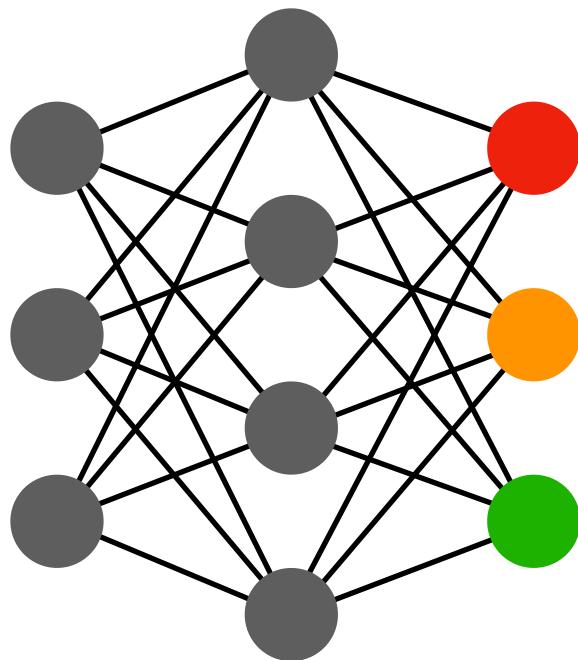


ניהול צמתי

תחברה

באמצעות רשת

נוירוניים



שם המגיש: סמואל ארביב
ת.ז: 323795658
תאריך הגשה: 3/5/2020
שם המנחה: יהודה אור
פרטי מכללה: מכללת הדסה נוורדים
סמל מוסד: 470146

מבוא	3
מטרות התוכנה	3
תיאור המערכת	3
קהל יעד	4
מבנה המערכת	5
מבנה התקיות	6
הסימולציה	7
מפת הסימולציה	7
מחלקת המפה - <i>Map</i>	8
מחלקת הצומת - <i>Intersection</i>	14
מחלקת הכביש - <i>Road</i>	18
מחלקת נתיב - <i>Lane</i>	25
מחלקת המסלול - <i>Route</i>	33
מחלקת הפaza - <i>Phase</i>	37
מחלקת המழזר - <i>Cycle</i>	40
מחלקת רמזו - <i>Light</i>	43
מנוע הסימולציה	44
מחלקת מנוע הסימולציה - <i>Engine</i>	46
מחלקת כלי הרכב - <i>Vehicle</i>	57
מחלקת הסימולציה - <i>Simulation</i>	72
מחלקת סט - <i>Set</i>	76
מחלקת ההגדרות - <i>Settings</i>	80
הבינה המלאכותית	85
רשת נוירוניים	85
אלגוריתם גנטי	86
מחלקת רשת - <i>Net</i>	87
ויזואליזציה של רשת הנוירוניים	92
מחלקת הנוירון - <i>Neuron</i>	93
אימפלמנטציה במערכת	95
פעולות רשת הנוירוניים	95
תוצאות	98
ממשק המשתמש	99

99	מבנה הממשק
101	הגדרת וניהול נתונים
107	מחלקת <i>QSFMLWindow</i>
108	חלון הסימולציה
108	ביבליוגרפיה
109	קוד המערכת
109	<i>main.cpp</i>
110	sim / map
110	<i>Cycle.cpp</i>
114	<i>Intersection.cpp</i>
121	<i>Lane.cpp</i>
126	<i>Light.cpp</i>
129	<i>Map.cpp</i>
150	<i>Phase.cpp</i>
154	<i>Road.cpp</i>
164	<i>Route.cpp</i>
167	sim / neural_network
167	<i>NeuralNet.cpp</i>
175	<i>Neuron.cpp</i>
178	sim / simulator
178	<i>DataBox.cpp</i>
180	<i>Engine.cpp</i>
203	<i>Set.cpp</i>
208	<i>Settings.cpp</i>
213	<i>Simulation.cpp</i>
215	<i>Vehicle.cpp</i>
226	ui / widgets
226	<i>QSFMLCanvas.cpp</i>
228	<i>SimModel.cpp</i>
230	ui
230	<i>mainwindow.cpp</i>

מבוא

מערכת זו מבוססת על ביצוע סימולציה המציגת פעילות בזמן אמת בקטע תחבורתי מסוים באופן ריאליסטי ומותאם, הפעלת אלגוריתמים חישוביים שונים, הפקת נתונים סטטיסטיים ומציאת מדיניות פועלה אופטימלית עבור כל צומת או מערכת צמתים המורכבת ככל שתהיה.

מטרות התוכנה

נושא התחבורה הוא אחד הנושאים הבוניים ביותר במדינה, ובעולם בכלל. בכך בו כמעט לכל אדם יש מכוניות פרטיט, ובפרט במדינות בהן ישנה מערכת תחבורה ציבורית כושלת, פקקי התנועה ואיינספור התאונות הנלוות לצפיפות בכבישים פוגעים בהיבט הכלכליים, המדיניים והחברתיים מגזר הציבורי והפרטי אחד.

מערכות התחבורה בהן משרדי התחבורה ברוחבי העולם משתמשים איןן מותאמות לתחבורה המודרנית - אלו מערכות שתוכננו לפני עשרות שנים, כאשר כמהן המכוניות בכבישים הייתה קטנה בהרבה מזו של היום, וצמחי התחבורה היו פשוטים יחסית.

כיום, נמצאות בשימוש טכנולוגיות חדשות לניהול תחבורה בצורה ממוחשבת, אך טכנולוגיות אלו התרבו להיות יקרות יותר, ובעקבות מחסור תקציבי ואף קיצוצים רבים בתקציב התחבורה, נאלצים משרדי תחבורה במקרים רבים לוותר על מערכות אלו ולהישאר עם המערכות הישנות.

במדינות כמו אוסטרליה, סינגפור ובמספר מדינות בארא"ב אומצו טכנולוגיות לניהול תחבורה בקנה מידה מסוימי (לדוגמה **SCATS GLIDE**) - אלו אומנם השקעות כלכליות עצומות, אך הן מוכחות את עצמן כ השקעות נבולות ויעילות יותר.

מטרת התוכנה היא לאפשר יעול של תנעה תחבורתית האמצעות אופטימיזציה של פעילות צמתי תחבורה, והאמת אופן פעילותם בזמן לتنעת התחבורה בכל רגע.

התוכנה אינטואיטיבית ונגישה כלל האפשר, תוך שמירה על דיק, פירוט מקסימלי, ורמת ביצועים ויעלות גבוהה.

תיאור המערכת

המערכת מאפשרת טעינה או יצירה של מערכת של צמתים (מכאן והלאה תיקרא "מפה") בצורה פשוטה וឥינטואיטיבית, והרצת סימולציה המנהלת את התנועה והזרימה התחבורתית. כמו כן, ניתן להריץ סימולציה בה תפעל מדיניות המונעת על ידי המשתמש, לצורך בדיקה והשוואה.

על כל מפה ניתן לבצע שינויים בכל רגע, ולשנות נתונים כמו רוחב נתיב, אורך כביש, הוספה/מחיקת נתיבים, עリכת המסלולים האפשריים בצומת, והצבת רמזורים ושיווכם לנתיבים ספציפיים.

כמו כן, מתאפשרה התאמת של התחבורה - שינוי של סוגי כלי הרכב המשתתפים בסימולציה (כלי רכב קטנים / גדולים / ביןוניים, משאיות, אוטובוסים), קביעת מהירות מקסימלית עבור כל אחד מהם, הסתברות הופעתם בכל נתיב, ו שינוי עומס התחבורה בכל רגע.

כאמור, אופן הפעולה של המערכת היא כזו:

1. טעינה / יצירה של מפה
2. האנה / עריכה של נתונים
3. הפעלת הסימולציה והפקת מדיניות פועלה אופטימלית

התוכנה אמורה לקבל מידע אודוט תנועת המכוניות בכל נתיב באמצעות חיישנים אלקטרו-מגנטיים המותקנים ביום ברוב הצמתים בארץ ישראל, ובאמצעות צלמות צומת המותקנות ברוב הצמתים העיקריים והחשובים בחשובים בארץ.

לאחר מכן, המדיניות המחשבת על ידי המערכת מופעלת ישירות על רמזורי המפה.

כיוון שאין צומת לרשותי, ואין לי גישה לחיישנים כאלה ואחרים, אלא לדמות זאת באופן ריאליסטי ככל הנition. כלומר, תנועת המכוניות אל תוך כל צומת תהיה רנדומלית אך מובוקרת, באופן הדומה לזרימת תחבורה במציאות - זרימת כלי רכב שאינה קבועה אך צפופה עד כמה מסוימת, בעלת רנדומליות כלשהי.

הפרויקט באתר Gitub :https://github.com/samuelarbibe/AI_TMS
סרטון תקציר: <https://www.youtube.com/watch?v=xJOcDKXWJXo>
סרטון הדוגמה: https://www.youtube.com/watch?v=BLz_PdU2oyo

קהל יעד

קהל היעד למערכת זו הוא רשותות תחבורה ציבוריות ופרטיות, ממשלות, ואך לחברי תחבורה וסטטיסטיקה.

מבנה המרחבת

התוכנה נבנתה כולה בשפת C++, שנבחרה עקב מהירותה וביוצואה. בנוסף לכך, נעשה שימוש ב-CMake. זהו כלי בניה המאפשר קישור של ספריות C++ שונות, ובנויות פרויקט המשמש ביכולן במקביל בקלות ובייעילות. במקרה זהה, נבנתה המערכת על גבי מערכת הפעלה X OS Mac, אך המערכת יכולה לפעול גם בסביבות Windows ו-Linux.

סביבת העבודה בה השתמשתי הייתה CLion. זו היא תוכנה המיצרת על ידי JetBrains, ומאפשרת תכונות מתקדם בשפות C/C++.

המערכת בנויה מ-2 חלקים עיקריים:

- מנוע הסימולציה
- ממשק המשתמש

מנוע הסימולציה נבנה תחילה לחוד, ולאחר מכן שולב אל תוך ממשק משתמש, אליו אפשר לשלוט על כל הפרמטרים השונים אודוט הסימולציה, ולקבל נתונים וסטטיסטיקה.

על מנת לבנות את מנוע הסימולציה, המשמש בגרפיקה ומנווע פיזיקלי אותו בנית, השתמש בספריה בשם SFML (Simple and Fast Multimedia Library). ספריה זו היא ספריה Cross-Platform, כלומר ספריה אשר מרכיבה עובדים במערכות הפעלה רבות.

הספריה מאפשרת קליטת מידע מהמשתמש דרך חלון אפליקציה, ונונ坦ן כלים גרפיים בסיסיים המבוססים על OpenGL, על מנת ליצור פلت אל חלון האפליקציה.

כאמור, ממשק המשתמש נבנה בשלב מאוחר יותר, ואליו שולבה מערכת הסימולציה כתת-חלון. ממשק המשתמש נבנה באמצעות ספריית Qt - ספריית ממשק המשתמש מהهواتف בעולם.

ספריה זו נונ坦ן כלים לבנות ממשק משתמש בקלות ובייעילות, אף מספקת תוכנה לייצור ממשק משתמש הנקראת Qt Creator, המאפשרת ברמה של Drag-and-Drop ליצור ממשק משתמש אינטואיטיבי ונגשימים.

כמובן שהספריה מאפשרת אך ורק יצירה של ממשק משתמש (כפתרונות, תיבות טקסט, סליידרים וכו'), אך ככל הלוגיקה שמאחוריה, ככלומר הקישור בין ממשק המשתמש למנוע האחורי, צריכה להיעשות במלואה על ידי המתכנת.

על מנת לשלב את הסימולציה הבנויה באמצעות ספריית SFML אל אפליקציה הבנויה באמצעות ספריית Qt, היה צורך לשלב את שני העולמות הללו באמצעות מחלקות מותאמות, עליהן יפורטו בהמשך.

* תהליך הפיתוח נעשה כולו בשימוש בטכנולוגיית [Qt](#), וקוד הפרויקט נמצא באתר [Github](#).

[הפרויקט המלא בGIT](#)

מבנה התקינות

```
> bin  
> data  
> public  
> resources  
> src  
◆ .gitignore  
M CMakeLists.txt  
≡ CMakeLists.txt.user
```

הפרויקט מחולק למספר תיקיות שונות, כאשר לכל אחת מהן תפקיד ותוכן שונה.

ה הפרדה נעשית על מנת להקל על תהליכי העבודה הכלול, ושמירה על סדר ואסתטיקה מסוימת.

להלן מבנה הפרויקט:

< **bin** - הקוד המומומפל והבנוי. ככלומר, תיקיה זו מכילה את התוכנה עצמה, אותה מרייצים בפועל.

< **data** - מידע שנאסף לאחר שימוש באפליקציה. כאן שמורות ملفות של צמתים, וסימולציות שנערכו.

< **public** - קבצים עוזר חיצוניים בהם נעזרתי במהלך הפרויקט. לדוגמה json/nlohmann, שהוא קובץ .hpp. המכיל פועלות עוזר המשלבות json אל תוך C++. שילוב זה יפורט בהמשך.

< **resources** - כל המשאבים בהם המערכת משתמש כגון טקסטורות וגופנים.

< **src** - הקוד עצמו. תיקיה זו מכילה את תת-תיקיות נוספות רבות, המחלקות את הקוד כתוב לחלקים שונים אשר לכל אחד מהם תפקיד ייחודי.

```
▽ src  
  > Simulator  
  > UI
```

< **simulator** - הקוד הקשור למנוע הסימולציה של המערכת.

< **ui** - הקוד הקשור למשתמש ממשך התוכנה.

הסימולציה

מפת הסימולציה

▼ map

- Cycle.cpp
- Cycle.hpp
- Intersection.cpp
- Intersection.hpp
- Lane.cpp
- Lane.hpp
- Light.cpp
- Light.hpp
- Map.cpp
- Map.hpp
- Phase.cpp
- Phase.hpp
- Road.cpp
- Road.hpp
- Route.cpp
- Route.hpp

חלק גדול ממערכת הסימולציה הוא המפה. המפה היא מערכת של צומת או כמה צמתים מחוברים זה לזה.

את המפה ניתן לבנות בתוכנה, או לטעון אותה מקובצי המחשב.¹

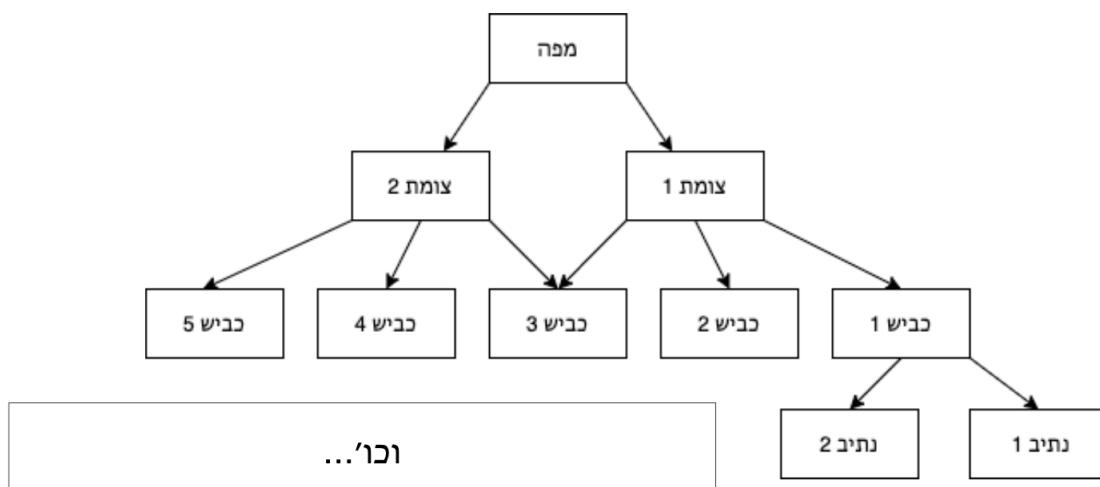
המפה בנויה ממרכיבים שונים, המסודרים בהיררכיה הבאה:

- מפה
- צומת
- כביש
- נתיב

כמו כן, מלאוים למפה מרכיבים נוספים:

- פאות
- רמזורים
- מסלולים

להלן תרשימים המתארים מפה לדוגמה:



¹ תיאור והסביר מפורט יבוא בהמשך.

המטרה היא לאפשר בניית מפה בצורה קלה וឥינטואיטיבית, כאשר רמת המורכבות של כל מפה יכולה לגדול.

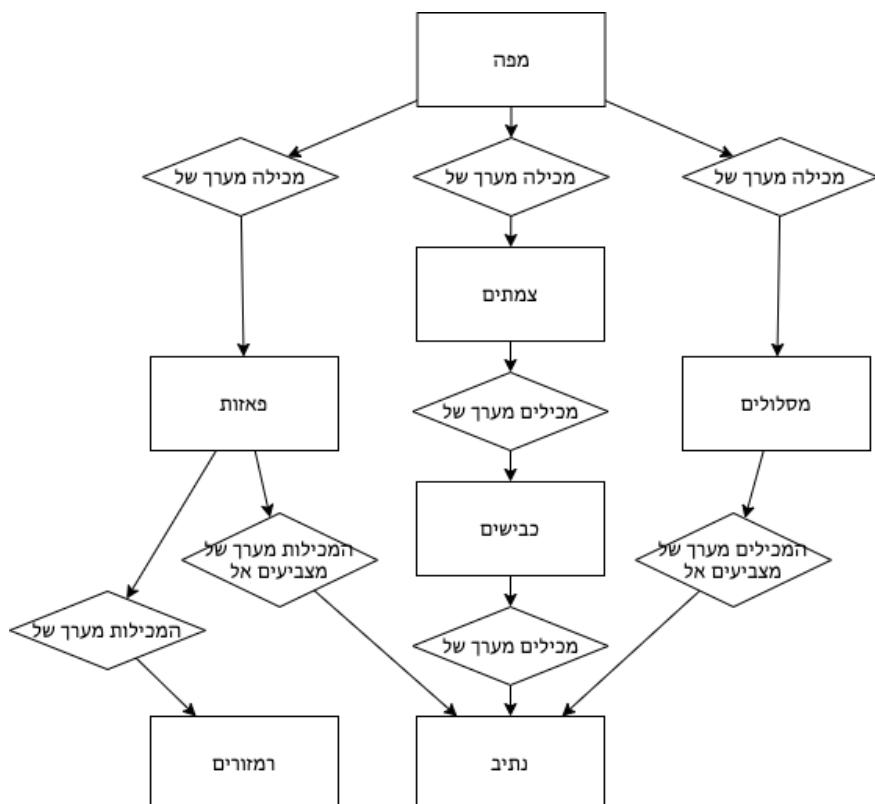
הבנייה עצמה נעשית באמצעות משק המשמש, כאשר ניתן ליצור אינטראקציה עם המפה בכל רגע על ידי להיצחה על מרכיביה השונים.

מחלקת המפה - Map

מחלקה זו היא מחלקה הראש של מפה בסימולציה.
בכל סימולציה קיים אובייקט אחד בלבד מסוג **Map**, והוא מכיל את כל שאר החלקים של המפה - כל הצמתים, הכבישים, הנטייבים, הפזיות והמסלולים.

