakresu domkniętego <minV, maxV> typu zmiennoprzecinkowego lub całkowitego

METODY

```
void setPos(const Vec3 p)
void setRot(const Vec3 r)
Vec3 getPos() const
Vec3 getRot() const
```

Ustawiają lub zwracają pozycję lub rotację obiektu

```
void updateObject(const float delta)
```

metoda wywoływana w każdej klatce programu; wartość delta jest podana w sekundach i wyraża czas od poprzedniej klatki

```
void drawObject()
```

metoda najpierw ustawia obiekt w odpowiednim miejscu i odpowiednią rotacją, a następnie wywołuje metodę rysującą obiekt

METODY WIRTUALNE

```
virtual void update(const float delta)
```

metoda wirtualna wywoływana w każdej klatce przez metodę updateObject(); klasy dziedziczące po tej klasie mogą tu zamieszczać wszystkie obliczenia jakie chcą wykonywać

ZMIENNE

```
std::string id
```

identyfikator obiektu

```
Vec3 pos;
```

Pozycja obiektu

```
Vec3 rot;
```

obrót obiektu

KLASA Road (dziedziczy po klasie GameObject)

OPIS

Klasa jest klasą bazową dla obiektów związanymi z jezdnią

DEKLARACJA KLASY

```
class Road : public GameObject
{
public:
    static Vec3 roadColor;
protected:
    virtual ~Road(){};
};
```

ZMIENNE STATYCZNE

`static Vec3 roadColor;`
wektor opisujący kolor w jakim będą się wyświetlać elementy drogi

KLASA Driveable (dziedziczy po klasie Road)

OPIS

Klasa reprezentuje obiekty, po których mogą jeździć pojazdy

DEKLARACJA KLASY

```
class Driveable : public Road
{
public:
    Vec3 getJointPoint(const bool dir) const;
    Vec3 getNormal() const;
    Vec3 getDirection() const;

protected:
    Driveable(Cross *begCross, Cross *endCross);
    Driveable(Vec3 p, Cross *endCross);

    std::queue<Vehicle*> vehiclesBeg;
    std::queue<Vehicle*> vehiclesEnd;

    Vec3 begPos;
    Vec3 endPos;

    Vec3 begJoint;
    Vec3 endJoint;

    Vec3 getBegJointWidth(const bool dir) const;
    Vec3 getEndJointWidth(const bool dir) const;

    Vec3 direction;
    Vec3 normal;

    virtual float freeSpace(const bool dir) const;

    Cross* crossBeg;
    Cross* crossEnd;

    void draw();

private:
    float reservedSpaceBeg;
    float reservedSpaceEnd;

    void commonConstructor();

    friend Vehicle;
};
```

KONSTRUKTORY

`Driveable(Cross *begCross, Cross *endCross);`

Konstruktor tworzy obiekt połączony z dwoma skrzyżowaniami

`Driveable(Vec3 p, Cross *endCross);`

Konstruktor tworzy obiekt połączony tylko z jednym skrzyżowaniem i początku w punkcie p

METODY

```
Vec3 getJointPoint(const bool dir) const;
```

Metoda zwraca punkt przyłączenia się obiektu do skrzyżowania, ponieważ obiekt ma początek i koniec to trzeba zdefiniować o co dokładnie chodzi za pomocą argumentu dir

```
Vec3 getNormal() const;
```

Metoda zwraca poziomy wektor prostopadły do kierunku drogi

```
Vec3 getDirection() const;
```

Metoda zwraca wektor opisujący kierunek drogi

```
Vec3 getBegJointWidth(const bool dir) const;
Vec3 getEndJointWidth(const bool dir) const;
```

Metody zwracają punkty skrajne drogi – każda droga ma cztery takie punkty, gdyż droga jest prostokątem. Metody potrzebne do rysowania drogi.

```
void commonConstructor();
```

Metoda zawiera operacje wspólne dla obu konstruktorów.

METODY WIRTUALNE

```
virtual float freeSpace(const bool dir) const;
```

Metoda zwracająca ile wolnego miejsca się na niej znajduje w zależności od argumentu dir, oznaczające z której strony na nią chcemy wjechać.