מכיוון שזו היא המחלקה העליונה במבנה המפה, כל גישה לכל אחד מרכיבי המפה נעשה דרךה.
זאת אומרת, רובהו מורכבת מפעולות הוספה/גישה/מחיקה של כבישים/צמתים/נטיבים/פזיות.

היררכיה של המפה, ובראשיה המחלקה זו, פועלת כך:



```

class Map
{
public:

    Map(int mapNumber, int width, int height);
    ~Map();

    void Update(float elapsedTime);
    void Draw(RenderWindow *window);
    bool DeleteLane(int laneNumber);
    void ReloadMap();
    void CyclePhase();

    // entity adding
    Intersection *AddIntersection(int intersectionNumber, Vector2f position);
    Road * AddRoad(int roadNumber, int intersectionNumber, int connectionSide,
float length);
    Road * AddConnectingRoad(int roadNumber, int intersectionNumber1, int
intersectionNumber2);
    Lane * AddLane(int laneNumber, int roadNumber, bool isInRoadDirection);
    Cycle * AddCycle(int cycleNumber, int intersectionNumber = 0);
    Phase * AddPhase(int phaseNumber, int cycleNumber, float cycleTime);
    Light * AddLight(int lightNumber, int phaseNumber, int parentRoadNumber);
    Route * AddRoute(int from, int to);

    // get
    Vector2f GetSize() { return Vector2f(width_, height_); }
    Road * GetRoad(int roadNumber);
    Lane * GetLane(int laneNumber);
    Cycle * GetCycle(int cycleNumber);
    Phase * GetPhase(int phaseNumber);

    set<QString> GetLaneIdList(int phaseNumber = 0);
    set<QString> GetRoadIdList();
    set<QString> GetIntersectionIdList();
    set<QString> GetPhaseIdList();
    set<QString> GetCycleIdList();
    set<QString> GetLightIdList();
    int GetIntersectionCount() { return number_of_intersections_; }
    int GetRoadCount();
    int GetLaneCount();
    vector<Route *> *GetRoutes() { return &routes_; }
    vector<Cycle *> *GetCycles() { return &cycles_; }
    vector<Phase *> *GetPhases();
    vector<Lane *> *GetLanes();
    vector<Light *> *GetLights();
}

```

```

Intersection *GetIntersection(int intersectionNumber);
vector<Intersection *> GetIntersectionByLaneNumber(int laneNumber);
vector<Intersection *> *GetIntersections() { return &(intersections_); };
Route *GetPossibleRoute(int from);
Route *GetRouteByStartEnd(int from, int to);
Lane * GetPossibleStartingLane();
list<Lane *> * GenerateRandomTrack();

// set
bool SetPhaseTime(int phaseNumber, float phaseTime);
bool AssignLaneToPhase(int phaseNumber, int laneNumber);
bool UnassignLaneFromPhase(int laneNumber);
void SelectLanesByPhase(int phaseNumber);
void UnselectAll();
void FindStartingLanes();
bool RemoveRouteByLaneNumber(int laneNumber);
void SelectRoutesByVehicle(list<Lane*> * instructionSet);
void UnselectRoutes();

pair<ConnectionSides, ConnectionSides> AssignConnectionSides(Vector2f pos1,
Vector2f pos2);

Lane *CheckSelection(Vector2f position);
Lane *SelectedLane;

// The total count of all the maps created this session
static int MapCount;

private:

// ID of this map
int map_number_;
// Number of intersection that belong to this
int number_of_intersections_;
// number of cycles that belong to this
int number_of_cycles_;
int width_;
int height_;
// ID of the current active phase
int current_phase_index_;

vector<Route *> routes_;
vector<Lane *> starting_lanes_;
vector<Intersection *> intersections_;
vector<Cycle *> cycles_;

vector<Lane *> selected_lanes_;
vector<Route *> selected_routes_;
};


```

פונקציה לדוגמה: `DeleteLane2`

פונקציה זו היא פונקציה המקבלת מספר זיהוי של נתיב כלשהו, ומוחקת את מופעו מהmph.

פונקציה זו מורכבת מכיוון שמבנה mph, שהוא היררכיה שבין כל אלמנט mph, מחייבת השתלשלות בין כל אחד ממרכיביה על מנת להסיר את תלותו מאותו הנתיב, עד שכולם ירכיב האחרון שהוא הנתיב, ורק שם מוחקים בפועל את הנתיב.

להלן אופן הפעולה של הפונקציה:

- > האם נתיב זה מתחבר לצומת (נתיב לא יכול להתקיים ללא צומת)?
 - > מחק את כל המסלולים העוברים דרך נתיב זה
 - > הורד נתיב זה מתלווה של פאה אליה הוא שייך
 - > האם נתיב זה מחבר בין 2 צמתים?
 - > מחק נתיב זה מ-2 הצמתים
- < אחרת
- > מחק נתיב זה מהצומת אליו הוא שייך
 - > האם לא נותרו נתיבים המובילים לצומת אליו הנתיב היה שייך?
 - > מחק צומת זה
 - > האם הנתיב הוביל לצומת אחר, שגם אליו הנתיב היה שייך?
 - > מחק צומת זה
- < טען את mph מחדש
- < אחרת
- > החזר שגיאה

² מכיוון שchod קבצי המחלקות(קבצי .cpp). אורך כדי יוכל להגיע עד ל-700 שורות קוד, רק דוגמאות של פונקציות ינתנו ויפורטו.
לקוד המלא וראה פרק : "קוד המערכת" בסוף הקובץ.

```

/// delete a given lane in this map
bool Map::DeleteLane(int laneNumber) {

    vector<Intersection *> targetIntersections =
GetIntersectionByLaneNumber(laneNumber);
    Lane *lane = GetLane(laneNumber);

    if (!targetIntersections.empty())
    {
        // delete all routes that go through this lane
        RemoveRouteByLaneNumber(laneNumber);

        // delete this lane from all phases it belongs to
        UnassignLaneFromPhase(laneNumber);
        // delete the given lane
        // if lane's road is connecting, send other intersection as well to handle
deletion
        if (targetIntersections.size() > 1)
        {
            targetIntersections[0]->DeleteLane(laneNumber, targetIntersections[1]);
        } else
        {
            targetIntersections[0]->DeleteLane(laneNumber);
        }

        // if intersection has no roads left, delete it as well
        if (targetIntersections[0]->GetRoadCount() == 0)
        {
            auto it = find(intersections_.begin(), intersections_.end(),
targetIntersections[0]);
            it = intersections_.erase(it);
            delete (*it);
            number_of_intersections--;
        }

        // if road was connecting, check if the connected intersection needs to be
deleted as well
        if (targetIntersections.size() > 1)
        {
            if (targetIntersections[1]->GetRoadCount() == 0)
            {
                auto it = find(intersections_.begin(), intersections_.end(),
targetIntersections[1]);
                int intersection_number = (*it)->GetIntersectionNumber();
                intersections_.erase(it);
                delete (*it);
            }
        }
    }
}

```

```

        // if exists, delete a cycle that was attached to this
intersection;
        for (Cycle *c : cycles_)
        {
            if (c->GetIntersection()->GetIntersectionNumber() ==
intersection_number)
            {
                auto it2 = find(cycles_.begin(), cycles_.end(), c);
                cycles_.erase(it2);
                delete (*it);
                number_of_cycles_--;
            }
            number_of_intersections_--;
        }
    }

    // set selected as nullptr
    this->SelectedLane = nullptr;

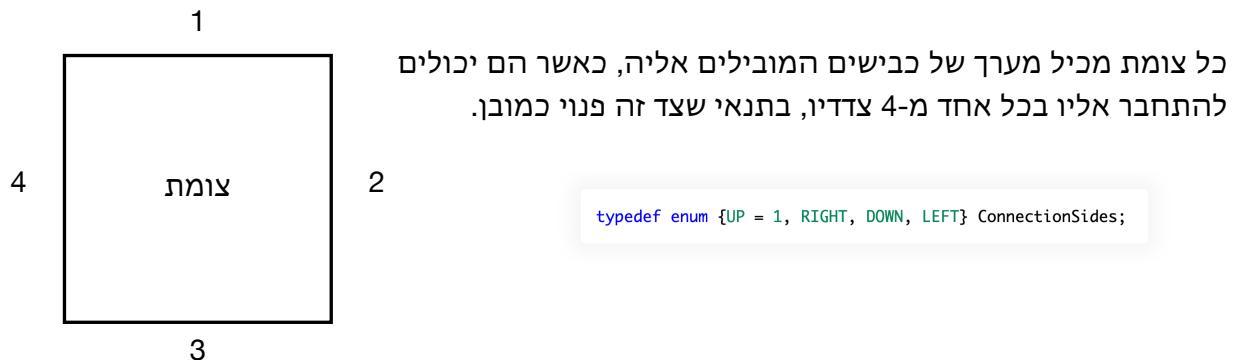
    // reload the map to display the changes
    this->ReloadMap();

    return true;
}
return false;
}

```

מחלקה הצומת - Intersection

מחלקה זו מייצגת עצם מסוג צומת. כאמור, מפה מכילה צמתים, כלומר עצם מסוג צומת הוא השני במקומו בהיררכית העצמים במפה.



```
typedef enum {UP = 1, RIGHT, DOWN, LEFT} ConnectionSides;
```

מחלקה זו יורשת מחת המחלקות הבסיסיות אותן מספקת ספריית SFML.

מחלקה זו נקראת `RectangleShape`, וירושה ממנה אפשרות להשתמש במאפיינים כמו רוחב וגובה, מיקום, צבע, מסגרת, וכמו כן מוסיפה פונקציונליות של ציור וטרנספורמציה (שינוי גודל, מיקום, נקודות יחס וכו').

קוד המחלקה

```
class Intersection : public RectangleShape
{
public:
    Intersection(Vector2f position, int intersectionNumber);
    ~Intersection();

    void ReloadIntersection();
    void ReAssignRoadPositions();
    void Update(float elapsedTime);
    void Draw(RenderWindow *window);
    bool DeleteLane(int laneNumber, Intersection *otherIntersection = nullptr);

    // Add entities
    Road *AddRoad(int roadNumber, int connectionSide, float length);
    Road *AddConnectingRoad(int roadNumber,
                           int connectionSide1,
                           int connectionSide2,
                           Intersection *connectedIntersection);
    Lane *AddLane(int laneNumber, int roadNumber, bool isInRoadDirection);
```

```

// get
Road *GetRoad(int roadNumber);
Road *GetRoadByConnectionSide(int connectionSide);
vector<Road *> *GetRoads() { return &(roads_); }
Lane *GetLane(int laneNumber);

int GetIntersectionNumber() { return intersection_number_; }
int GetRoadCount() { return roads_.size(); }
int GetLaneCount();
Vector2f GetPositionByConnectionSide(int connectionSide);

Lane *CheckSelection(Vector2f position);

// A static count of how many intersections have been
// created overall
static int IntersectionCount;

private:

// The current active vehicle count in this intersection
int current_vehicle_count_;
// The total count of vehicles that passed in this intersection
int total_vehicle_count_;
// ID of this intersection
int intersection_number_;
// The number of roads that belong to this intersection
int number_of_roads_;

int width_;
int height_;

Vector2f position_;

vector<Road *> roads_;
};

```

דוגמה לפונקציה - AddConnectingRoad

פונקציה זו אחראית על על הוספת כביש כך שייחבר בין 2 צמתים שונים.

הfonקציה מקבלת את הפרמטרים הבאים:

- מספר זיהוי חדש לביבש שעתיד להיווצר (בהינתן 0, מספר זיהוי זה יוצר בצורה סידורית אוטומטית)
- צד החיבור של הכביש אל הצומת הראשון
- צד החיבור של הכביש אל הצומת השני
- מצביע על הצומת השני

פונקציה זו מופעלת על אחד מהצמתים בהםם רוצים לחבר באמצעות כביש, וכך רק מצביע אל הצומת השני נשלח אל הפונקציה שכן פרטי הצומת הראשון נמצאים לרשות הפונקציה.

להלן אופן הפעולה של הפונקציה:

- > האם נשלח לפונקציה מספר זיהוי 0 ?
- > צור מספר זיהוי חדש בצורה סידורית
- > צור כביש חדש וווסף אותו לצומת ה затה
- > הווסף את אותו הכביש גם לצומת השני
- > החזר מצביע לכביש החדש שנוסף

קוד הפונקציה

```
/// add a connecting road between 2 different intersections
Road *Intersection::AddConnectingRoad(int roadNumber,
                                         int connectionSide1,
                                         int connectionSide2,
                                         Intersection *connectedIntersection) {
    if (!roadNumber)
    {
        roadNumber = Road::RoadCount + 1;
    }

    roads_.push_back(new Road(roadNumber,
                             this->intersection_number_,
                             connectedIntersection->intersection_number_,
                             connectionSide1,
                             connectionSide2,
                             this->GetPositionByConnectionSide(connectionSide1),
                             connectedIntersection
                             ->GetPositionByConnectionSide(connectionSide2),
                             (connectionSide1 - 1) * 90));

    connectedIntersection->roads_.push_back(roads_[number_of_roads_]);
    connectedIntersection->number_of_roads_++;

    number_of_roads_++;
    Road::RoadCount++;

    if (Settings::DrawAdded)
        std::cout << "Connecting Road " << roadNumber << " added between
intersections "
                     << this->intersection_number_ << " <--> "
                     << connectedIntersection->intersection_number_ << "" << endl;
}

return roads_[number_of_roads_ - 1];
}
```

מחלקת הכביש - Road

מחלקת הכביש היא המיצגת את אובייקט הכביש, שהוא האלמנט השלישי במקומו בהיררכיה מערכת המפה.

כאמור, כל צומת מכילה עד 4 כבישים המתחברים אליו בצדין, וכןו כן כל כביש מכיל בין-1 לאינסוף נתיבים.

בדומה לצומת, מחלקה הכביש ירושת גם היא ממחלקה RectangleShape המסופקת על ידי ספריית SFML, ומורישה ממנו מאפיינים ופונקציונליות כמו ציור אובייקט מרובע.

ישנם שני תת-סוגים של כבישים:

- כביש רגיל
- כביש חיבור

ההבדל ביניהם הוא שכביש רגיל מת לחבר לצומת אחד, ואילו כביש חיבור מחבר בין 2 צמתים.

ב כדי להימנע מיצירת מחלקה נוספת עבור שני סוגי הכבישים הללו שביניהם הבדל יחסית מינורי, הוחלט ליצור 2 פונקציות בונות שונות ("קונסטרוקטורים"), הדורסות אחת את השנייה בהתאם לצורך.

נוסף לכך, משתנים כמו מספר הזיהוי של הצומת אליו שייך אותו הכביש נהפכו למערכים בעלי 2 ערכיים:

- `מספר_צומת_אליו_אני_מתחבר[0]` - מייצג את מספר הצומת הראשון אליו הכביש מחובר.
- `מספר_צומת_אליו_אני_מתחבר[1]` - מאותחל כ-0. במקרה ונעשה שימוש בפונקציה הבונה של הכביש המחבר, המקום זה במערך יושם מספר הזיהוי של הצומת השני אליו הכביש מתחבר.

כמו כן, ישנו מאפיין בוליאני (boolean) `אם_אני_כביש_מחבר?` שמתאר האם הכביש הוא מחבר או לא, כך נדע מתי המיקום השני במערך "מספר_צומת_אליו_אני_מתחבר" רלוונטי.

לכביש ישנו גם אלמנט של כיוון. הכוון מיוצג על ידי זווית, כאשר באופן גלובלי זווית 0 נקבעה כלפי מעלה.

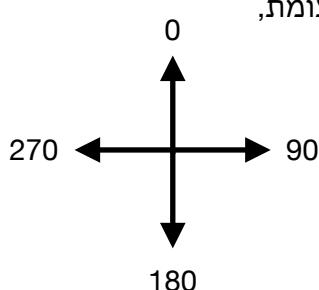
כיוונו של כביש נקבע בצורה הבאה:

1. כביש מחבר - כיוונו של הכביש נקבע כלפיון הווקטור שבין הצומת המחבר ה-1 לבין הצומת המחבר ה-2.

2. כביש שאינו מחבר - כיוונו של הכביש נקבע ביחס לנקחת החיבור שלו עם הצומת, כאשר בין כל צד חיבור יש הפרש של 90 מעלות כלפיון.

להלן תרשימים המתאר את הכוונים שנקבעו בצורה גלובלית.

כך לדוגמה, כביש המתחבר לצומת מצד מספר 2 (ראה תרשימים צדי חיבור בעמ' 12) יהיה כלפיו 90 מעלות.



```

class Road : public RectangleShape
{
public:

    Road(int roadNumber,
        int intersectionNumber,
        int connectionSide,
        Vector2f startPosition,
        float length,
        float direction);
    Road(int roadNumber,
        int intersectionNumber1,
        int intersectionNumber2,
        int connectionSide1,
        int connectionSide2,
        Vector2f conPosition1,
        Vector2f conPosition2,
        float direction);
    ~Road();

    void Draw(RenderWindow *window);
    void Update(float elapsedTime);
    void ReloadRoadDimensions();
    void BuildLaneLines();

    // Add entities
    Lane *AddLane(int laneNumber, bool isInRoadDirection);

    // get
    Lane *GetLane(int laneNumber);
    float GetWidth() { return width_; }
    int GetRoadNumber() { return road_number_; }
    int GetIntersectionNumber(int index = 0) { return
intersection_number_[index]; }
    int GetConnectionSide(int index = 0) { return connection_side_[index]; }
    bool GetIsConnecting() { return is_connecting_; }
    int GetLaneCount() { return lanes_.size(); }
    float GetRoadDirection() { return direction_; }
    int GetCurrentVehicleCount() { return current_vehicle_count_; }
    int GetTotalVehicleCount() { return total_vehicle_count_; }
    Vector2f GetStartPosition() { return start_pos_; }
    Vector2f GetEndPosition() { return end_pos_; }
    vector<Lane *> *GetLanes() { return &(lanes_); }

    // set
    void ReAssignLanePositions();
    void UpdateStartPosition(Vector2f position);
}

```

```

void UpdateEndPosition(Vector2f position);
bool DeleteLane(int laneNumber);

Lane *CheckSelection(Vector2f position);

// The total count of all the roads created this session
static int RoadCount;

private:

// ID of this road
int road_number_;
// The number of lanes that belong to this road
int number_of_lanes_;
// ID of intersection / intersections this road connects to
// if [1] is empty, this is a non-connecting road
int intersection_number_[2];
// The connection side of this road to intersection_number_[0] and
// intersection_number_[1], respectively
int connection_side_[2];
// The current count of the active vehicles in this road
int current_vehicle_count_;
// The count of all the vehicles that passed through this road
int total_vehicle_count_;
// Is this road connecting
bool is_connecting_;

Vector2f start_pos_;
Vector2f end_pos_;

float direction_;
float length_;
float width_;

vector<Lane *> lanes_;

vector<LaneLine> lane_lines_;

DataBox *data_box_;
};

```

דוגמה לפונקציה - () ReAssignLanePositions

כasher מתווסף נתיב לתוך כביש, יש צורך לסדר את מיקום כל הנתיבים מחדש בתחום הכביש.

לאחר שהתווסף נתיב חדש למערך הנתיבים של כביש מסוים, נחשב את רוחבו החדש של הכביש בצוותה הבאה:

```
width_ = Settings::LaneWidth * number_of_lanes_;
```

כמו שניתן לראות, רוחב הכביש מחושב על ידי הכפלת מספר הנתיבים שהוא מכיל באורך הנתיבים, שהוא ערך המשותף לכל הנתיבים.

לאחר שוחשב רוחב הכביש, יש צורך בפונקציה שתடע למקם כל נתיב מחדש, כך שהנתיבים יתפסו את מקומם בכביש באופן התואם את סדרם במערך הנתיבים.

להלן דוגמה של כביש בעל נתיב אחד. מיקום הנתיב במצב זה הוא כמיקום הכביש, שכן גודל הכביש הוא כגודל הנתיב היחיד הזה. ככלומר במצב פשוט זה, lane_position = road_position = lane_.position. ברגע שמתווסף נתיב, יש צורך לחשב מחדש את מיקומם של כל הנתיבים בכביש.



מכיוון שמייקום הכביש במפה (הנקודות המסומנות באדום) חייב להיות סטטי, על הנתיבים לשנות את מיקומם ביחס אליו (מיוקמי הנתיבים מסומנים בנקודה כחולה).

בכדי למקם נתיב מחדש, יש צורך ב 2 נקודות - נקודת התחלה, ונקודת סוף.

- בכדי לחשב את נקודות אלו ביחס למיקום הכביש, נעשה שימוש ב-2 ווקטור עזר.
1. ווקטור רוחב נתיב (חץ כתום), כאמור רוחב הנתיב הוא מוגדר ומשותף לכל הנתיבים.
 2. ווקטור אורך נתיב (חץ ורוד), שהוא מייצג את אורך הנתיב שווה לאורך הכביש.

בנוסף, נעשה שימוש בנקודות עזר אחת, והיא מיקום התחלת הנתיב העליון, אשר ביחס אליו



ממוקמיםשאר הנתיבים. נתיב זה מסומן בנקודה יירוקה.

מכיוון שרוחב הנתיב שמור בווקטור הכתום, נוכל להוסיף אותו לנקודה הירוקה וכך לקבל את מיקום הנתיב הבא, וכן להלее, שכן המרחק בין מרכזי הנתיבים הוא כרוחב נתיב אחד.

בכדי לקבל את נקודת הסוף של כל נתיב, פשוט נוסיף לנקודות התחלתה של הנתיב שהיחסנו את הווקטור הוורד, שהוא ווקטור אורך הנתיב, ונקבל את נקודת הסוף של אותו הנתיב.

הנקודה הירוקה, שהיא מיקום התחלת הנתיב העליון, מחושבת בצורה הבאה:

```
t.rotate(direction_ + 90);
laneDifference = t.transformPoint(0.f, -1.f) * Settings::LaneWidth;

(number_of_lanes_ % 2) ?
x.scale(number_of_lanes_ / 2, number_of_lanes_ / 2) :
x.scale((number_of_lanes_ - 1) / 2 + 0.5, (number_of_lanes_ - 1) / 2 + 0.5);

firstLaneDifference = x.transformPoint(laneDifference);

firstLanePoint = start_pos_ - firstLaneDifference;
```

< אם מספר הנתיבים אי-זוגי
 < הכפל את הווקטור הכתום במחצית מספר הנתיבים בכביש
 < אחרת
 < הכפל את הווקטור הכתום המחזית מספר הנתיבים בכביש, ועד חצי
 < הוריד את הווקטור הכתום מנקודת ההתחלה של הכביש לקבלת הנקודה הירוקה

לאחר שמנצחנו את **הנקודה הירוקה**, עברו כל נתיב נוסף ווקטור כתום לנק' ההתחלה ועד וורטור וורוד לנק' הסוף.

קוד הפקנץיה

```
// re-locate all lanes in road to align with the road
void Road::ReAssignLanePositions() {

    Vector2f firstLanePoint;
    Vector2f firstLaneDifference;
    Vector2f laneDifference;
    Vector2f lengthVec;

    // temporarily deactivate Deletion drawing
    bool prevState = Settings::DrawDelete;
    Settings::DrawDelete = false;

    Transform t, x;

    t.rotate(direction_ + 90);
    laneDifference = t.transformPoint(0.f, -1.f) * Settings::LaneWidth;

    (number_of_lanes_ % 2) ?
    x.scale(number_of_lanes_ / 2, number_of_lanes_ / 2) :
    x.scale((number_of_lanes_ - 1) / 2 + 0.5, (number_of_lanes_ - 1) / 2 + 0.5);

    firstLaneDifference = x.transformPoint(laneDifference);

    firstLanePoint = start_pos_ - firstLaneDifference;

    for (int i = 0; i < number_of_lanes_; i++)
    {

        Transform z, y;

        z.scale(i, i);

        int tempLaneNumber = lanes_[i]->GetLaneNumber();
        float tempLaneDirection = lanes_[i]->GetDirection();

        //if lane is in road direction
        if (tempLaneDirection == direction_)
        {
            // send calculated starting point
            *lanes_[i] = Lane(tempLaneNumber,
                               road_number_,
                               intersection_number_[1],
                               firstLanePoint + z.transformPoint(laneDifference),
                               Settings::CalculateDistance(start_pos_, end_pos_),
                               direction_, true);
        } else
        {
            // send starting point + length vector
        }
    }
}
```

```

y.rotate(direction_);
lengthVec = y.transformPoint(Vector2f(0.f, -1.f)) * length_;

*lanes_[i] = Lane(tempLaneNumber,
                   road_number_,
                   intersection_number_[0],
                   firstLanePoint + z.transformPoint(laneDifference) +
lengthVec,
                   Settings::CalculateDistance(start_pos_, end_pos_),
                   (direction_ + 180.f), false);
}
}

Settings::DrawDelete = prevState;

BuildLaneLines();
}

```

מחלקה נתיב - Lane

מחלקה זו מייצגת אובייקט מסווג נתיב. נתיב או האובייקט על גביו נסועות המכוניות, והוא מוכן בתוך כביש.

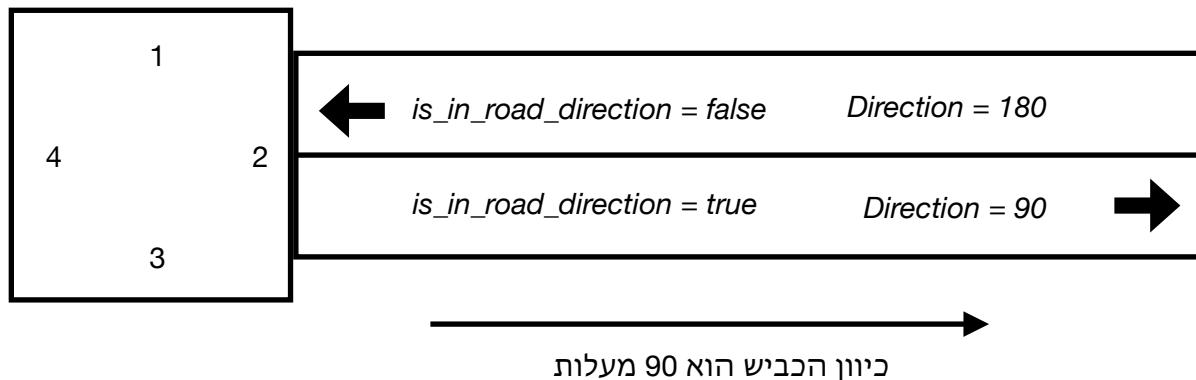
בדומה לכביש ולצומת, מחלקה זו יורשת גם כן ממחלקה RectangleShape שספקה על ידי ספריית SFML.

בנוסף למאפיינים אותם מספקת מחלקת האב RectangleShape, ישנו מאפיינים נוספים כמו:

- האם_נתיב_סגור לתנועה?
- צפיפות_נתיב
- לאיזה_פaza_נתיב_שייך?
- אם_נתיב_בכיוון_הכביש?

בנוגע למאפיין האחרון, שמו האמתי הוא *is_in_road_direction* והוא מסווג בוליאני, מייצג את האפשרות של נתיב להיות באחד משני הכוונים האפשריים בכביש.

במקרה וערך משתנה זה הוא חיובי, יהיה כיוון הנתיב בכיוון הכביש אליו הוא משתייך. במקרה וערך משתנה זה הוא שלילי, יהיה כיוון הנתיב ההפוך לכיוון הכביש, ככלומר 180 מעלות יותר.



המידע שיש לנטיב לגבי המכוניות שנושאות עליו הוא:

- מערך של מספרי זהירות של המכוניות שנושאות עליו ברגע
- מספר המכוניות שנסעו על נתיב זה מתחילה הסימולציה.

לנתיב לא ניתן מידע מעבר למידע זה, כגון מידע על מיקום המכוניות, או מידע של מצביעים על המכוניות שנוסעות עליו, מכיוון למחלקה זו אין גישה למחלקות המכוניות מטעמי היררכיה וסדר.

כלומר, היחסים בין עלי הרכב לבין הנתיבים יכולים להתנהל בשתי דרכים:

1. הנתיב הוא השולט על אובייקט המכונית. יש לו גישה למחלקה כל הרכב, והוא יכול לתת פקודות תנועה לכל רכב שנמצא על גבו.
2. המכונית היא השולטת על עצמה. המכונית יש גישה למחלקה הנתיב, ולא ההפך.

השיטה השנייה היא שנבחרה, מכיוון נתינת שליטה לנתק, שהוא האובייקט האחרון במיקומו במערכת המפה, על אובייקטו כל הרכב, שהוא מרכז הסימולציה, משבשת לחלווטו את יחס הכוחות וסדר הפעולה של התוכנה.

נוסף לכך, בשפות רבות אחרות, לא ניתן ליצור גישה דו-כיוונית בין מחלקות. כלומר, מחלקה *a* לא יכולה ליצור גישה למחלקה *b* בזמן של מחלקה *b* יש גישה למחלקה *a*. זה יוצר מצב של רקורסיה של *include*, שאינה יכולה להתקיין.

אפשר הגבלה זו, גישה של נתיב למחלקה כל הרכב הייתה מונעת מכל הרכב גישה למחלקה הנתיב.

אך מכיוון שמחלקת הנתיב היא האחרונה בהיררכיה מערכת המפה, שכן הכביש יכול לגשת אליה, והצומת יכול לגשת לכביש, והmphella יכולה לגשת לצומת - למחלקה המכונית לא תתאפשר הגישה לכל מחלקות המפה, ויוצר מחסור תקשורת בין כל המפה לבין כל הרכב.

לכן, הוחלט שלכל כל רכב יהיה מצביע לנתק עליו הוא נושא, ומצבע לנתק שהוא עדיף לנסוע עליו - וכך ניתנת לו גישה לכל הפרטים שהוא צריך בצד לבצע נסעה.

עם המידע המועט אך מספיק שיש לנתק על כל הרכב שנוסעים על גבו, מחושבים כמה נתונים:

1. **צפיפות הנתיב** - צפיפות הנתיב מחושבת על ידי כמות המכוניות על פני האורך של הנתיב. צפיפות הנתיב נמדדת לפי מספר המכוניות בכל מטר אחד של נתיב.
2. **זרימה תחבורתית** - עם הספירה הכללית של מספר המכוניות שעוברו בנתיב זה במהלך הסימולציה, ניתן לחשב את זרימת הנציג הממוצעת על ידי חילוק של מספר המכוניות שעוברו בנתיב לדוגמה, ניתן לחשב את זרימת הנציג הממוצעת על ידי חילוק של מספר המכוניות שעוברו בנתיב בזמן הסימולציה חלקי זמן הסימולציה, ונקבל את זרימת הנתיב הממוצעת במהלך הסימולציה.

```

class Lane : public RectangleShape
{
public:

    Lane(int laneNumber,
          int roadNumber,
          int intersectionNumber,
          Vector2f startPosition,
          float length,
          float direction,
          bool isInRoadDirection);
    ~Lane();

    void Update(float elapsedTime);
    void Draw(RenderWindow *window);

    // get
    int GetLaneNumber() { return lane_number_; };
    int GetIntersectionNumber() { return intersection_number_; };
    int GetTotalVehicleCount() { return total_vehicle_count_; };
    int GetRoadNumber() { return road_number_; };
    int GetPhaseNumber() { return phase_number_; };
    bool GetIsBlocked() { return is_blocked_; };
    bool GetIsInRoadDirection() { return is_in_road_direction_; };
    float GetDirection() { return direction_; };
    int GetFrontVehicleId() {
        if (!vehicles_in_lane_.empty())
            return vehicles_in_lane_.front();
        return 0;
    }
    int GetBackVehicleId() {
        if (!vehicles_in_lane_.empty())
            return vehicles_in_lane_.back();
        return 0;
    }
    int GetCurrentVehicleCount() { return vehicles_in_lane_.size(); }
    float GetQueueLength() { return queue_length_; }
    float GetDensity() { return density_; }
    float GetTraversalTime() { return traversal_time_; }
    float GetNormalizedDensity() { return density_ / Settings::MaxDensity; }

    Vector2f GetStartPosition() { return start_pos_; };

    Vector2f GetEndPosition() { return end_pos_ };
    // set
    void Select();
    void Unselect();
}

```

```

void PushVehicleInLane(int vehicleId) {
    vehicles_in_lane_.push_back(vehicleId);
    total_vehicle_count_++;
}
void PopVehicleFromLane() {
    vehicles_in_lane_.pop_front();
}
void SetIsBlocked(bool blocked) {
    is_blocked_ = blocked;
    if (!blocked)
        queue_length_ = 0;
}
void SetPhaseNumber(int phaseNumber) { phase_number_ = phaseNumber; }
void ColorRamp();
void ClearLane() {
    total_vehicle_count_ = 0;
    density_ = 0;
    Unselect();
    vehicles_in_lane_.clear();
}
void SetQueueLength(float distance);

// The count of the overall Lanes that have been created
static int LaneCount;

private:

float calculate_traversal_time();
// Is this intersection block
bool is_blocked_;
// Is this lane the same direction of the parent road
bool is_in_road_direction_;
// ID of the father intersection
int intersection_number_;
// ID of the father road
int road_number_;
// ID if this lane
int lane_number_;
// Total count of vehicles that passed in this lane
int total_vehicle_count_;
// The ID of the phase this lane belongs to
int phase_number_;
// Is this lane currently selected
bool selected_;
// Density of the lane.
// Measured by Car-per-Meter of lane
float density_;
// The length of the queue in this lane;
// calculated in Vehicle class,
// as the distance between the

```

```
// first and the last car in lane with a state of STOP;
float queue_length_;
// the time it will take for the last vehicle int
// the queue to reach the intersection
float traversal_time_;

list<int> vehicles_in_lane_;

Vector2f start_pos_;
Vector2f end_pos_;

float direction_;
float width_;
float length_;

void create_arrow_shape(Transform t);
ConvexShape arrow_shape_;

void create_block_shape();
RectangleShape lane_block_shape_;

DataBox *data_box_;
};
```

דוגמה לפונקציה - ColorRamp

באמצעות נתון הצפיפות של נתיב, ניתן ליצור ויואלייזציה גרפית הפעלתה לפי צבעים. ניתן לקבוע סקלאר בתחום מסוים, שעבור כל ערך אחר ניתן צבע המתאים לו על גבי סולם צבעים נבחר.

לדוגמה, ניתן לקבוע שעבור כל מספר בין 0 ל-1, ניתן צבע על תואם על גבי הסולם, בהתאם למיקום המספר בתחום זה:



בכדי לעשות זאת, נשתמש בשיטת הצבעים המקובלת במחשבים כיום, והוא RGB. שיטה זו מחברת את הצבעים אדום, ירוק וכחול בكمויות ובצורות שונות על מנת ליצור צבע חדש. לכל צבע יש חזק של בין 0 ל-255, וכל שערכו של צבע יהיה גבוה יותר כך יהיה דומיננטי יותר בצבע החדש שיוצר.

ובכן, המטרה היא להשתמש בשיטת RGB בכדי להמיר מספר המוגבל בתחום לצבע, ובהתאם להтенגוט המוח האנושי - צבע המכיל גוון אדום יותר יתאר מצב קיצוני, לא אידיאלי ואילו צבע המכיל גוון כחול יותר יתאר מצב רגוע ולא לחוץ.

ראשית, יש להמיר את צפיפות הנתיב לערך בין 0 ל-1, כאשר 0 המתאר נתיב חופשי לחלוין, ויביא ליצרת צבע כחול, ו-1 המתאר נתיב צפוף מאוד יביא ליצירת צבע אדום.

בכדי לעשות זאת, נבצע נורמליזציה: התאמת ערך לתחום מוגדר. כאמור, התחום המוגדר הוא בין 0 ל-1, לכן נשתמש בפונקציה הבאה:
 x מתאר את הערך הנוכחי, $(x)\min$ את רצף התחום (במקרה זה צפיפות 0), $-(x)\max$ מתאר את

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

תקרת התחום (במקרה זה הוגדר צפיפות של 0.2 מכוניות בכל מטר). נוסחה זו מקבלת את התחום הנוכחי של ערך, וממיר אותה בהתאם לתחום של 0 ל-1.

לאחר קיבלת ערך מנורמל, נשלח אותו לפונקציה `GetHeatMapColor()`, שתחזיר את הצבע המתאים בסולם הצבעים.

הfonקציית **GetHeatMapColor** היא פונקציה המתקבלת ערך מנורמל, ומחזירה ערכי R G B תואמים בהתאם לסולם צבעים מוגדר.

במקרה זה, סולם הצבעים הוא כחול, אדום, והוא נראה כך:



אופן הפעולה של פונקציית **GetHeatMapColor** הוא כזה:

כל הערך המנורמל יהיה גבוה יותר, ערך האדום יהיה גבוה יותר ב-RGB, ואילו ערך הכחול יהיה נמוך יותר.

כל שהערך המנורמל יהיה נמוך יותר, ערך האדום יהיה נמוך יותר וערך הכחול יהיה גבוה יותר ב-RGB.