ZMIENNE

```
Vec3 begPos;
Vec3 endPos;
```

Położenie odpowiednio początku i końca drogi

```
Vec3 begJoint;
Vec3 endJoint;
```

Punkt przyłączenia się do skrzyżowania odpowiednio na początku lub końcu drogi

```
Vec3 direction;
```

Wektor reprezentujący kierunek drogi

```
Vec3 normal;
```

Wektor poziomy normalny do wektora direction

```
Cross* crossBeg;
Cross* crossEnd;
```

Wskazania na skrzyżowania odpowiednio na początku i końcu drogi

```
float reservedSpaceBeg;
float reservedSpaceEnd;
```

Zmienne przedstawiają ilość zarezerwowanego miejsca dla pojazdów znajdujących się aktualnie na skrzyżowaniu

KLASA Street (dziedziczy po klasie Driveable)

OPIS

Klasa opisuje obiekt łączący dwa skrzyżowania po którym mogą poruszać się pojazdy

DEKLARACJA KLASY

```
class Street : public Driveable
{
public:
    Street(Cross *begCross, Cross *endCross);

private:
    void draw();
};
```

KLASA Garage (dziedziczy po klasie Driveable)

OPIS

Klasa opisuje obiekt zdolny do tworzenia i usuwania pojazdów. Jest podłączony do jednego skrzyżowania

DEKLARACJA KLASY

```
class Garage : public Driveable
{
public:
    Garage(Vec3 p, Cross *c);

    int vehType;

    bool checkReadyToSpot() const;
    bool checkReadyToDelete() const;

private:
    float frecSpot;
    float curTimeSpot;

    float frecDelete;
    float curTimeDelete;

    void draw();
    void update(const float delta);
    std::string itos(const int x);
    Vehicle* spotVeh();
    Vehicle* deleteVeh();

    bool isReadyToSpot;
    bool isReadyToDelete;

    int spottedVehicles;
    int maxVehicles;

    friend Simulator;
};
```

METODY

```
bool checkReadyToSpot() const;
bool checkReadyToDelete() const;
```

Metody zwracają informację, czy garaż jest gotowy do uworzenia lub usunięcia obiektu. Metody są utworzone przez program, aby symulator był w stanie rejestrować lub wyrejestrować pojazdy

```
std::string itos(const int x);
```

Metoda konwertuje liczbę całkowitą do ciągu znaków. Używana do nadawania nazw nowo powstałym pojazdom

```
Vehicle* spotVeh();
```

Metoda tworzy nowy pojazd, ustawia go i zwraca na niego wskaźnik

```
Vehicle* deleteVeh();
```

Metoda usuwa pojazd i zwraca adres na już nieistniejący obiekt, aby program mógł go usunąć z listy wszystkich obiektów

ZMIENNE

```
int vehType;
```

określa rodzaj tworzonych pojazdów przez garaż

```
float frecSpot;
```

określa w sekundach jak często próbować tworzyć nowy pojazd

```
float curTimeSpot;
```

zmienna liczy czas w sekundach od ostatniego utworzenia pojazdu

```
float frecDelete;
```

określa w sekundach jak często próbować usuwać pojazd

```
float curTimeDelete;
```

zmienna liczy czas w sekundach od ostatniego usunięcia pojazdu

```
bool isReadyToSpot;
bool isReadyToDelete;
```

wartości zwracane przez metody checkReadyToSpot() oraz checkReadyToDelete()

```
int spottedVehicles;
```

ilość pojazdów będących nadal w użyciu, utworzonych przez ten garaż

```
int maxVehicles;
```

maksymalna ilość pojazdów utworzonych przez ten garaż (będących nadal w użyciu)

KLASA Cross (dziedziczy po klasie Road)

OPIS

Klasa opisuje skrzyżowanie

DEKLARACJA KLASY

```
class Cross : public Road
{
public:
    Cross(Vec3 position);
    virtual void setDefaultPriority(Driveable *s0 = NULL, Driveable *s1 = NULL,
Driveable *s2 = NULL, Driveable *s3 = NULL);

protected:
    std::vector<OneStreet> streets;

    virtual void updateCross(const float delta);
    virtual bool dontCheckStreet(const int which);

    virtual void tryPassVehiclesWithPriority();
    virtual void tryPassAnyVehicle();

    void draw();

private:
    bool isSet;
    int allowedVeh;

    bool checkSet();
    void update(const float delta);

    friend Driveable;
    friend Vehicle;
    friend Simulator;
};
```