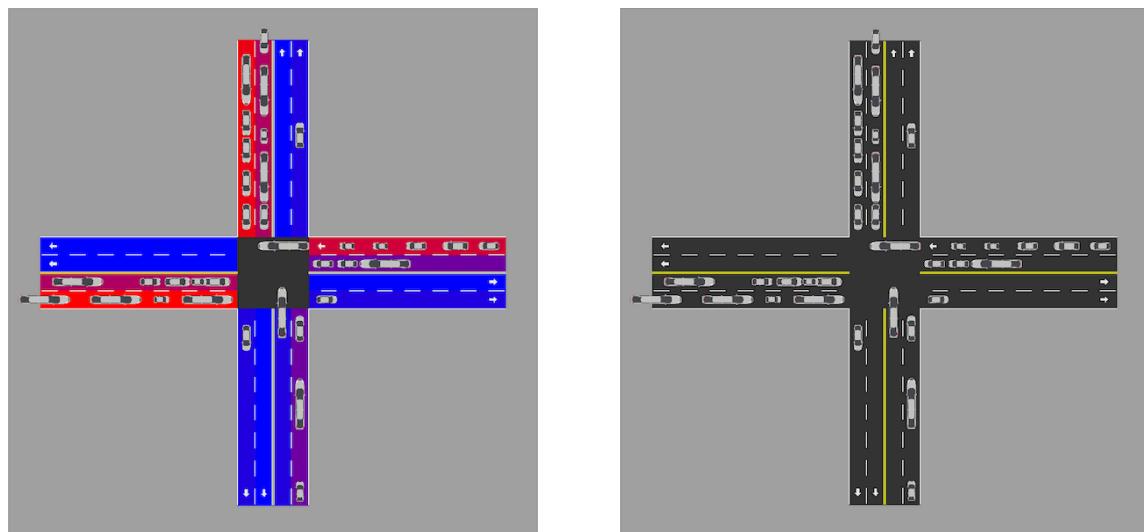
```
void Settings::GetHeatMapColor(float value, float *red, float *green, float *blue)
{
    int aR = 0;    int aG = 0;    int aB=255;    // RGB for our 1st color blue
    int bR = 255;  int bG = 0;    int bB=0;    // RGB for our 2nd color red

    *red    = (float)(bR - aR) * value + aR;      // 255 * value.
    *green = (float)(bG - aG) * value + aG;      // 0.
    *blue   = (float)(bB - aB) * value + aB;      // 255 * (1 - value).
}
```

לאחר קבלת ערכי הRGB הרצויים, נקבע את הנטייה בצבע זה, ונקבל רפרזנטציה צבעונית של צפיפות הנתיבים במפה.

בתמונה הימנית, ניתן לראות מפה רגילה, כאשר אפשרות צביעת הצפיפות כבואה. בתמונה השמאלית, אותה המפה בדיק, כאשר אפשרות צביעת הצפיפות דלוכה.

כמו שניתן לראות בבירור בתמונה השמאלית, נתיבים בהם צפיפות המכוניות גבוהה יותר, נקבע צבע בעל גוון אדום יותר, ובנתיבים בהם הצפיפות נמוכה עד אפסית, נקבע גוון כחול יותר.



קוד הפונקציה ColorRamp

```
/// create a color visualisation of lane density
void Lane::ColorRamp() {

    // normalizing the lane density.
    // density is vehicle-per-meter

    float value = (density_ / Settings::MaxDensity);
    if (value > 1.f) value = 1.f;
    float r, g, b;

    Settings::GetHeatMapColor(value, &r, &g, &b);

    this->setFillColor(Color(r, g, b, 255));
}
```

מחלקה המסלול - Route

מחלקה זו אינה חלק מבניית המפה, אך היא חלק חשוב בನיסיון כל הרכב על גבי המפה. מסלולים הם כל צירופי הנתיבים האפשריים עליהם כל רכב יכול לחצות את המפה.

את המסלולים קובע המשתמש, על ידי קביעת נתיב מקור ונתיב יעד, כאשר עליהם להיות על אותו הצומת.

לאחר שהוגדו כל המסלולים במפה, כלומר המעברים מנתיב לנתיב האפשריים, נבנה מערך של מסלולים בצורה רנדומלית שייצורו מסלול שלם אשר חוצה את המפה, והוא נחשב וניתן לכל רכב שנוצרו כ "הוואות ניוט".

כך נוכל להבטיח שכל כלי רכב יסע במסלול שהוגדר בצורה רנדומלית.

אובייקט מסוג מסלול מחזק 2 דברים בלבד:

1. מצביע לנטיב המקור
2. מצביע לנטיב היעד

כאמור, לאובייקט מסוג מסלול אין שימושות מיוחד, אלה רק כמערך של מסלולים.

קוד המחלקה

```
class Route
{
public:

    Lane *FromLane;
    Lane *ToLane;

    Route(Lane *from, Lane *to);
    ~Route();

    void Draw(RenderWindow *window);
    void ReloadRoute();

    // get
    int GetRouteNumber() { return route_number_; }

    // set
    void SetSelected(bool selected) { selected_ = selected; }

    // The total count of all the routes created this session
    static int RouteCount;

private:
```

```

void BuildRadiusLine();
void BuildLaneLines();

bool selected_;

int route_number_;

vector<Vertex *> lines_;
VertexArray radius_line_;
};


```

דוגמה לפונקציה - `GenerateRandomTrack3`

הרכבת מסלול שלם נעשית בזמן אמיתי בעת ייצור רכב חדש, באופן רנדומלי.

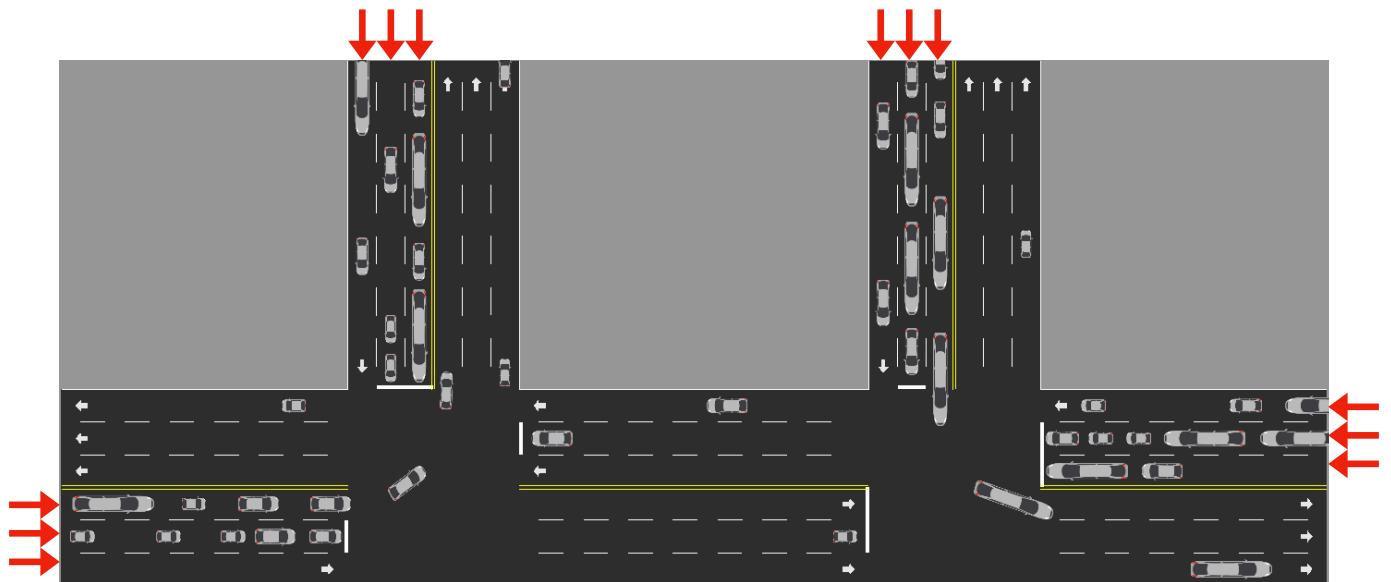
להלן אופן ההרכבה והשימוש של המסלולים:

אובייקט מסוג מפה מחזק את 2 המרכיבים הבאים:

1. `vector<Route *> routes_` - מערך של מצביעים אל כל המסלולים האפשריים במפה.
2. `vector<Lane *> starting_lanes_` - מערך של מצביעים אל כל "נתיבי התחלה" במפה.

"נתיב התחלה" הוא נתיב ממנו ניתן כלי רכב יכול להיכנס אל המפה. בrama הטכנית, נתיב התחלה הוא **נתיב שאיןו לחבר**, אשר כיוונו הוא **נגד כיוון הכביש אליו הוא שייך**. כדי להזכיר, כיוון כביש שאינו לחבר הוא לפני חוץ הצומת.

בתמונה הבאה, כל הנתיבים המסומנים בחץ אדום הם "נתיבי התחלה":

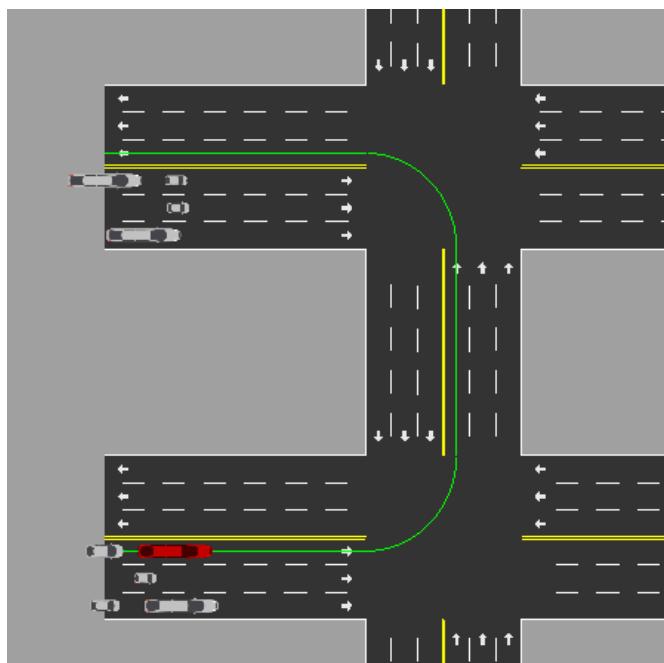


³ פונקציה זו שיכת למחלקה `Map.hpp`, אך היא מבטאת את מימוש אובייקט המסלול בצורה הטובה ביותר.

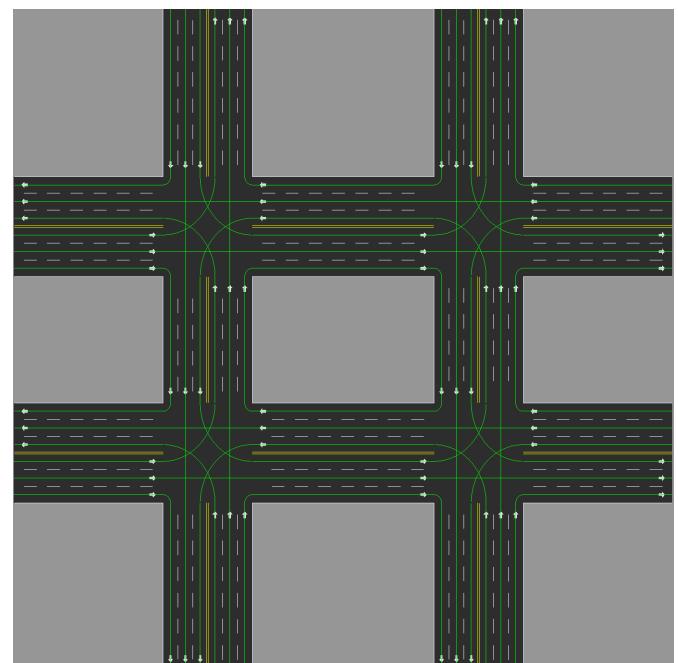
כאמור, תפקידו של פונקציה `GenerateRandomTrack` הוא ליצור מסלול שלם בצורה רנדומלית.

להלן אופן הבניה של מסלול שלם:

- > צור מערך נתיבים חדש `Arr`
- > מצא נתיב התחלה `N`
- > מצא מסלול התחלה `X`, אשר נתיב המקור שלו הוא נתיב `N`.
- > כל עוד יש למסלול `X` מסלול המשך (מסלול שנמצא התחילה שלו הוא נתיב הסוף של `X`)
 - > הוסף ל-`Arr` את נתיב התחלה של `X`
 - > שמור את נתיב הסוף של `X` ב-`L`
 - > מצא והחלף את מסלול `X` במסלול המשך שלו.
 - > הוסף ל-`Arr` את נתיב `L`
 - > החזר את `Arr`



דוגמה למסלול שלם ייחודי.



כל המסלולים השלימים האפשריים במפה מסוימת.

```

/// randomly generate a track
list<Lane *> *Map::GenerateRandomTrack() {
    // find a random starting point
    Lane *l = GetPossibleStartingLane();
    if (l == nullptr)
    {
        cout << "no starting lanes available." << endl;
        return nullptr;
    }
    // find a starting route from starting lane
    Route *r = GetPossibleRoute(l->GetLaneNumber());

    if (r == nullptr)
    {
        cout << "no routes available. please add them to the map" << endl;
        return nullptr;
    }

    // while new routes to append are available
    // new routes will be searched starting from the previous route end
    list<Lane *> *track = new list<Lane *>();
    Lane *lastLane = nullptr;

    while (r != nullptr)
    {
        track->push_back(r->FromLane);
        lastLane = r->ToLane;
        r = GetPossibleRoute(r->ToLane->GetLaneNumber());
    }
    if (lastLane != nullptr)
    {
        track->push_back(lastLane);
    }

    return track;
}

```

מחלקת הפאזה - Phase

מחלקה זו היא אחת המחלקות החיוניות באופן פועלות מנוע הסימולציה, והיא ה-”סוכן” במערך הבינה המלאכותית אשר יפורט בהמשך.

אובייקט מסווג פאזה ”שולט“ על מערך של נתיבים - הוא המורה להם מתי להיפתח, ומתי להיסגר.

מי שיצפה בפעולות צומת מרומר, ישים לב שרמזוריים אינס פועלים לחוד - אלה בקבוצות. בכל פעם שרמזור נדלק, סביר להניח שרמזור נוסף נוסף ידלק ביחד איתו, וזה סימן לכך ש-2 הרמזורים האלה באותה ”קבוצה“, ושיכים לאותה פאזה. חשוב לציין לא יכולות לפעול באותו הזמן, שכן במצב כזה יש להגדיר אותם כאותה הפאזה.

לשם הגדרת הפאזה, ראיית יש להגדיר כמה מושגים:

- **זמן מחזור שלם** - זהו סכום זמני הפעולה של כל הפאות בצומת.
- **זמן מחזור פאזה** - הזמן שפאזה נשארת דלוכה.
- **סדר פאות** - הסדר בו הפאות מסודרות במחזור השלם.
- **רמזור** - אמצעי להציג סטוס הפאזה. רמזור רק מורה למכוניות איך לפעול.

*אופן קביעת המדיניות וניתוח הנ吐נים יתואר בהרחבה בפרק העוסק **בבינה המלאכותית**.

לאובייקט מסווג פאזה המאפיינים הבאים:

- מספר זיהוי
- מספר הרמזוריים הכלפפים לפאזה זו
- האם פאזה זו פתוחה? (אם יש אויר יירוק?)
- הזמן שעבר מאז פתיחת הפאזה
- זמן מחזור פאזה
- מערך של מצביעים לנטיים הנשלטים על ידי פאזה זו
- מערך של רמזוריים הכלפפים לפאזה זו
- **אפשרות הנתיב המקסימלית בפאזה**
- אורך תור המכוניות הארוך ביותר בפאזה
- הזמן שיקח לתור הכוי אורך לחצות את הצומת

מ-3 המאפיינים, ובאמצעות הבינה המלאכותית, נבנה מאפיין שנקרוא **”עדיפות פאזה“**.
באמצעות מאפיין זה יכולת על מדיניות הפאות.
הסביר בפרק על מחזוריים.

```

class Phase
{
public:
    Phase(int phaseNumber, int cycleNumber, float cycleTime);
    ~Phase();

    void Draw(RenderWindow * window);
    void Update(float elapsedTime);
    void ReloadPhase();

    // add entities
    Light * AddLight(int lightNumber, Road * parentRoad);
    void AddLane(Lane * lane);

    // get
    int GetPhaseNumber(){return phase_number_;}
    int GetCycleNumber(){return cycle_number_;}
    bool GetIsOpen(){return open_;}
    float GetCycleTime(){return cycle_time_;}
    vector<Light*> * GetLights(){return &lights_;}
    vector<Lane*> * GetAssignedLanes(){return &lanes_;}
    void GetInputValues(vector<double> & inputValues);
    float GetMaxQueueLength();
    float GetMaxLaneDensity();
    float GetMaxTraversalTime();
    float GetPriorityScore() {return priority_;}

    // set
    void Open(){open_ = true;}
    void SetCycleTime(float cycleTime){cycle_time_ = cycleTime;}
    bool UnassignLane(Lane * lane);
    void SetPhasePriority(float points) { priority_ = points;}

    // The total count of all the phases created this session
    static int PhaseCount;

private:
    // ID of this phase
    int phase_number_;
    // Id of the cycle it belongs to
    int cycle_number_;
    // Number of lights that represent this phase
    int number_of_lights_;
    // Is this phase active and open
    bool open_;

    // The time that the phase has been open over
}

```

```
float open_time_;
// Cycle time of this phase
float cycle_time_;
// The points given to a phase according to its need to be open
float priority_;

// the current state of this phase
LightState state_;

vector<Light*> lights_;
vector<Lane*> lanes_;

};
```

מחלקה המחזור - Cycle

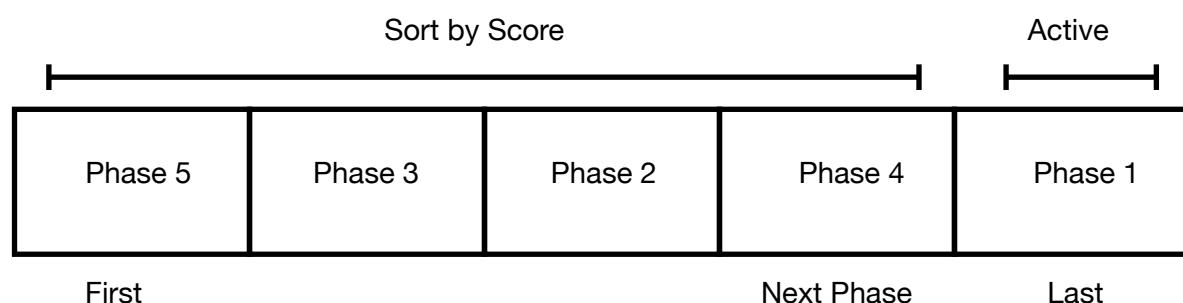
מחלקה זו מייצגת מחזור של פאות. מחלקה זו היא השולטת על כל הפאות במערכת, כאשר כל מחזור מחזיק רשימה של פאות.

כל מחזור יכול להשתיק לצומת וליצג אותה, או ליצג מפה שלמה או כמה צמתים.

מחלקת המחזור מאפשרת להריץ מספר מחזורי פאות שונים ואוטונומיים אותן אותה המפה. כמובן, ניתן יהיה לפתח 2 פאות שונות או יותר באותו הזמן, בתנאי שהיו שייכים למחזורים שונים.

מחלקת המחזור אחראית לעדכן את מצב הפאות, והיא מחליטהמתי להפעיל כל פאה ולכמה זמן.

רשימת הפאות מסודרת בצורה כזו:



הרשימה מסודרת לפי נקודות העדיפות של הפאות.

הפאה שמקומה אחרון ברשימה, כאמור זו שיש לה הכי הרבה נקודות עדיפות, תופעל ראשונה. בזמן הפעולה של פאה זו, שאר הפאות שמאחוריה ימשכו להשתנות וכן גם נקודות העדיפות שלהן, וכך פאות אלו יסודרו לפי עדיפות בכל רגע שהפאה הפעילה פתוחה.

ברגע שהפאה הפעילה נסגרת, הוא מוחלפת על ידי הפאה שנמצאת מקום לפניה ברשימה, שכן היא הבאה הכי עדיפה ברשימה.

וכך הרשימה זה מסודרת ומודכנת במהלך פעילות המערכת, ובतיבחה שפאות יפעול לפי סדר העדיפות שלהם בתוך מחזור.

קוד המחלקה

```
class Cycle
{
public:

    Cycle(int cycleNumber, Intersection * intersection = nullptr);
    ~Cycle();

    void Update(float elapsedTime);
    void Draw(RenderWindow * window);
    void ReloadCycle();

    Phase * AddPhase(int phaseNumber, float cycleTime);

    int GetCycleNumber(){ return cycle_number_; }
    Phase * GetPhase(int phaseNumber);
    vector<Phase *> *GetPhases() { return &phases_; }
    Intersection * GetIntersection(){ return intersection_; }

    static int CycleCount;

private:

    void cycle_phases();
    void calculate_priority();

    int cycle_number_;
    int number_of_phases_;

    vector<Phase *> phases_;
    Intersection * intersection_;

    vector<double> input_values_;
    vector<double> output_values_;
};

};
```

קוד פונקציה סידור ורשימת הפאות

```
/// cycle the phases by the phase array order.  
// all the phases but the active one are constantly evaluated  
// and sorted by the score they have been given by the NN  
// when phase is finished, it gets swapped with the previous one in the array  
// to advance and start the next phase in order.  
  
void Cycle::cycle_phases() {  
    // if cycle has a minimum of 2 phases  
    if (number_of_phases_ >= 2)  
    {  
        // when current phase is closed, advance to next phase and open it  
        if (!phases_.back()->GetIsOpen())  
        {  
            Phase *backPhase = phases_[number_of_phases_ - 1];  
            phases_[number_of_phases_ - 1] = phases_[number_of_phases_ - 2];  
            phases_[number_of_phases_ - 2] = backPhase;  
  
            phases_[number_of_phases_ - 1]->Open();  
  
            //Settings::NeuralNetwork.printNet();  
        }  
        // constantly sort the list by their priority score  
        else  
        {  
            // calculate the priority of each phase  
            calculate_priority();  
            // sort(arr[0:-2])  
            partial_sort(phases_.begin(), phases_.end() - 1, phases_.end() - 1,  
            compare_priority);  
        }  
    }  
}
```

לצורך חישוב נקודות העדיפות של כל פאה, נעשו שימוש בפונקציה `calculate_priority()`. פונקציה זו היא הפונקציה שמדרגת את עדיפות הפאות באמצעות **בינה מלאכותית**. פונקציה זו תתואר בהרחבה בפרק על הבינה המלאכותית.

מחלקה הרמזור - Light

כפי שצוין במחלקה הפהזה, הרמזור אינו דורך דרך להציג את מצב הפהזה לכל נוסעים הנתיבים השיכים אותה פזה, ולא מכיל פונקציונליות חיונית למערכת.

הرمزור הוא פריט ויזואלי בלבד, אותו ניתן להוסיף למטרות המראה של פועלות המערכת.

את הרמזור ניתן "להדביק" לכביש מסוים, והוא יימקם לצד ימינו ליד פגישתו עם צומת.



כמו לפזה, לרמזור שלושה מצבים: אדום, כתום וירוק. מצבים אלה מתעדכנים על ידי הפהזה אליו כפוף הרמזור.

קוד המחלקה

```
class Light : public RectangleShape
{
public:
    Light(int lightNumber, int phaseNumber, Road *parentRoad);
    ~Light();

    void Draw(RenderWindow *window);
    void Update(float elapsedTime);

    // get
    int GetPhaseNumber() { return phase_number_; }
    int GetLightNumber() { return light_number_; }
    Road *GetParentRoad() { return parent_road_; }

    // set
    void SetState(LightState state) { state_ = state; }
    void UpdatePosition();

    static int LightCount;
private:
    // ID of this light
    int light_number_;
    // ID of the Phase this light represents
    int phase_number_;

    // The current state of the light.
    LightState state_;
    Road *parent_road_;
    DataBox *data_box_;
    vector<CircleShape *> circles_;
};
```

מנוע הסימולציה

simulator

• DataBox.cpp
• DataBox.hpp
• Engine.cpp
• Engine.hpp
• Set.cpp
• Set.hpp
• Settings.cpp
• Settings.hpp
• Simulation.cpp
• Simulation.hpp
• Vehicle.cpp
• Vehicle.hpp

פרק זה עוסק במחוקות הקשורות להפעלת הסימולציה, והשימוש במפה ובכליים שתוארו בפרק הקודם.

בפרק זה יתואר מנגנון הסימולציה, סדר העבודה הכללי של מערכת, התנהגות כלי הרכב, וכו'.

על מנת לשמר על דיקוק ותיאום בין חלקי הרובים של המערכת, כל האלמנטים בסימולציה הונם במפה והונם במנוע הסימולציה, עובדים על פי השיטה הבאה:

1. **Input** - קבלת מידע חדש

2. **Update** - וערכו כל האובייקטים בצורה המתאימה להם, בהתייחסות למידע החדש שהתקבל בשלב הקודם.

3. **Draw** - ציור על האובייקטים על גבי הkanous.

בדרך כלל, נהוג לקבוע תדריות מסוימת (בד"כ 60) הנקראת FPS (Frames-per-Second), כלומר תמונות בכל שנייה, ועל פיה בכל שנייה נערך על הרץ המצוין לעיל מספר פעמיים מסוימים.

הבעיה בשיטה זו, היא שהיא עובדת טוב לצרכים שאינם מצריים דיקוק גובה ותדריות חישוב גובה מאוד (עד 1000 פעמיים בשניה), ולכן חישוב, עדכון וציור המערכת 60 פעמיים בשניה מספק את צרכי המערכת.

השלב הכוי "יקר" במשאבי המחשב, הוא שלב הציור, שבדרך כלל מטופל על ידי כרטיס מסך במחשב.

אך למחשבים שאינם כרטיס מסך, כמו למחשב שלי ולמחשבים רבים אחרים, חישוב שלב זה מתווסף יחד עם שלב הקלט והעדכון על גבי המעבד, ויכול ליצור עומס.

עקב הרכבים המינוחדים של המערכת, שהם דיקוק ומהירות חישובים גבוהה (שלב 2), אך גם כן הפחיתה מחיר המשאבי של שלב הציור, נאלצתי להפריד את התהיליך זהה ל- 2 מוחזרים נפרדים:

1. **המחזר החישובי** - מוחזר זה משלב את שלבים 1 ו-2, שמכיוון שהם צריכים להיות מסונכרנים בצורה מושלמת אין יכולת להפרידם. מוחזר זה קורה בתדריות גבוהה המוגדרת מראש (בין 120 ל-1000 פעמים בשנית)
2. **המחזר הציורי** - מוחזר זה אחראי אך ורק על שלב ה-3, שהוא שלב הציור. מכיוון שציור המערכת אינו קשור בשום צורה לתפקיד המערכת, אלה אפשר למשתמש לצפויות במתרחש במוחזר החישובי, ניתן להפרידו בצורה מוחלטת מהמחזר החישובי. עקב ה"מחיר" המשאבי הגבוה של מוחזר זה, יש לו תדריות נמוכה המוגדרת מראש. (בין 0 ל- 1000 פעמים בשנית, ממוצע על בין 30 ל-60).

```
/// do the game cycle (input->update)
/// draw and display are separate for different fps
/// this allows running logic cycle in high rate -> better accuracy
// and running draw cycle in low rate -> better performance

void Engine::logic_cycle() {

    input();
    update((floatlogic_timer_.interval() / 1000.f));
}

/// do the rest of the game cycle independently
void Engine::draw_cycle() {
    render();
    display();
}
```

* קוד מחלוקת תפורט לעומק בעמוד הבא Engine.cpp. המחלקה

מחלקה מנוע סימולציה - Engine

מחלקה זו היא האחראית על רוב פעילות המערכת, והיא משקיפה כלל התהליכיים ודווגת לתיאום מוחלט בין כל מרכיבי הסימולציה.

ניתן ליזוף מנוע סימולציה אחד עבור כל מערכות.

המנוע אחראי על קיומם של הדברים הבאים:

- בניית והחזקה של חלון הסימולציה, הפעלת מצלמה ומיני-מפה (minimap).
- העברת מידע וΚαιוורדינציה בין חלק הסימולציה למפה לחלק משק המשמש.
- הפעלת המחזוריים:
 - קבלת קלט מהמשתמש (קלט הקשור לחalon הסימולציה) כמו גירירה ובחירה של אובייקטים
 - עדכון כל חלקי מערכת הסימולציה, כולל עדכון המפה וככל כלי הרכיב
 - ציור כל מרכיבי המפה וכל הרכיב בחalon הסימולציה
 - שימרת מפה, טעינת מפה וניהול
 - שימרת סימולציות, טעינת סימולציות וניהול סימולציות (מחיקה הוספה ועדכון)
 - ייצור כל כלי הרכיב
- החזקה והפעלה של הבינה המלאכותית (יפורט לעומק בפרק "בינה מלאכותית")

קוד המחלקה

```
class Engine : public QSFMLCanvas
{
    Q_OBJECT
public:
    Engine(QWidget *Parent);
    ~Engine() {};

    bool RunSet(int vehicleCount = 1000, int generations = 10);
    bool RunDemo(int simulationNumber);
    Set *AddSet(int setNumber, int vehicleCount, int generations);

    // get
    Vector2f GetSnappedPoint(Vector2f point);
    Vector2f DrawPoint(Vector2f position);
    vector<Simulation *> *GetSimulations();
    Simulation *GetSimulation(int simulationNumber);
    Set *GetSet(int setNumber);

    // set
    void SetSnapToGrid(bool snapToGrid) { this->snap_to_grid_ = snapToGrid; }
    void SetView();
```

```

void BuildGrid(int rows, int cols);

void UpdateView(Vector2f posDelta = Vector2f(0, 0), float zoom = 0);
void SaveMap(string saveDirectory);
void LoadMap(string loadDirectory);
void SaveSets(string saveDirectory);
void LoadSets(string loadDirectory);
void ResetMap();
void ClearMap();
bool AddVehicleRandomly();
void ResizeFrame(QSize size);
bool DeleteSimulation(int simulationNumber);

Map *map;
signals:
void SimulationFinished();
void SetFinished();

private:
void on_init() override;

void logic_cycle() override;
void draw_cycle() override;
void render();
void input();

void render_minimap();
void update_shown_area();
void update(float elapsedTime);
void add_vehicles_with_delay(float elapsedTime);
void check_selection(Vector2f position);
void set_minimap(float size, float margin);

// Snap click points to set grid
bool snap_to_grid_;
Grid snap_grid_;

// The simulator camera
View view_;
// The minimap camera
View minimap_;

// The actual view position
Vector2f view_pos_;
// The current view position while dragging mouse
Vector2f temp_view_pos_;

RectangleShape minimap_bg_;

```

```

RectangleShape shown_area_index_;
CircleShape click_point_;

int number_of_sets_;
// an array of simulation sets
vector<Set *> sets_;
vector<double> target_results_;
};

```

דוגמה 1 לפונקציה - LoadMap

בפרויקט זה הוחלט לשמר מידע כמו מפות וסימולציות בפורמט JSON - JavaScript Object Notation.

זהו פורמט שאינו מקובל בסביבות פיתוח כמו C++, אך מכיוון שהוא פורמט שאני מכיר ואוהב את פשטותו, החלטתי לכתה על האתגר לשלב אותו לסביבת הפיתוח שלי.

שילוב זה נעשה באמצעות מחלוקת החיזונית json/nlohmann/json⁴ אותה מצאת ב-GitHub. מחלוקת זו דואגת לשלב את עולם ה-JSON לתוך עולם ה-C++ בצורה פשוטה ואלגנטית.

שיטת השמירה היא שמירת כמה נתונים מינימלית על כל פריטי המערכת, על מנת ליצור "הוראות יצרן" על פיהם המערכת יכולה לפעול בצד לבנות מפה זהה בעת הצורך.

לדוגמה, כל נתיב בקובץ של מפה שמור בצורה הבאה:

```

"lanes": [
  {
    "id": 1,
    "is_in_road_direction": false,
    "road_number": 1
  },

```

על ידי שמירה וטעינה של מפות בסדר התואם את הסדר היררכי של מערכת המפה, אותו ניתן לראות בעמוד 7, נוכל להבטיח שלאחר הייצור של מפה ריקה חדשה תוכל המערכת להוסיף צומת למפה מסוימת לפי מספר הזיהוי הייחודי שלה, ולאחר מכן נוכל ליצור כביש המתחבר לצומת על פי מספר הזיהוי שלו בלבד, וכן האלה... עד שמנוע הסימולציה ייצור את כל האלמנטים המרכיבים את המפה.

*תהליך הטעינה והשמירה של סימולציות מתבצעת באופן זהה.

<https://github.com/nlohmann/json> ⁴

```

/// build a map using instructions from a given json file
void Engine::LoadMap(const string loadDirectory) {
    // first, delete the old map.
    ResetMap();

    try
    {
        json j;
        // open the given file, read it to a json variable
        ifstream i(loadDirectory);
        i >> j;

        // build intersections
        for (auto data : j["intersections"])
        {
            map->AddIntersection(data[ "id" ],
                                   Vector2f(data[ "position" ][0], data[ "position" ]
[1]));
        }

        // build connecting roads
        for (auto data : j["connecting_roads"])
        {
            map->AddConnectingRoad(data[ "id" ],
                                     data[ "intersection_number" ][0],
                                     data[ "intersection_number" ][1]);
        }

        // build roads
        for (auto data : j[ "roads" ])
        {
            map->AddRoad(data[ "id" ],
                          data[ "intersection_number" ],
                          data[ "connection_side" ],
                          Settings::DefaultLaneLength);
        }

        for (auto data : j[ "lanes" ])
        {
            map->AddLane(data[ "id" ], data[ "road_number" ],
data[ "is_in_road_direction"]);
        }

        for (auto data : j[ "routes" ])
        {
            map->AddRoute(data[ "from" ], data[ "to" ]);
        }
    }
}

```

```

        for (auto data : j[ "cycles" ])
    {
        int interId = data[ "attached_intersection_id" ];
        map->AddCycle(data[ "id" ], interId);
    }

    for (auto data : j[ "phases" ])
    {
        map->AddPhase(data[ "id" ], data[ "cycle_id" ], data[ "cycle_time" ]);
    }

    for (auto data : j[ "assigned_lanes" ])
    {
        map->AssignLaneToPhase(data[ "phase_number" ], data[ "lane_number" ]);
    }

    for (auto data : j[ "lights" ])
    {
        map->AddLight(data[ "id" ], data[ "phase_number" ],
data[ "parent_road_number" ]);
    }

    cout << "map has been successfully loaded from '" << loadDirectory << "'.
           << endl;
}

catch (const std::exception &e)
{
    cout << "Could not load map from this directory." << endl;
    cout << e.what() << endl;
}
}

```

דוגמה 2 לפונקציה - ()

כפי שצוין קודם לכן, לכל אובייקט במערכת יש פועלת update משלו. קריאת update חייבת להיות מסונכרנת - זאת על מנת לשמור על קואורדינציה מושלמת בין כל מרכיבי המערכת.

ב כדי לעשות זאת, קיימת מחלוקת (Update) האחראית על עדכון כל האובייקטים במערכת, שכן מחלוקת Engine בעלת גישה לכל האלמנטים המרכיבים את הסימולציה.

בצורה פשוטה מאוד, פונקציה זו מקבלת את הזמן שעבר מאז הקריאה האחורה של הפונקציה הזאת, וקוראת לכל הפונקציות update של כל האלמנטים במערכת.

בסביבת עבודה אחרת כגון #C, שיטה זו לא הייתה נבחרת, אלה שיטת ה-Events.

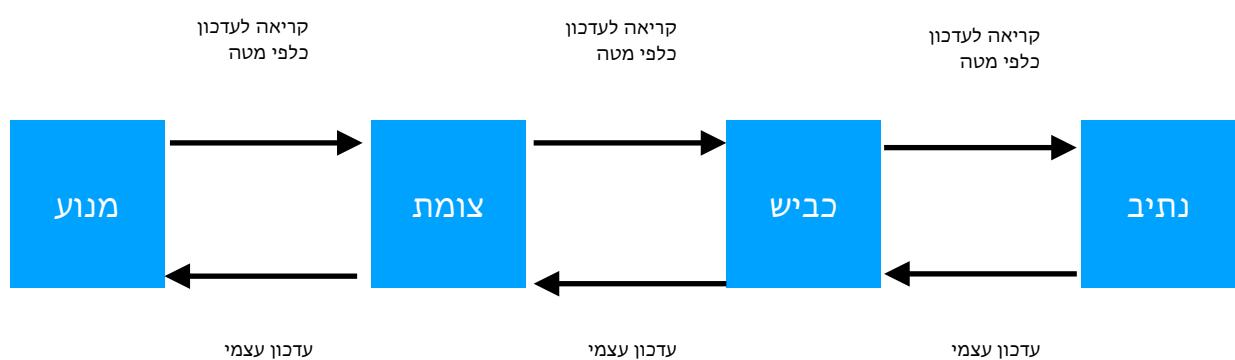
שיטת ה-Events היא שיטה על פיה מחלוקת מקשות את הפונקציות הפרטיות שלhn ל"AIROU" כללי שלו. במקרה ש"AIROU" זהה קורה, כל הפונקציות שמקשות לAIROU זה מופעלות גם כן.

בסביבת Qt קיימת גם כן מערכת דומה - Signal ו-Slots, אך בחרתי לא להשתמש בשיטה זו מ-2 סיבות:

1. הסימולציה משתמשת ברובה בספריית SFML, ורקטי לשמר על הקוד מסודר ולערב כמה פcharted מספריית Qt לתוך מערכת הסימולציה, שכן אופן הכתיבה והשימוש ב-2 ספריות אלו שונות.
2. מערכת ה-Signal ו-Slots של Qt אינה ישרה מבחינות משאבים, ובמערכת בה תדיות העדכון היא עד 1000 פעמים בשניה, שימוש בשירות קריאות לפונקציה ב- C++ טבעי מהירר בהרבה.

הקריאה לפונקציות העדכון היא בצורה שרשורת התואמת להיררכיה המערכת, ולתרשים שבעמוד 6, אך ביצוע העדכון הוא דווקא בדרך חוזרת מהקריאות. כלומר, כל אובייקט קורא לעדכון כל האובייקטים שייכים לו, ורק אז מעדכן את עצמו.