METODY WIRTUALNE

```
virtual void setDefaultPriority(Driveable *s0 = NULL, Driveable *s1 = NULL,
Driveable *s2 = NULL, Driveable *s3 = NULL);
```

Metoda ustawia struktury odpowiedzialne za ustępowanie pierwszeństwa. Argumenty są wskazaniami na drogi i powinny być podane w kolejności przeciwnej do ruchu wskazówek zegara

```
virtual void updateCross(const float delta);
```

Metoda aktualizuje skrzyżowanie

```
virtual bool dontCheckStreet(const int which);
```

Metoda pozwala na zdefiniowanie, które drogi mają być zignorowane podczas sprawdzania pierwszeństwa. Przydatne np. przy ignorowaniu ulic z czerwonym światłem w klasie CrossLights

```
virtual void tryPassVehiclesWithPriority();
```

Sprawdza czy na skrzyżowaniu czeka pojazd, który ma pierwszeństwo względem wszystkich innych, jeżeli tak to go nadaje mu pozwolenie na wjazd

```
virtual void tryPassAnyVehicle();
```

Jeżeli nie udało się znaleźć pojazdu mającego pierwszeństwo przed innymi, to metoda próbuje wpuścić kogokolwiek, kto możliwość wjazdu i opuszczenia skrzyżowania

METODY

```
bool checkSet();
```

Metoda sprawdza czy jest ustawione pierwszeństwo na skrzyżowaniu, jeżeli nie to próbuje ustawić jakiekolwiek pierwszeństwo

ZMIENNE

```
std::vector<OneStreet> streets;
```

wektor obiektów typu OneStreet opisujący wszystkie ulice połączone z tym skrzyżowaniem

```
bool isSet;
```

zmienna określa czy skrzyżowanie zostało skonfigurowane

```
int allowedVeh;
```

określa aktualną liczbę pojazdów, którym pozwolono na wjazd na skrzyżowanie

STRUKTURA OneStreet

Opisuje pojedynczą ulicę połączoną ze skrzyżowaniem

```
struct OneStreet
    {
        Driveable *street;
        std::vector<Vehicle*> vehicles;
        bool direction;
        Vec3 getJointPos();

        std::vector<std::vector<int> > yield;
    };
```

```
Driveable *street;
```

wskaźnik na obiekt drogi

```
std::vector<Vehicle*> vehicles;
```

wektor wskaźników na pojazdy chcące wjechać na skrzyżowanie

```
bool direction;
```

zmienna opisuje czy obecne skrzyżowanie jest początkiem czy końcem drogi

```
std::vector<std::vector<int> > yield;
```

wektor wektorów na indeksy dróg, którym trzeba ustąpić pierwszeństwa jadąc z tej drogi i chcąc skręcić w odpowiednią następną drogę.

```
Vec3 getJointPos();
```

Zwraca punkt przyłączenia drogi do skrzyżowania

KLASA CrossLight (dziedziczy po klasie Cross)

OPIS

Klasa zawiera opis skrzyżowania ze światłami jako szczególny przypadek skrzyżowania

DEKLARACJA KLASY

```
class CrossLights : public Cross
{
public:
    CrossLights(Vec3 position);
    void setLightsDurations();

    LightsDuration durLight;

private:
    std::vector<bool> defaultPriority;
    std::vector<bool> curPriority;

    void setDefaultPriority(Driveable *s0 = NULL, Driveable *s1 = NULL, Driveable *s2
= NULL, Driveable *s3 = NULL);
    void setDefaultLights(Driveable *s0, Driveable *s1, Driveable *s2, Driveable *s3);
    void setLightsPriority();

    float curTime;

    enum State{G1, Y1, B1, G2, Y2, B2};
    State curState;
    void getNextState();

    bool dontCheckStreet(const int which);

    void update(const float delta);
    void draw();
};
```

METODY

`void setDefaultLights(Driveable *s0, Driveable *s1, Driveable *s2, Driveable *s3);`

Na skrzyżowaniu ze światłami dwie grupy ulic mają na przemian pozwolenie na wjazd. Metoda ustawia te grupy. Podane drogi powinny być podane w kolejności przeciwnej do ruchu wskazówek zegara

`void setLightsPriority();`

Metoda ustawia, które stany automatu obsługującego światła, mają przepuszczać pojazdy, a które nie

`void getNextState();`

Metoda obsługuje automat służący do zmiany świateł.