אופן העדכון נראה כך:



```

/// update all the engine's objects
void Engine::update(float elapsedTime) {

    map->Update(elapsedTime);

    // deploy vehicles if needed
    if (Vehicle::VehiclesToDeploy > 0)
    {
        add_vehicles_with_delay(elapsedTime);
    }

    for (Vehicle *v : Vehicle::ActiveVehicles)
    {
        v->Update(elapsedTime);
    }

    //clear all cars to be deleted
    Vehicle::ClearVehicles();

    if (Settings::DrawFps)
        cout << "FPS : " << 1000.f / elapsedTime << endl;

    // follow the selected car
    if (Settings::FollowSelectedVehicle && Vehicle::SelectedVehicle != nullptr)
    {
        view_pos_ = Vehicle::SelectedVehicle->GetPosition()
            - Vector2f(map->GetSize().x / 2, map->GetSize().y / 2);
        temp_view_pos_ = view_pos_;
        SetView();
    }

    if (Set::SetRunning)
    {
        for (Set *s : sets_)
        {
            // when an update on a set returns true
            // it means that a simulation has finished
            if (s->Update(elapsedTime) == true)
            {
                if (s->IsFinished())
                {
                    SetFinished();
                } else
                {
                    SimulationFinished();

                    // set the new score as result
                }
            }
        }
    }
}

```

```
    float result = s->GetLastSimulationResult();

    Settings::NeuralNetwork.printWeights();

    Settings::NeuralNetwork.setActualResults(result);

    // back propogate on default target value (1.0)
    Settings::NeuralNetwork.backProp(target_results_);

    if (Settings::DrawNnProgression)
    {
        cout << "Sim no. " << s->GetGenerationsSimulated() + 1
            << " result :" << result << " avg. error :"
            << Settings::NeuralNetwork.getRecentAverageError()
            << endl;
    }
}

}

}

}
```

קוד לדוגמה 3 - `set_minimap`, `render_minimap`

כפי שצוינו, אחד מתפקידיה הרבים של מחלוקת מנוע הסימולציה הוא לצייר את המפה. ב כדי לצייר את המפה, נעשה שימוש באובייקט `View`. אובייקט זה מובנה בתוך מחלוקת המנווע, שיווק מחלוקת `RenderWindow` של `SFML`.

אובייקט `View` מדומה פעילות של מצלמה: ניתן לעשות `Zoom`, ניתן לשנות את המיקום עליו מסתכלים בקנוווס.

בנוסף לציור המפה, ישנה האפשרות לצייר גם מיני-מפה (`minimap`).

ב כדי ליצור את המיני-מפה, נעשה שימוש ב-2 כלים הנינתנים על ידי `SFML`:

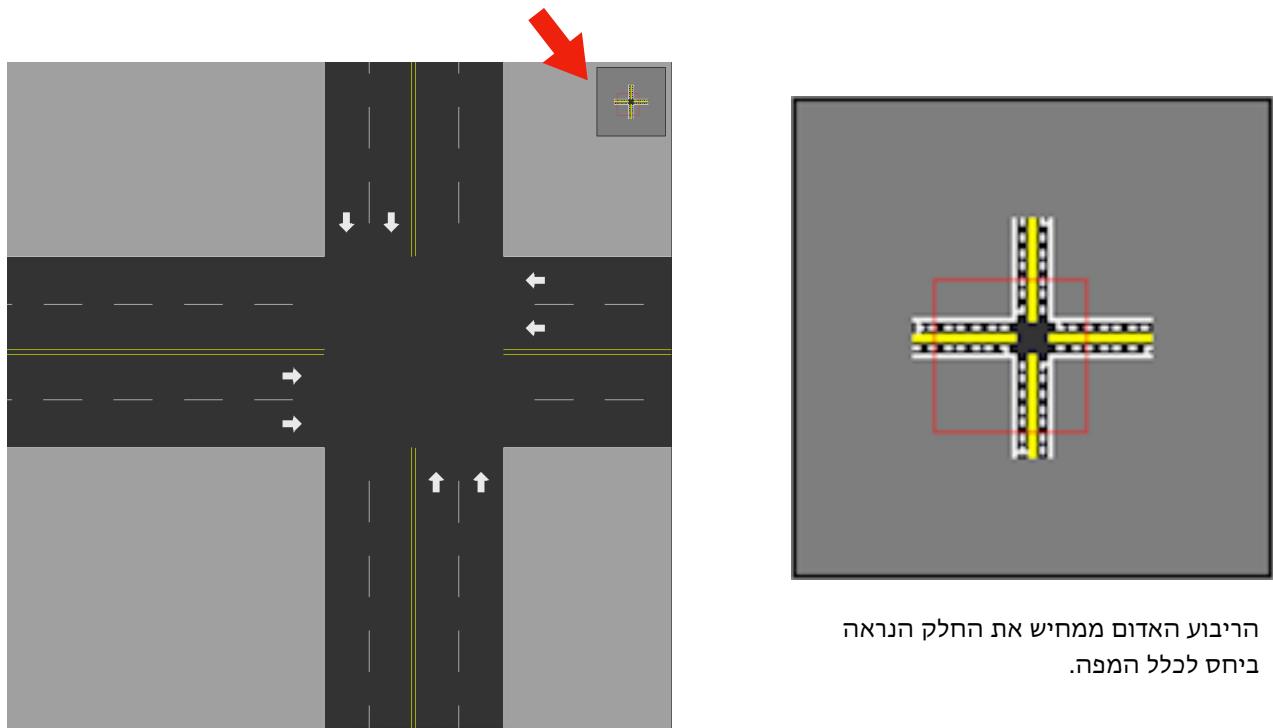
1. `View` - מצלמה המאפשרת "חופש תנועה" ויזואלי מעל המפה.
2. `Viewport` - "חלון ראייה" המהווה מין מסך להקרנת תוכן המצלמה.

ראשית, ה-`Viewport` מוגדר כריבוע קטן בגודל ובמקום מוגדר בחילון, והוא ה"מסך" עליו תוקרנו המיני-מפה.

שנית, נוספת מסגרת אדומה בתוך המיני-מפה המתארת איזה חלק מהמפה העיקרית "מצולם" ביחס למפה כולה.

לאחר שנוצרה המיני-מפה, המנווע מציר את המפה פעמיים:

1. ציור המפה על גבי ה-`View` הראשי, יחד עם כל שאר האלמנטים (כלי רכב, מסלולים, רמזורים...).
2. ציור המפה בלבד על גבי ה-`View` של המיני-מפה, וציור המסגרת האדומה המתארת את הנראה במפה הגדולה.



הרכיב האדום ממחיש את החלק הנראה ביחס לכלל המפה.

קוד הפקונקציה - `set_minimap()`

```
/// build the minimap
void Engine::set_minimap(float size, float margin) {
    // minimap viewPort setup
    minimap_.reset(sf::FloatRect(0,
        0,
        Settings::DefaultMapWidth,
        Settings::DefaultMapHeight));
    minimap_.setSize(Vector2f(size, size));
    minimap_.setViewport(sf::FloatRect(
        1.f - (size / this->width()) - (margin / this->width()),
        margin / this->height(),
        size / this->width(),
        size / this->height()));
    float zoomFactor = Settings::DefaultMapWidth / size;
    minimap_.zoom(zoomFactor);
    float outlineThickness = 30;

    // background setup
    minimap_bg_ =
        RectangleShape(Vector2f(Settings::DefaultMapWidth - outlineThickness * 2,
                               Settings::DefaultMapHeight - outlineThickness *
2));
    minimap_bg_.setPosition(outlineThickness, outlineThickness);
    minimap_bg_.setFillColor(Color(110, 110, 110, 220));
    minimap_bg_.setOutlineColor(Color::Black);
    minimap_bg_.setOutlineThickness(outlineThickness);

    shown_area_index_ = RectangleShape(Vector2f(0, 0));
    shown_area_index_.setOutlineColor(Color(255, 0, 0, 100));
    shown_area_index_.setOutlineThickness(30.f);
    shown_area_index_.setFillColor(Color::Transparent);
    update_shown_area();
}
```

קוד הפונקציה - render_minimap()

```
/// drawing the minimap is drawing everything but the vehicles and the grid,
on a smaller scale
void Engine::render_minimap() {
    // Draw the minimap's background
    this->draw(minimap_bg_);

    // Draw the map
    this->map->Draw(this);

    // Draw the click index
    if (Settings::DrawClickPoint)
        this->draw(this->click_point_);

    // Draw the shown area index
    this->draw(shown_area_index_);
}
```

מחלקה כלי הרכב - Vehicle

מחלקה זו מייצגת את אובייקט כלי הרכב.

מחלקה זו מייצרת, מאחסנת, מעדכנת, ומוחקת את כל כלי הרכב במערכת, ככלומר מחלוקת זו אוטונומית לוחותין.

כלי רכב יורשים ממחלקה RectangleShape מספרית SFML, המורישה לו מאפיינים בסיסיים כמו מיקום, גודל, טקסטורה, וכיוון.

יצירת כלי רכב

כלי רכב נוצרים לפי בקשת מנוע הסימולציה. עבור כל סימולציה, ישנו ביקוש מסוים לכמות מכוניות, בתדרות ייצור מסוימת.

התדרות של הייצור נשלטת על ידי מנוע הסימולציה, באמצעות פונקציות אלו:

קוד הפונקציה - add_vehicles_with_delay()

```
/// deploy vehicles in a time controlled manner
void Engine::add_vehicles_with_delay(float elapsedTime) {
    static float totalElapsedTime = 0;

    totalElapsedTime += elapsedTime * Settings::Speed;

    if (totalElapsedTime > (Settings::VehicleSpawnRate))
    {
        AddVehicleRandomly();
        totalElapsedTime -= Settings::VehicleSpawnRate;
        Vehicle::VehiclesToDeploy--;
    }
}
```

פונקציה זו מוסיפה מכונית חדשה במסלול שיוצר ונדומלית (באמצעות הפוקנץיה הבאה) בכל זמן מוגדר. אם הזמן שהחצבר מאז היצירה הקודמת עולה על זמן היצירה שהוגדר מראש, צור מכונית חדשה ואפס את הזמן שהחצבר.

קוד הפונקציה - AddVehicleRandomly()

```
/// add a vehicle at a random track
bool Engine::AddVehicleRandomly() {

    list<Lane *> *track = map->GenerateRandomTrack();

    if (track != nullptr && !track->empty())
    {

        int randomIndex = 0;

        if (Settings::MultiTypeVehicle)
            randomIndex = rand() % 4;
        return (Vehicle::AddVehicle(track,
                                     this->map,
                                     static_cast<VehicleTypeOptions>(randomIndex))
                != nullptr);
    } else
    {
        cout << "Could not add a new vehicle as tracks could not be generated." <<
    endl;
        return false;
    }
}
```

פונקציה זו מייצרת מסלול נסיעה רנדומלי כפי שתואר בפרק "מחלקת מסלול - Route".

סוג כל הרכב, הנשלח לבונה מחלקת כלי הרכב, נבחר בצוරה רנדומלית באמצעות פונקציית `rand()` - פונקציה זו היא חלק מפונקציות מחלקת הבסיס של `c++` ששם הוא `<cstdlib>`.

הפונקציה מחזירה מספר רנדומלי בין 0 ל - 32767. בשביל לקבל מספר בתחום מסוים, משתמש בפונקציית `%`, אשר באמצעותה נקבל שארית לאחר חילוק במספר מסוים, המבטיחה קבלת תוצאה הקטנה מהמספר אליו בצענו את החילוק.

כלומר, כדי לקבל מספר בתחום 110 - 0, נבצע:

```
int x = rand() % 110
```

הfonקציות שתוארו לעיל יוצרות כלי רכב בתדריות קבועה, כאשר סוג הרכב הנוצר הוא רנדומלי.

ישנם 4 סוגי כלי רכב:

- מכונית קטנה
- מכונית בינונית
- מכונית גדולה
- משאית / אוטובוס

עבור כל סוג כלי רכב יש מאפיינים שונים

```
struct
{
    VehicleTypeOptions Type;
    string VehicleTypeName;
    string ImageDir;
    int ImageCount;
    Vector2f Size;
    vector<Texture> *Textures;
}
VehicleType;
```

כיוון שהבדלים בין סוגי כלי הרכב השונים הם מינוריים למדי, הוחלט שלא ליצר תת - מחלוקת עבור כל אחד מהם ואיז להשתמש בפולימורפים על מנת להתייחס אליהם כרשימה אחת של כל רכב, אלה ליצור מבנים (Struct) סטטיים וקבועים שמכילים את המאפיינים השונים של סוגי כלי הרכב.

כאשר נכנסת הבקשת ליצור כלי רכב חדש, נשלחת לפונקציית הבניה גם סוג כלי הרכב אותו יש ליצר:

```
Vehicle::Vehicle(VehicleTypeOptions vehicleType,
                  int vehicleNumber,
                  list<Lane *> *instructionSet,
                  Map *map) {
```

בקבלת סוג הרכב, פונקציית הבניה "MDBיקה" לאותו כלי הרכב את המאפיינים הקיימים באותו מבנים שהוגדרו מראש.

להלן המבנים השונים המכילים את המאפיינים של כל סוג רכב:

```
VehicleType Vehicle::SmallCar{
    SMALL_CAR,
    "SmallCar",
    "../..../resources/cars/car_image_",
    3,
    Vector2f(1.6 * 100 / Settings::Scale, 3 * 100 / Settings::Scale)
};

VehicleType Vehicle::MediumCar{
    MEDIUM_CAR,
    "MediumCar",
    "../..../resources/cars/car_image_",
    3,
    Vector2f(1.8 * 100 / Settings::Scale, 4 * 100 / Settings::Scale)
};

VehicleType Vehicle::LongCar{
    LONG_CAR,
    "LongCar",
    "../..../resources/cars/car_image_",
    3,
    Vector2f(2 * 100 / Settings::Scale, 5 * 100 / Settings::Scale)
};

VehicleType Vehicle::Truck{
    TRUCK,
    "Truck",
    "../..../resources/cars/car_image_",
    3,
    Vector2f(2.1f * 100 / Settings::Scale, 10 * 100 / Settings::Scale)
};
```

המסלול הוא בעצם רשימה של מציגים לנתיבים, דרכם צריך לעبور כלי הרכב **לפי הסדר**. כאשר כלי רכב נוצר, הוא מקבל גם את המסלול אחריו עליו לעקוב.

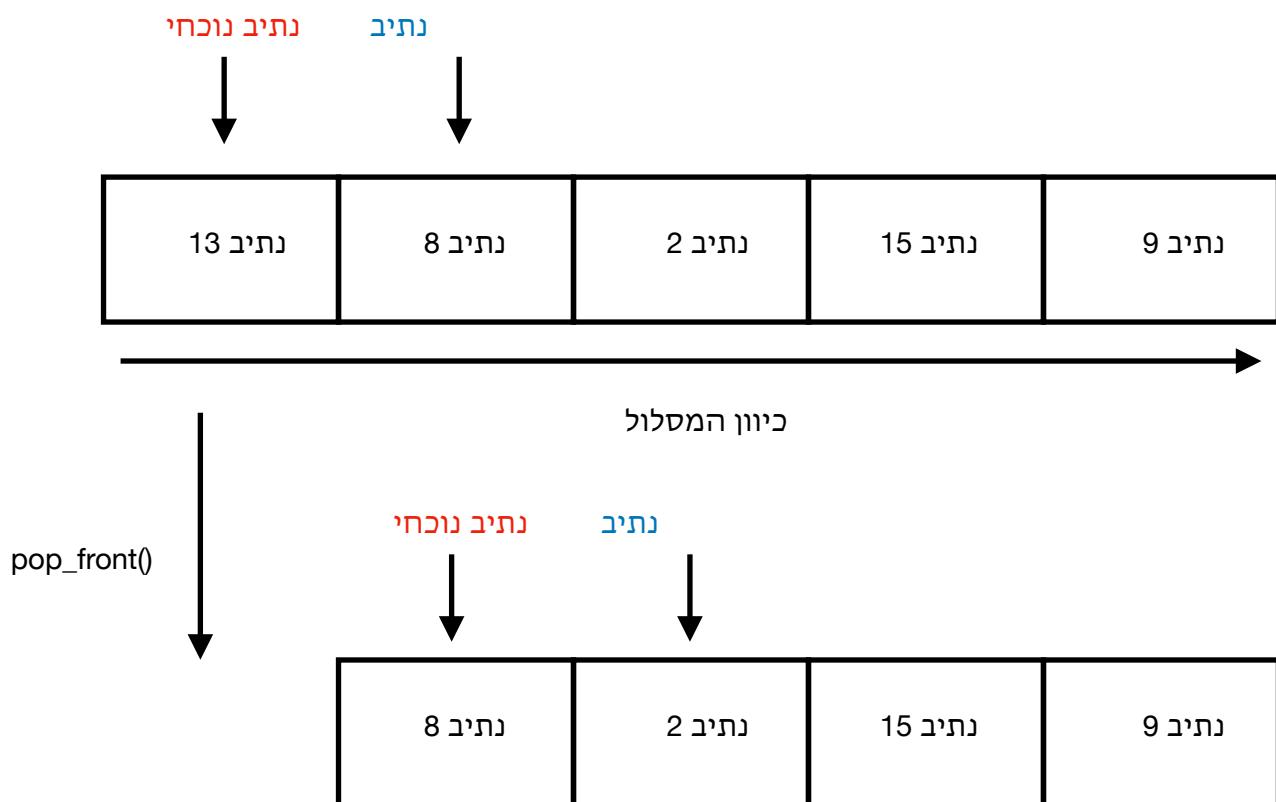
אופן הפעולה של הוראות המסלול הוא כזה:

בכל רגע נתון יש לכלי רכב 2 מצלבים לנתיבים:

- נתיב נוכחי
 - נתיב מטרה

מטרתו של כל הרכיב היא לעبور מהנתיב הנוכחי לנתייב המטרה, באמצעות חציית צומת.

בכל רגע שכל רכב עובר מנתיב הנווכחי לנתיב המטרה, הנתיב הנווכחי נמחק מהרשימה, נתיב המטרה הופך לננתיב הנווכחי ומוגדר נתיב מעלה חדש בהתאם לרשימת נתיבי המסלול.



פונקציה לדוגמה - TransferVehicle

לביצוע שינוי הנתיב הנוכחי לאחר מעבר בין נתיבים, נוצרה הפונקציה `TransferVehicle`. פונקציה זו אחראית לעדכן את כל מאפייני כלי הרכב בהתאם לנ珪ב החדש בו הוא נמצא.

אופן פעולה הפונקציה:

- > הגדר את נתיב המטרה כנתיב הנוכחי
- > הגדר את כיוון הרכב בכיוון הנתיב (לא הכרחי, לצורך דיקוק בלבד)
- > הגדר את מיקום כלי הרכב כמקום התחלת הנתיב (לא הכרחי, לצורך דיקוק בלבד)
- > הגדר את הצומת הנוכחי כצומת של הנתיב הנוכחי
- > הגדר את כלי הרכב שמקדימה ככלי הרכב האחרון שנכנס לנתיב הנוכחי
- > הכנס את כלי הרכב לרשימה כל כלי הרכב שנמצאים בנתיב הנוכחי
- > הוצאה את הנתיב הקודם מרישימת הוראות המסלול.
- > האם נשארו הוראות בהוראות המסלול?
- > הגדר נתיב מטרה לפי הוראות המסלול
- < אחרת
- < הגדר נתיב מטרה כ `Null`, כך לאחר שכלי הרכב יצא מהנתיב הנוכחי הוא ימחק מהמפה.

קוד הפונקציה

```
/// transfer a vehicle from a lane to another lane
void Vehicle::transfer_vehicle(Lane *toLane) {
    this->source_lane_ = toLane;
    this->setRotation(this->source_lane_->GetDirection());
    this->angular_vel_ = 0;
    this->setPosition(this->source_lane_->GetPosition());
    this->curr_intersection_ = this->curr_map_->GetIntersection(
        this->source_lane_->GetIntersectionNumber());
    this->vehicle_in_front_ = (this->source_lane_->GetBackVehicleId())
        ? GetVehicle(this->source_lane_->GetBackVehicleId())
        : nullptr;
    this->source_lane_->PushVehicleInLane(this->vehicle_number_);

    this->instruction_set_->pop_front();
    // if there are instructions left, transfer them to this
    if (!this->instruction_set_->empty())
    {
        this->dest_lane_ = this->instruction_set_->front();
    } else
    {
        this->dest_lane_ = nullptr;
    }
}
```

נהיגה

הנהיגה של כלי הרכב נעשית בצורה אוטונומית - כל כלי רכב נהוג בעצמו, עם באמצעות מידע מינימלי.

לכלי רכב יש 4 מצבים:

- נהוג
- עוצר
- פונה
- נמחק

```
enum State
{
    STOP, DRIVE, TURN, DELETE
};
```

את המצב כלי הרכב מגדר כל רגע ורגע מחדש, בפונקציה `drive()`.

הפונקציה מבצעת 5 בדיקות שונות:

1. **מרחק עם המכונית שמקדימה** - לכל כלי רכב מצביע לרכב שמקדימה אליו בנתיב, רכב זה נמצא לפי רשימות כלי הרכב בנתיב. בדיקה זו מודדת את המרחק שבין 2 כלי הרכב האלה, ואם מרחק זה קטן מרחק מינימלי מוגדר מראש, היא מורה לכלி הרכב לעצור. נוסף לכך, אם העצירה מתבצעת בזמן שהנתיב סגור, כלי רכב זה נוסף לחישוב אורך תור המכוניות בנתיב.

```
// check for distance with car in front
if (vehicle_in_front_ != nullptr && vehicle_in_front_->state_ != DELETE
    && dest_lane_ != nullptr)
{
    float distanceFromNextCar =
        Settings::CalculateDistance(this->getPosition(),
                                     vehicle_in_front_
                                         ->getPosition())
        - this->getSize().y / 2 - vehicle_in_front_->getSize().y / 2;
    float brakingDistance = -(speed_ * speed_) / (2 * deceleration);

    if (distanceFromNextCar
        < brakingDistance + Settings::MinDistanceFromNextCar ||
        distanceFromNextCar < Settings::MinDistanceFromNextCar)
    {
        turning_ = false;
```

```

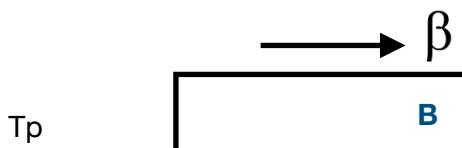
state_ = STOP;
acc_ = deceleration;

// if the lane is blocked, send the stopline-distance
// to the lane and try to set the queue length
if (source_lane_ != nullptr && source_lane_->GetIsBlocked()
    && speed_ == 0
    && active_)
{
    float distanceFromStop =
        Settings::CalculateDistance(this->GetPosition(),
                                    source_lane_
                                    ->GetEndPosition());
    // if this is the last car with STOP state in lane,
    // the queue length is the distance from this vehicle
    // to the end of the lane;
    source_lane_->SetQueueLength(distanceFromStop);
}

return STOP;
}
}

```

2. **ביצוע פניה** - בדיקה האם כל הרכיב נמצא בתוך הצומת עצמו, אם כן בצע פניה.
הפניה מתבצעת בצורה הבא:



עלינו לחשב את מסלול התנועה
מן תיב A לנתיב B.

המטרה שלנו היא למצוא את קצב שינוי הזרות,
כך בהכפלת המהירות הנוכחיות נקבל את השינוי הנכון בכיוון כלי הרכיב.

ב כדי לחשב את קצב שינוי הזרות, יש לחלק את הפרש הזרות בין הנתיבים
במרחק הנסעה ברדיוס בפניה.

$$(\alpha - \beta) = \gamma \rightarrow \text{הפרש הזרות}$$

$$\omega = \gamma / Tp \rightarrow \text{קצב שינוי הזרות}$$



לחישוב מרחק הנסעה K נכפול את היקף עיגול שרדיויסו הוא רדיוס הפניה ביחס
זרות הפניה לכלל המעגל.

$$Tp = P * (\gamma / 360)$$

לחישוב היקף המעגל, משתמש בנוסחה הידועה $r * 2\pi = p$.

חישוב רדיוס הפניה:

משיק למעגל מאונך לרדיויסו.

אם נמתח קו בין נקודות ההשקה של המשיקים עם
המעגל, נקבל משולש שווה-שוקיים.

עם משולש זה, ניתן לחשב את אורך השוקיים
שהוא בעצם הרדיוס של המעגל.

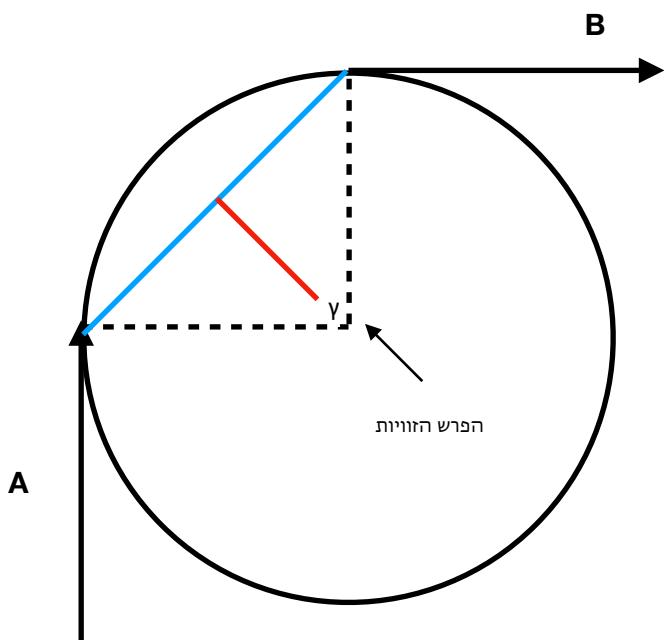
חישוב הרדיוס:

$$B = \text{base}$$

$$H = \text{height}$$

$$A = \text{angle}$$

$$r = (B/2) / (\sin(angle/2))$$



```

// check if car is in between lanes (inside an intersection) and turning
if (curr_intersection_->getGlobalBounds().contains(this->getPosition()) &&
    source_lane_ != nullptr &&
    dest_lane_ != nullptr)
{
    if (!turning_)
    {
        float distanceSourceTarget =
            Settings::CalculateDistance(source_lane_->GetEndPosition(),
                                         dest_lane_->GetStartPosition());
        float angle = Settings::CalculateAngle(source_lane_->GetDirection(),
                                                dest_lane_->GetDirection());

        // if going in a straight line
        if (angle < 1.f && angle > -1.f)
        {
            angular_vel_ = 0;
        } else
        {
            float turningRadius =
                (distanceSourceTarget / 2.f) / (sin(angle * M_PI / 360.f));
            float parameter = 2.f * M_PI * turningRadius;
            float turningParameter = (angle / 360.f) * parameter;

            angular_vel_ = angle / turningParameter;
        }

        turning_ = true;
    }

    if (source_lane_ != nullptr)
    {
        prev_intersection_ =
            curr_map_-
                ->GetIntersection(source_lane_->GetIntersectionNumber());
        source_lane_->PopVehicleFromLane();
        source_lane_ = nullptr;
    }

    state_ = TURN;
    //set rotation
    acc_ = (Settings::AccWhileTurning) ? acceleration / 2.f : 0;
    return TURN;
}

```

3. **עצירה ברמזור** - בדיקה של מרחק כלי הרכב מסופו של נתיב, במקרה והוא סגור.
אם מרחק זה קטן ממרחק מוגדר, בצע עצירה.

```
// check distance from stop (if lane is blocked)
if (source_lane_ != nullptr && source_lane_ != dest_lane_
    && source_lane_->GetIsBlocked() &&
    !this->getGlobalBounds().contains(source_lane_->GetEndPosition()))
{
    float
        distanceFromStop = Settings::CalculateDistance(this->getPosition(),
                                                       source_lane_
                                                       ->GetEndPosition())
        - this->getSize().y / 2;
    float brakingDistance = -(speed_ * speed_) / (2 * deceleration);

    if (distanceFromStop < brakingDistance + Settings::MinDistanceFromStop)
    {
        // if the lane is blocked, send the stopline-distance
        // to the lane and try to set the queue length
        if (speed_ == 0 && active_)
        {
            // if this is the last car with STOP state in lane,
            // the queue length is the distance from this vehicle
            // to the end of the lane;
            source_lane_->SetQueueLength(distanceFromStop);
        }

        // ignore the vehicle in front
        vehicle_in_front_ = nullptr;
        turning_ = false;
        state_ = STOP;
        acc_ = deceleration;
        return STOP;
    }
}
```

4. סיום פניה - בדיקה אם כלי הרכב יצא מהצומת עצמו, משמעו סיים לפנות. במצב זה, יתבצע מעבר בין נתיבים, וכלי הרכב יעבור מהצומת לנטייב המטרה שיוגדר באמצעות הפונקציה ()`TransferVehicle` שתוארה קודם לכן לנטייב הנוכחי.

```
// check if car has left intersection and is now in targetLane
if (dest_lane_ != nullptr
    && dest_lane_->getGlobalBounds().contains(this->getPosition()))
{
    // set previous intersection to nullptr
    prev_intersection_ = nullptr;

    // we need to transfer vehicle to target lane
    transfer_vehicle(dest_lane_);

    turning_ = false;
    acc_ = acceleration;
    state_ = DRIVE;
    return DRIVE;
}
```

5. יציאה מהמפה - בדיקה האם המכונית יצאה מהמפה. נדע זאת כשלכל רכב לא יהיה נתיב מטרה, כלומר רשיימת הוראות המסלול שלו נגמרה, והוא כבר לא נמצא בנטייב הנוכחי. במצב זה, ישלח כלי הרכב למחיקה.

```
// check if car is no longer in intersection
if (dest_lane_ == nullptr
    && !source_lane_->getGlobalBounds().contains(this->getPosition()))
{
    source_lane_->PopVehicleFromLane();

    turning_ = false;
    ++to_be_deleted_;
    state_ = DELETE;
    return DELETE;
}
```

6. **נסעה רגילה** - אם כלי הרכב לא "נפל" באף אחד מ - 5 התנאים הקודמים, מצבו הוא רגיל, כלומר עליו לנסוע בזרחה ורגילה.

```
// default = just drive
active_ = true;
turning_ = false;
acc_ = acceleration;
state_ = DRIVE;
return DRIVE;
```

לאחר שהוגדר מצב הנהיגה הנוכחי של כלי הרכב, וכל המאפיינים הרלוונטיים הוגדרו, יש להוציא את מדיניות זו לפועל.

הfonקציה שאחראית על ההוצאה לפועל היא פונקציית `apply_changes()`.

הfonקציה אחראית על שינוי המהירות בהתאם לתאוצת כלי הרכב, שינוי הכוון בהתאם לקצב שינוי הזרווית, ייצור וקטור המתאר את ההפרש בין המיקום הנוכחי למיקום הבא שחושב והוספתו למיקום הנוכחי של כלי הרכב.

```
/// apply the calculated next position
void Vehicle::apply_changes(float elapsed_time) {
    // apply acceleration
    speed_ += acc_ * elapsed_time * Settings::Speed;
    //float running_speed_ = speed_ * Settings::Speed;

    // apply max Speed limit
    if (speed_ > max_speed_)
        speed_ = max_speed_;

    // apply min Speed limit
    if (speed_ < 0)
        speed_ = 0;

    // set rotation relative to currentSpeed, to create a constant turning radius
    Transform t;
    this->rotate(angular_vel_ * speed_ * elapsed_time * Settings::Speed);

    t.rotate(this->getRotation());

    // rotate the movement vector in relation to the forward vector (0,1)
    movement_vec_ = t.transformPoint(Settings::BaseVec);

    // apply movement vector on position, relative to elapsed time to ensure
    // a constant Speed at any FPS

    this->move(movement_vec_ * speed_ * elapsed_time * Settings::Speed);
}
```

קוד המחלקה

```
class Vehicle : RectangleShape
{
public:

    Vehicle(VehicleTypeOptions vehicleType,
             int vehicleNumber,
             list<Lane *> *instructionSet,
             Map *map);
    ~Vehicle();

    void Draw(RenderWindow *window);
    void Update(float elapsedTime);

    // add entities
    static Vehicle *AddVehicle(list<Lane *> *instructionSet,
                               Map *map,
                               VehicleTypeOptions vehicleType = SMALL_CAR,
                               int vehicleNumber = VehicleCount + 1);

    // get
    int GetVehicleNumber() { return vehicle_number_; }
    bool GetIsActive() { return active_; }
    Lane *GetSourceLane() { return source_lane_; }
    Lane *GetTargetLane() { return dest_lane_; }
    Lane *GetCurrentLane() { return source_lane_; }
    State GetState() { return state_; }
    Vector2fGetPosition() { return getPosition(); }
    list<Lane *> *GetInstructionSet() { return instruction_set_; }
    static VehicleType *GetVehicleTypeByOption(VehicleTypeOptions
vehicleTypeOptions);
    static Vehicle *GetVehicle(int vehicleNumber);

    // set
    void Select();
    void Unselect();

    static Vehicle *CheckSelection(Vector2f position);

    static void DeleteAllVehicles();
    static void ClearVehicles();

    static bool LoadVehicleTextures(VehicleType *vehicleType);

    static list<Vehicle *> ActiveVehicles;
    static int GetActiveVehicleCount() { return ActiveVehiclesCount; }
    static int VehicleCount;
    static Vehicle *SelectedVehicle;
```

```

static int VehiclesToDeploy;

private:

State drive();
void apply_changes(float elapsedTime);
void transfer_vehicle(Lane *toLane);

// A Count of vehicles due to be deleted
static int to_be_deleted_;
// The Count of the active running vehicles
static int ActiveVehiclesCount;

static VehicleType SmallCar;
static VehicleType MediumCar;
static VehicleType LongCar;
static VehicleType Truck;

// ID of this vehicle
int vehicle_number_;
VehicleType *vehicle_type_;

// The movement vector of this vehicle
Vector2f movement_vec_;

float speed_;
float acc_;
float max_speed_;
float acceleration;
float deceleration;
float angular_vel_;
bool turning_;
bool active_;

Vehicle *vehicle_in_front_;

list<Lane *> *instruction_set_;

Map *curr_map_;
Lane *source_lane_;
Lane *dest_lane_;
Intersection *curr_intersection_;
Intersection *prev_intersection_;

State state_;

DataBox *data_box_;
};


```

מחלקה הסימולציה - Simulation

מחלקה זו מייצגת אובייקט מסווג סימולציה. סימולציה היא בעצם "תמונה מסך" של הרצה שביצע מנוע הסימולציה, כאשר היא מכילה מידע מינמלי בלבד אודות הרצה.

המידע שהסימולציה שומרת הוא:

- זמן התחלת הסימולציה
- זמן סיום הסימולציה
- מספר המכוניות שהורצו
- ציון הסימולציה
- זמן הסימולציה

רק סימולציה אחד יכולה לזרז בכל רגע.

ישנה גם האופציה לבצע דמונסטרציה (Demo) של סימולציה נבחרת. דמונסטרציה של סימולציה היא הרצה של סימולציה באותה סביבה ובאותן ההגדרות, אך ללא שמירה של זמנים, ציונים וכו'.

סימולציה מתבצעת ממשך זמן לא מוגדר, אך עבור מספר כל רכב מוגדר. ציונה של סימולציה ניתנת לפי הזמן שלקח למספר המכוניות זהה לעבור את המפה. לעומת, אם נרים 2 סימולציות עם אותו מספר כל רכב אך מדיניות פאות שונות וחתת מהן תסת沆ים לפני האחרת, נדע שהסימולציה עם זמן הביצוע הקצר יותר היא בעלת המדיניות הטובה מבין השניים.

ציון של סימולציה מוגדר כמספר מכוניות שעברו את המפה בשניה של סימולציה. כך, ככל להבטיח שזמן סימולציה קצר יותר עברו כמות מוגדרת של מכוניות יניב ציון גבוה יותר.

לדוגמה, אם סימולציה קיבלה ציון של 0.44, המשמעות היא שבמפה הנוכחית, תחת ההגדרות ומדיניות הפאות הנוכחית, זרימת התחבורה במפה הייתה 0.44 מכוניות בשניה.

לצורך נתינת ציון למדיניות פאות מסוימת ניתן להשתמש במספר ורק של נתוניים:

- זמן המבחן המוצע של מכוניות בזומת
- מספר הפעמים שככל מכונית הייתה צריכה לעצור
- אורכי התורים בצמתים

אך לפי דעתך זרימת התחבורה דרך צומת או כמה צמתים היא דרך נכונה לשלב את הנתוניים שצווינו לעיל, כמו כן היא בעלת חשיבות ואופן הביטוי הći גדול במציאות.

הסימולציות נשמרות בקובץ NSJL באופן זהה לשמירת המפות.