ZMIENNE

```
LightsDuration durLight;
```

Obiekt opisujący czas trwania poszczególnych świateł

```
std::vector<bool> defaultPriority;
```

wektor o ilości elementów równej liczbie dróg wchodzących do tego skrzyżowania; grupuje obiekty na dwa zbiory (albo prawda albo fałsz), ponieważ na zmianę dwie grupy dróg mają pozwolenie na wjazd na skrzyżowanie

```
std::vector<bool> curPriority;
```

określa z której drogi pojazdy w obecnej sytuacji mogą starać się wjechać na skrzyżowanie (mają zielone lub żółte światło)

```
float curTime;
```

zmienna licząca czas od ostatniej zmiany stanu automatu

```
State curState;
```

Aktualny stan automatu

STRUKTURA LightsDuration

Opisuje czasy trwania poszczególnych stanów automatu

```
struct LightsDuration
    {
        float durationGreen1;
        float durationGreen2;
        float durationYellow1;
        float durationYellow2;
        float durationBreak;
    }
```

KLASA Vehicle (dziedziczy po klasie GameObject)

OPIS

Klasa reprezentuje informacje o pojeździe oraz zajmuje się jego decyzjami i animacją

DEKLARACJA KLASY

```
class Vehicle : public GameObject
{
public:
    Vehicle(Driveable *spawnRoad);
    virtual ~Vehicle(){};
    float getXPos() const;
    float getDstToCross() const;
    static int getNumberId();

    void initRandValues();

    Adjustable specs;

protected:
    float velocity;
    float xPos;
    bool isBraking;

    void update(const float delta);

    float getDst() const;
    bool isEnoughSpace() const;

    CrossingState crossState;

    Blinker blinker;

private:
    static int numVeh;

    void initPointers(Driveable *spawnRoad);

    void setVelocity();
    void checkVelocity(const float delta, float prevVelocity);
    void setNewPos();
    void registerToCross();

    void tryBeAllowedToEnterCross();
    void leaveRoad();
    void setCornerPosition();
    void enterNewRoad();

    float dstToCross;

    bool direction;

    int desiredTurn;
    Driveable *nextRoad;
    bool allowedToCross;

    Vec3 nextRoadJoint;

    Driveable *curRoad;
    Cross *curCross;
    Vehicle *frontVeh;
    Vehicle *backVeh;
    bool isFirstVeh;

    friend Garage;
    friend Cross;
};
```

METODY

```
float getXPos() const;
```

zwraca dystans pokonany na obecnym odcinku drogi

```
float getDstToCross() const;
```

zwraca dystans do najbliższego skrzyżowania

```
static int getNumberId();
```

zwraca wartość licznika liczącego wszystkie utworzone pojazdy

```
void initRandValues();
```

ustawia losowe wartości zmiennym obiektu specs typu Adjustable

```
float getDst() const;
```

zwraca dystans do najbliższego obiektu – pojazdu jadącego przed obecnym pojazdem lub w przypadku braku takiego pojazdu zwraca dystans do najbliższego skrzyżowania

```
bool isEnoughSpace() const;
```

zwraca informację czy jest wystarczająco dużo wolnego miejsca za skrzyżowaniem na drodze, na którą planuje wjechać pojazd

```
void initPointers(Driveable *spawnRoad);
```

ustawia elementy struktury CrossingState oraz wskaźniki na elementy obecnej drogi (aktualna ulica, najbliższe skrzyżowanie)

```
void setVelocity();
```

ustawia nową prędkość pojazdu kierując się odległością przed pojazdem

```
void checkVelocity(const float delta, float prevVelocity);
```

sprawdza czy prędkość otrzymana od metody setVelocity() jest poprawna dla aktualnego pojazdu oraz ewentualnie ją poprawia. Wykrywa również hamowanie dla świateł stop.

```
void setNewPos();
```

metoda znając odległość pokonaną na danym odcinku drogi, oblicza współrzędne pojazdu w świecie trójwymiarowym

```
void registerToCross();
```

metoda zapisuje się do skrzyżowania, ustawiając również parametry skrętu na najbliższym skrzyżowaniu

```
void tryBeAllowedToEnterCross();
```

gdy pojazd dostał zgodę na wykonanie manewru, sprawdza czy manewr jest w ogóle możliwy – czy za skrzyżowaniem jest odpowiednia ilość miejsca, aby zjechać, aby nie blokować ruchu

```
void leaveRoad();
```

metoda wykonuje czynności potrzebne do opuszczenia obecnej drogi i wjazdu na skrzyżowanie

```
void setCornerPosition();
```

metoda ustawia pozycję pojazdu w świecie trójwymiarowym podczas jego manewru skręcania na skrzyżowaniu

```
void enterNewRoad();
```

metoda wykonuje czynności związane z wjazdem na nową drogę

ZMIENNE

```
Adjustable specs;
```

Obiekt przechowuje wartości opisujące w jaki sposób pojazd będzie się poruszał

```
float velocity;
```

obecna prędkość pojazu

```
float xPos;
```

przybyty dystant na aktualnym odcinku drogi

```
bool isBraking;
```

zmienna przechowuje informacje czy pojazd powinien mieć zapalone światła stopu

```
CrossingState crossState;
```

Obiekt zawiera zmienne potrzebne przy korzystaniu ze skrzyżowania

```
Blinker blinker;
```

Obiekt przechowuje informacje o kierunkowskazie

```
float dstToCross;
```

dystans do następnego skrzyżowania

```
bool direction;
```

kierunek ruchu – może być od początku drogi do jej końca lub na odwrót

```
int desiredTurn;
```

indeks drogi na skrzyżowaniu, na którą pojazd planuje wjechać

```
Driveable *nextRoad;
```

wskazanie na pożądaną drogę po skorzystaniu ze skrzyżowania

```
bool allowedToCross;
```

zmienna przechowuje informację czy pojazd uzyskał zgodę na wjazd na skrzyżowanie

```
Vec3 nextRoadJoint;
```

Punkt połaczenia następnej wybranej drogi ze skrzyżowaniem

```
Driveable *curRoad;
```

wskazanie na obecną drogę po której porusza się pojazd

```
Cross *curCross;
```

wskazanie na następne skrzyżowanie, do którego jedzie pojazd

```
Vehicle *frontVeh;
```

wskazanie na pojazd jadący przed obecnym pojazdem lub NULL, gdy pojazd nie ma innych przed sobą

```
Vehicle *backVeh;
```

Wskazanie na pojazd jadący za obecnym pojazdem lub NULL, gdy nikt za nim nie jedzie

```
bool isFirstVeh;
```

zmienna przechowuje informacje czy pojazd nie ma nikogo przed sobą

STRUKTURA Blinker

Przechowuje informacje o miganiu kierunkowskazów

```
struct Blinker
    {
        int which;
        bool isLighting;

    private:
        float time;
        float duration;

        void init();
        void updateBlinkers(const float delta);

        friend Vehicle;
    }
```

`void init();`
metoda inicjalizuje wartości struktury Blinker

`void updateBlinkers(const float delta);`
metoda zapala I gasi kierunkowskaz

STRUKTURA Adjustable

Przechowuje informacje określające sposób jazdy pojazdu

```
struct Adjustable
    {
        float maxV;
        float minV;
        float cornerVelocity;
        float stopTime;
        float acceleration;
        float vehicleLength;
        float remainDst;
    }
```

STRUKTURA CrossingState

Przechowuje dane wykorzystywane podczas rejestrowanie się na skrzyżowanie oraz podczas korzystania z niego

```
struct CrossingState
    {
        bool isChanging;
        bool didReachCross;
        bool isLeavingRoad;

        float begRot;
        float endRot;
        float crossProgress;
    }
```

KLASA Car (dziedziczy po klasie Vehicle)

OPIS
Klasa definiuje samochód. Ustawia parametry i określa sposób rysowania samochodu.

DEKLARACJA KLASY

```
class Car : public Vehicle
{
public:
    Car(Driveable *spawnRoad);

private:
    void update(const float delta);
    void draw();
    void drawRoof();
};
```

METODY

```
void drawRoof();
```
metoda rysuje dach samochodu

KLASA Bus (dziedziczy po klasie Vehicle)

OPIS
Klasa definiuje autobus przegubowy. Ustawia jego parametry, a także rysuje go

DEKLARACJA KLASY

```
class Bus : public Vehicle
{
public:
    Bus(Driveable *spawnRoad);

private:
    float busAngle;

    void update(const float delta);
    void draw();
};
```

ZMIENNE

```
float busAngle;
```
zmienna trzymająca kąt pod jakim zgina się autobus na zakrętach

KLASA EngineCore

OPIS

Abstrakcyjna klasa będąca silnikiem programu. Inicjuje ona biblioteki potrzebna do działania programu: OpenGL do rysowania oraz X11 do tworzenia okna i interakcji z użytkownikiem

DEKLARACJA KLASY

```
class EngineCore
{
protected:

    int init(int argc, char **argv);
    static bool didInit;

    virtual ~EngineCore(){};

    void run();

    virtual void keyPressed(char k) = 0;
    virtual void update(float delta) = 0;
    virtual void singleUpdate(float delta) = 0;
    virtual void redraw() = 0;
    virtual void mouseMove(int dx, int dy) = 0;

private:
    int prevMouseX;
    int prevMouseY;

    timeval startTime;
    timeval lastTime;

    bool updateRatio;

    Display   *dpy;
    Window     win;
    GLboolean  doubleBuffer;

    long eventMask;

    void initLight();

public:
    static int width;
    static int height;
};
```

METODY

```
int init(int argc, char **argv);
```

Metoda inicjalizuje program poprzez inicjalizację OpenGL i X11

```
void run();
```

Metoda uruchamia pętlę główną programu

```
void initLight();
```

Metoda inicjalizuje światło na scenie

METODY WIRTUALNE

```
virtual void keyPressed(char k) = 0;
```

Abstrakcyjna metoda wywoływana przy każdym naciśnięciu klawisza k

```
virtual void update(float delta) = 0;
```

Abstrakcyjna metoda wywoływana w każdej klatce; może być wywołana wiele razy w ciągu jednej klatki w zależności od ustawienia rzeczywistego przyśpieszenia. Metoda powinna zawierać obliczenia dla aktualnego stanu programu

```
virtual void singleUpdate(float delta) = 0;
```

Metoda abstrakcyjna wywoływana zawsze tylko jeden raz w ciągu klatki. Powinna być wykorzystywana do zarządzaniem programem np. obliczanie pozycji kamery.

```
virtual void redraw() = 0;
```

Metoda abstrakcyjna wywoływana w każdej klatce. Powinno się w niej wywoływać jedynie funkcje związane z rysowaniem obiektów

```
virtual void mouseMove(int dx, int dy) = 0;
```

Abstrakcyjna metoda wywoływana zawsze podczas poruszania myszką z wciśniętym przyciskiem. Parametry dx oraz dy zawierają przesunięcie myszki w pikselach w porównaniu do poprzedniej klatki

ZMIENNE STATYCZNE

```
static bool didInit;
```

zmienna przechowuje informację czy zainicjowano program

```
static int width;
static int height;
```

trzymają rozmiar okna jakie ma się utworzyć oraz aktualny rozmiar okna po jego utworzeniu

ZMIENNE

```
int prevMouseX;
int prevMouseY;
```

pozycja kursora myszy w ostatniej klatce

```
timeval startTime;
timeval lastTime;
```

obiektu służące obługi czasu w programie np. liczenie czasu między klatkami potrzebnego do prawidłowego wykonywania animacji

```
bool updateRatio;
```

zmienna ma wartość true, gdy został zmieniony rozmiar okna

```
Display   *dpy;
Window     win;
GLboolean  doubleBuffer;
long eventMask;
```

obiekty potrzebne do utworzenia i działania okna programu

KLASA Simulator (dziedziczy po klasach EngineCore i Graphics)

OPIS

Klasa przygotowuje symulator do działania, a następnie nim zarządza. Może powstać tylko jeden obiekt tej klasy.

DEKLARACJA KLASY

```
class Simulator : public EngineCore, public Graphics
{
    friend GameObject;

public:
    static Simulator *getInstance();
    static Simulator *getInstance(int argc, char **argv);

    void loadRoad(const std::string fileName);
    void loadPriority(const std::string fileName);

    Vec3 cameraPos;
    Vec3 cameraRot;

    void run();

    GameObject* findObjectByName(const std::string on) const;

private:
    static Simulator *instance;
    Simulator(int argc, char **argv);

    void registerObject(GameObject *go);
    void destroyObject(GameObject *go);

    void cleanSimulation();

    std::vector<GameObject*> objects;
    std::vector<Garage*> spots;

    void keyPressed(char k);
    void update(const float delta);
    void singleUpdate(const float delta);
    void redraw();
    void mouseMove(const int dx, const int dy);

    enum DirectionMove
    {
        FORWARD,
        BACK,
        LEFT,
        RIGHT,
        UP,
        DOWN,
        STAY
    };

    DirectionMove cameraDirection;
    float cameraVelocity;

    void cameraMove(const float delta);
};
```

METODY STATYCZNE

```
static Simulator *getInstance();
static Simulator *getInstance(int argc, char **argv);
```

Metody zwracają instancję do obiektu, ponieważ może być utworzony tylko jeden obiekt. Ponieważ obiekt wymaga inicjalizacji z dodatkowymi argumentami to pierwsza próba zdobycia instancji powinna być wywołana z podanymi argumentami, w przeciwnym wypadku metoda wyrzuci wyjątek

METODY

```
void loadRoad(const std::string fileName);
```

Metoda wczytuje dane o obiektach z pliku wskazanego przez nazwę fileName

```
void loadPriority(const std::string fileName);
```

Metoda wczytuje informacje o pierwszeństwie na skrzyżowaniach z pliku o nazwie fileName

```
GameObject* findObjectByName(const std::string on) const;
```

Metoda zwraca wskazanie na obiekt o nazwie on. Jeżeli taki obiekt nie istnieje to jest zwracany NULL.

```
void registerObject(GameObject *go);
```

Metoda dopisuje wskazany obiekt do listy obiektów w symulatorze

```
void destroyObject(GameObject *go);
```

Metoda usuwa wskazany obiekt z listy obiektów w symulatorze.

```
void cleanSimulation();
```

Metoda jest wywoływana przed zamknięciem programu. Jej zadaniem jest wyczyszczenie pamięci po sobie

```
void cameraMove(const float delta);
```

Metoda ma za zadanie obliczać położenie kamery

ZMIENNE

```
Vec3 cameraPos;
```

Aktulna pozycja kamery

```
Vec3 cameraRot;
```

Kierunek, w którym patrzy kamera

```
std::vector<GameObject*> objects;
```

wszystkie obiekty znajdujące się na scenie. Wywoływane są dla metody update() i draw()

```
std::vector<Garage*> spots;
```

wszystkie garaże znajdujące się na scenie. Wskazania na nie są potrzebne, aby rejestrować tworzenie i usuwanie pojazdów przez te garaże

```
DirectionMove cameraDirection;
```

Kierunkek poruszania się kamery w zależności od wciskanego klawisza

```
float cameraVelocity;
```

aktualna prędkość kamery

ZMIENNE STATYCZNE

```
static Simulator *instance;
```

wskazanie na jedyną instancję obiektu Simulator

STAŁE WYSTĘPUJĄCE W PROGRAMIE

EngineCore.h

```
#define MULTIPLY_TIME        1
```

Mnożnik czasu płynące w programie

```
#define MAX_DELTA            0.15
```

Maksymalna różnica czasu podawana jako różnica między klatkami

```
#define MIN_DELTA            0.007
```

Minimalna różnica czasu podawana jako różnica między klatkami

```
#define REAL_INT_MULTIPLY    10
```

Ilość powtórzeń obliczeń w pojedynczej klatce. Dzięki temu można dokonywać dokładnych symulacji w przyśpieszeniu bez manipulowania zegarem programu

Simulator.h

```
#define CAMERA_VELOCITY      3
```

Prędkość poruszania się kamery używanej przez użytkownika