אופן הפעולה של עדכון סימולציה לאחר הפעלה הוא כזה:

- < האם סימולציה זו פעליה האם היא לא נגמרה?
- < הוסיף את הזמן ש עבר מאז הקריאה הקודמת לעדכון לזמן הסימולציה
- < האם אין כלי רכב פעילים, והאם לא נותרו כל רכב לייצר?
- < הגדר את הסימולציה כ"נגמרה" וכ"לא פעליה"
- < הגדר זמן סיום הסימולציה
- < חשב את ציון הסימולציה

```
// function returns true if simulation has ended
bool Simulation::Update(float elapsedTime) {

    if (running_ && !finished_)
    {
        elapsed_time_ += elapsedTime * Settings::Speed;

        current_vehicle_count_ = Vehicle::GetActiveVehicleCount();

        if (current_vehicle_count_ == 0 && Vehicle::VehiclesToDeploy == 0)
        {
            running_ = false;
            finished_ = true;
            Simulation::SimRunning = false;
            Simulation::DemoRunning = false;

            // get simulation end time
            end_time_ = time(nullptr);

            score_ = float(vehicle_count_) / elapsed_time_;

            if (Settings::PrintSimulationLog)
            {
                PrintSimulationLog();
            }
        }

        return true;
    }
}
return false;
}
```

בנוסח, ישנה האפשרות להזפיס לקונסול דו"ח של הסימולציה שהרגע נגמרה. דו"ח סימולציה נראה כך:

Set 1
Simulation 1 ended at Tue Apr 7 19:54:51 2020

Simulation Type : Vehicle Count
Results:
 Vehicles Simulated: 50
 Simulation Time: 114.539 seconds
 SCORE: 0.436534

קוד המחלקה

```
class Simulation{
public:

    Simulation(int simulationNumber, int setNumber, int vehicleCount = 1000);
    ~Simulation();

    bool Update(float elapsedTime);
    void Run() {
        running_ = true;
        SimRunning = true;
        start_time_ = time(nullptr);
        Vehicle::VehiclesToDeploy = vehicle_count_;
    }
    void Demo() {
        running_ = true;
        DemoRunning = true;
        Vehicle::VehiclesToDeploy = vehicle_count_;
    }
    void PrintSimulationLog();
    void StopDemo() {
        finished_ = true;
        DemoRunning = false;
    }
    void StopSimulation() {
        finished_ = false;
        finished_ = true;
        SimRunning = false;
    }
    // get
```

```

int GetSimulationNumber() { return simulation_number_; }
int GetVehicleCount() { return vehicle_count_; }
int GetSetNumber() { return set_number_; }
float GetResult() { return result_; }
int GetCurrentVehicleCount() { return current_vehicle_count_; }
int IsFinished() { return finished_; }
int IsRunning() { return running_; }
time_t *GetStartTime() { return &start_time_; }
time_t *GetEndTime() { return &end_time_; }
float GetElapsedTime() { return elapsed_time_; }

// set
void SetStartTime(time_t time) { start_time_ = time; }
void SetEndTime(time_t time) { end_time_ = time; }
void SetSimulationTime(float time) { elapsed_time_ = time; }
void SetFinished(bool fin) {
    finished_ = fin;
    running_ = false;
}

// The total count of all the simulations created this session
static int SimulationCount;
// Is a simulation currently active
static bool SimRunning;
// Is a demo of a simulation currently active
static bool DemoRunning;

private:
// ID of this simulation
int simulation_number_;
// the set number of this simulation
int set_number_;
// Vehicles created in this simulation
int vehicle_count_;
// The Vehicles left in this simulation
int current_vehicle_count_;
// Is this simulation finished
bool finished_;
// Is this simulation active and running
bool running_;
// The start time of this simulation
time_t start_time_;
// The end time of this simulation
time_t end_time_;
// Time elapsed since simulation has begun
float elapsed_time_;
// The result of the simulation
// vehicles per second
float result_;
};


```

מחלקה ה **Set**

מחלקה זו מייצגת אובייקט ה **set**, שהוא בעצם אוסף של סימולציות.

ה **set** מייצג אימון הבינה המלאכותית של המערכת באמצעות הרצת מספר מוגדר של סימולציות, שלאחר כל אחד מהן המערכת מסיקה מסקנות ומשפרות את עצמה.

ה **set** הוא בעצם תקופת אימון של מערכת הבינה המלאכותית, עליה יפורט בהמשך.

ה **set** מחזק רשימה של סימולציות, והוא אחראי על יצרתן והפעלתן. בהגדרת מספר סימולציות בסט אותו אנחנו רוצים להפעיל, יתחליל ה **set** בהפעלת סימולציות.

לאחר שהסימולציה הראשונה מסתיימת ונתוניה נשמרים, באופן אוטומטי מפעיל ה **set** שירות סימולציה חדשה, וכך עד שבוצעו מספר הסימולציות המבוקש.

על מנת הרצת הסימולציות זו אחר זו, פונקציית (*Update* יוצרת, מפעילה ועוקבת אחרי הסימולציות של ה **set**).

אוףן הפעולה של הפונקציה:

- < האם ה **set** פעיל?>
- < חשב את התקדמות ה **set** (כמה סימולציות בוצעו לעומת מטרת הביצוע)>
- < האם מספר הסימולציות שהורךו קטן ממספר הסימולציות שיש להריץ?>
- < האם ישנה סימולציה פעילה?>
- < עדכן את הסימולציה הזאת (פונקציית העדכון תוארה בעמ' 56)>
- < האם הסימולציה הפעילה הושלמה?>
- < הגדר שאין סימולציות פעילות>
- < אחרת, צור סימולציה חדשה והפעיל אותה>
- < אחרת, הגדר שה **set** נגמר והוא אינו פועל יותר>

כמו כן, המחלקה אחראית על הפעלת דמונסטרציה של סימולציה. כפי שצוין בפרק של הסימולציה, זהה הדגמה בלבד של הסימולציה, ולכן נתוניה אינם נשמרים ואין נחשיים כחלק מה **set**.

קוד הפונקציה `Update()`

```
bool Set::Update(float elapsedTime) {

    if (running_)
    {
        //update set progress
        progress_ = float(generations_simulated_) / float(generations_count_);

        // if set is not finished
        if (generations_simulated_ < generations_count_)
        {
            // is there a simulation running ?
            if (running_simulation_ != nullptr)
            {
                running_simulation_->Update(elapsedTime);

                // check if simulation has finished
                if (running_simulation_->IsFinished())
                {
                    last_simulation_result_ = running_simulation_->GetResult();

                    running_simulation_ = nullptr;
                    ++generations_simulated_;
                    return true;
                }
            } else
            {
                // start a new simulation and run it
                running_simulation_ = StartNewSimulation();
            }
        } else
        {
            running_ = false;
            finished_ = true;
            SetRunning = false;
            end_time_ = time(nullptr);
            return true;
        }
    }

    if (running_demo_ != nullptr)
    {
        running_demo_->Update(elapsedTime);
        if (running_demo_->IsFinished())
        {
            running_demo_ = nullptr;
        }
    }
    return false;
}
```

```

class Set
{
public:

    Set(int setNumber, int generationCount = 20, int vehicleCount = 1000);
    ~Set();

    bool Update(float elapsedTime);
    void StopSet();
    void RunSet() {
        running_ = true;
        SetRunning = true;
        start_time_ = time(nullptr);
    }
    bool DemoSimulation(int simulationNumber);
    Simulation *StartNewSimulation();
    Simulation *AddSimulation(int simulationNumber, int vehicleCount);
    bool DeleteSimulation(int simulationNumber);
    bool StopDemo();

    // set

    void SetStartTime(time_t time) { start_time_ = time; }
    void SetEndTime(time_t time) { end_time_ = time; }
    void SetFinished(bool finished) {
        finished_ = finished;
        running_ = false;
    }
    void SetGenerationsSimulated(int count) { generations_simulated_ = count; }
    void SetGenerationCount(int count) { generations_count_ = count; }
    void SetProgress(float progress) { progress_ = progress; }

    // get
    int GetSetNumber() { return set_number_; }
    int GetNumberOfSimulations() { return number_of_simulations_; }
    int GetGenerationsSimulated() { return generations_simulated_; }
    int GetGenerationsCount() { return generations_count_; }
    int GetVehicleCount() { return vehicle_count_; }
    int GetRunning() { return running_; }
    int GetProgress() { return progress_; }
    time_t *GetStartTime() { return &start_time_; }
    time_t *GetEndTime() { return &end_time_; }
    bool IsFinished() { return finished_; }
    bool IsRunning() { return running_; }
    Simulation *GetSimulation(int simulationNumber);
    vector<Simulation *> *GetSimulations() { return &simulations_; }
    float GetLastSimulationResult() { return last_simulation_result_; }
}

```

```

static int SetCount;
static bool SetRunning;

private:

    // ID of this set
    int set_number_;
    // number of simulations in this set
    int number_of_simulations_;
    // number of simulations that have been done in this set
    int generations_simulated_;
    // number of simulation that have to be done in this set
    int generations_count_;
    // number of vehicles to run each simulation;
    int vehicle_count_;

    bool running_;

    bool finished_;
    // the generation progress.
    // calculated by gen_simulated_ / gen_count_
    float progress_;

    // The start time of this simulation
    time_t start_time_;
    // The end time of this simulation
    time_t end_time_;

    // simulations in this set
    vector<Simulation *> simulations_;
    // a pointer to the simulation that is currently being ran
    Simulation *running_simulation_;
    Simulation *running_demo_;

    float last_simulation_result_;
};


```

מחלקה הגדרות - Settings

מחלקה זו מחזיקה בכל הגדרות הקשורות להפעלת הסימולציה. מחלקה זו היא סטטית ופומבית, כלומר כל אובייקט במערכת יכול לגשת למשתנים בהגדירה זו.

קיים מחלקה זו מקל מאוד על העבודה והשימוש במערכת, מכיוון שהוא מרכז בצורה מאוד פשוטה תחום רחב מאוד של הגדרות.

אומנם ישנו מחלקות שליהם לא אמורה להיות גישה לחלק מפרמטרים אלו, אך האלטרנטיבתה שהיא לשמר כל הגדרה רק במחלקה (או המחלקות) המשמשות בה יהיה יוצר סיבוך, קושי למצוא הגדרה כשרוצים לשנות אותה, הופעתה והגדرتה מחדש מחלקות אחרות (רק כך ניתן לעשות ב`++`).

כאמור, מחלקה זו מחזיקה מגוון רחב מאוד של הגדרות, ובאמצעותן ניתן:

- האפשרות להדפיס / לא להדפיס מגוון רחב של נתונים לקונסול
- לציר / לא לציר טקסטורות של מכוניות
- לציר / לא לציר את המיני-מפה
- לעקוב / לא לעקוב אחרி מכוניות שנבחרה
- לשנות את תדיות הפעולה של המחזר הלוגי
- לשנות את תדיות הפעולה של המחזר הצירוי
- להגדיר מרחק מינמלי בין מכוניות למוכנית
- להגדיר מרחק מינימי בין מכוניות לקו עצירה
- להגדיר רוחב נתיב
- להגדיר את קנה המידה של המערכת
- להגדיר את מהירות ההריצה של הסימולציה
- לקבוע את קצב יצירתן של מכוניות
- לקבוע את האזום של המצלמה
- להמיר בין יחידות מרחק ומהירות שונות
- להגדיר מהירות / גאוזות מקסימליות של כל הרכיב השונים
- להגדיר זמן מחזור מקסימלי ומינימי
- ועוד ...

דוגמה לשימוש במחלקה

להלן דוגמה של שימוש בהגדרת ציור טקסטורות של מכוניות.
כאשר נוצרת מכונית, נבדקת הגדרה זו ובההתאם אליה נטענת / לא נטענת טקסטורה.

```
// if vehicle texture hasn't been loaded yet, load it
if (Settings::DrawTextures && Vehicle::LoadVehicleTextures(vehicle_type_))
{

    // set up sprite
    int textureNumber;
    if (Settings::MultiColor)
    {
        textureNumber = (vehicle_number_ % vehicle_type_->ImageCount);
    } else
    {
        textureNumber = 1;
    }

    this->setTexture(&(vehicle_type_->Textures->at(textureNumber)));
} else
{
    this->setOutlineThickness(10.f);
    this->setOutlineColor(Color::Blue);
    this->setFillColor(Color::Transparent);
}
```

```

enum DistanceUnits
{
    M, CM, FEET, INCH, PX
};

enum VelocityUnits
{
    CMS, KMH, MS, MPH, PXS
};

enum VehicleTypeOptions
{
    SMALL_CAR, MEDIUM_CAR, LONG_CAR, TRUCK
};

class Settings
{
public:

    static Net NeuralNetwork;

    static float GetLaneWidthAs(DistanceUnits unit);
    static float GetMaxSpeedAs(VehicleTypeOptions vehicleType,
                               VelocityUnits unit);
    static float ConvertSize(DistanceUnits fromUnit,
                           DistanceUnits toUnit,
                           float value);
    static float ConvertVelocity(VelocityUnits fromUnit,
                               VelocityUnits toUnit,
                               float value);

    static const Vector2f BaseVec;

    static bool DrawFps;
    static bool DrawActive;
    static bool DrawDelete;
    static bool DrawAdded;
    static bool DrawVehicleDataBoxes;
    static bool DrawRoadDataBoxes;
    static bool DrawLightDataBoxes;
    static bool DrawRoutes;
    static bool DrawLaneBlock;
    static bool DrawTextures;
    static bool DrawClickPoint;
    static bool DrawMinimap;
    static bool DrawSimTable;
    static bool FollowSelectedVehicle;
    static bool LaneDensityColorRamping;
    static bool ShowSelectedPhaseLanes;
    static bool PrintSimulationLog;
};

```

```

static bool DrawNnProgression;

static int Interval;
static int Fps;
static int AntiAliasing;
static bool MultiColor;
static bool MultiTypeVehicle;
static float MinDistanceFromNextCar;
static float MinDistanceFromStop;
static bool AccWhileTurning;

static float LaneWidth;
static float MinLaneWidth;
static float MaxLaneWidth;
static float DashLineLength;
static float DashLineSpace;
static float Scale;
static float Speed;
static bool DoubleSeparatorLine;
static float VehicleSpawnRate;
static float MaxDensity;

static float DefaultLaneLength;
static int GridColumns;
static int GridRows;
static bool DrawGrid;

static int SFMLRatio;
static int DefaultMapWidth;
static int DefaultMapHeight;
static bool MapOverflow;

static float Zoom;
static float DragFactor;

static float MinimapSize;
static float MinimapMargin;

// an array of scales for each distance unit
// 0 - M
// 1 - CM
// 2 - Feet
// 3 - Inches
// 4 - Pixels
static float DistanceUnitScales[5];

```

```

// an array of scales for each distance unit
// 0 - cm/s
// 1 - km/h
// 2 - m/s
// 3 - mph
// 4 - px/s
static float VelocityUnitScales[5];

// and array of max speeds for each vehicle in km/h
// 0 - Car
// 1 - Truck
// 2 - Motorcycle
static float MaxSpeeds[4];
// same for accelerations
static float Acceleration[4];
// same for braking
static float Deceleration[4];

static float CalculateDistance(Vector2f a, Vector2f b);
static float CalculateAngle(float a, float b);
static string ConvertTimeToString(tm *time);
static tm *ConvertStringToTime(const string str);

static void GetHeatMapColor(float value,
                           float *red,
                           float *green,
                           float *blue);

static float OrangeDelay;
static float DefaultCycleTime;
static float MaxCycleTime;
static float MinCycleTime;
static float PhaseDelay;
};


```

הבינה המלאכותית

בינה מלאכותית הוא ענף של מדעי המחשב שעוסק בפעולות מחשבים בצורה שאפיינה עד כה את פעילות המוח האנושי. מכאן שישיות בינה מלאכותית רבות נובעות יישירות מישיות העבודה שידועות לנו מהאדם, כגון אופן פעילות המוח שלנו.

בינה מלאכותית היא תחום גדול ועדיין נחקרת בצורה אינטנסיבית.

רשת נוירוניים

למידת מכונה היא תת-תחום בענף הבינה המלאכותית שעוסק בפיתוח אלגוריתמים המיעדים לאפשר למחשבים ללמידה על נושאים שונים ומtower כך להסיק מידע ולפעול בצורה שתכונות קונבנציונאלית איננו מסוגל.

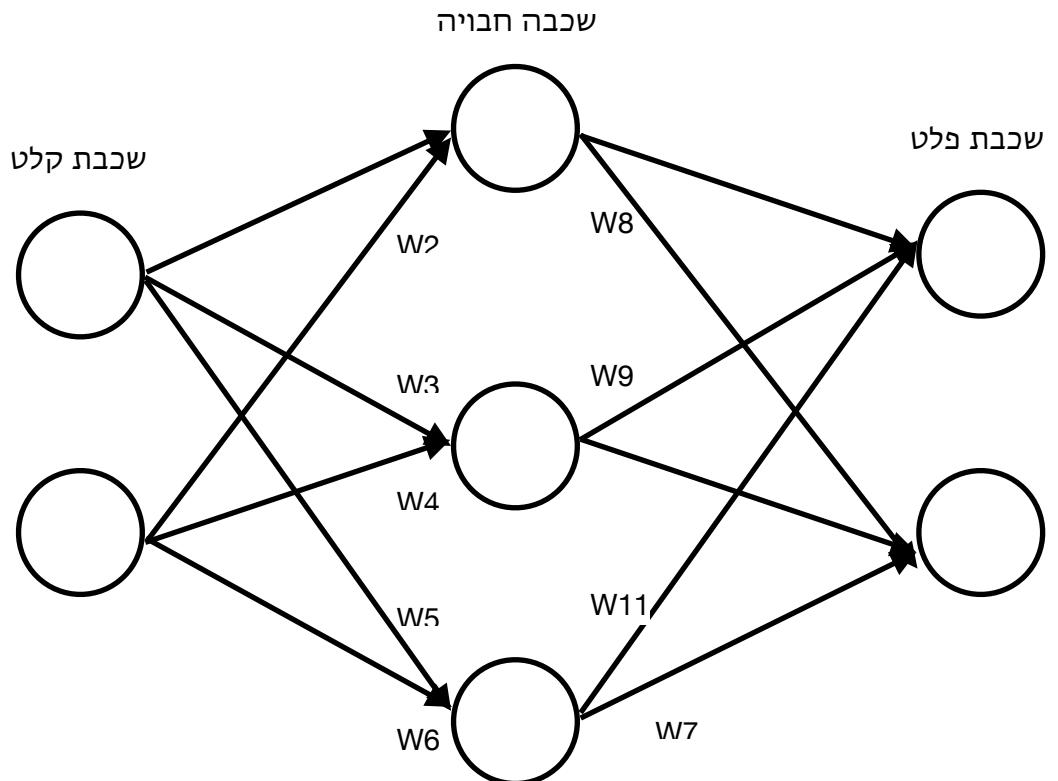
במערכת זו נעשה שימוש בלמידה תוכנה בצורה של **רשת נוירוניים**.

רשת נוירוניים היא מודל לחישובים מתמטיים שפותח בהשראת פעילות המוח האנושי.

לרשת נוירוניים כמה מרכיבים:

1. **נוירון** -נוירון הוא יחידת עבודה מידע.נוירון קולט מידע כלשהו, ומסוגל (אך לא תמיד) לבצע עלייה פעולה מתמטית (בד"כ לא לינארית) פשוטה, ופולט אותו החוצה.
2. **משקל** - משקל הוא חיבור בין נוירון לנוירון, בעל ערך מסוים. כל מידע שעובר דרך משקל זה מנוירון לנוירון, מוכפל במסקל זה בצורה לינארית.
3. **שכבה** - שכבה היא אוסף של נוירונים שלהם תפקיד מסותי. לדוגמה, שכבת הקלט של רשת נוירוניים היא שכבה בה כל הנוירונים הם נוירוני קלט, ותפקידם להעביר את המידע אל השכבה הבאה. השכבה הבאה היא בדרך כלל שכבה "נסתורת". זהה שכבה בה המידע עובר דרך המשקלים מנוירון לנוירון, ועובד עיבודים לינאריים ולא לינאריים במעברו בין נוירון לנוירון. השכבה האخונה היא שכבת הפלט, שתפקידו הנוירונים שלה הוא לפולוט את המידע שעובר בשכבה הנסתורת החוצה.
4. **פונקציית אקטיבציה** - פשוט מאוד, פונקציה זו מחליטה האם בהתאם למידע שנכנס אליה, האם על הנוירון "להידלק" הוא "להיכבות". קיימות פונקציות רבות שונות לביצוע פעולה זו, בהתאם לצורכי המערכת.

במערכת שלילי מבנה רשת הנוירונים הוא כזה:



אלגוריתם גנטי

אלגוריתם גנטי הוא אלגוריתם המבוסס על עקרון הבחירה הטבעית, לפיו "החזק שורד". אלגוריתם גנטי מפעיל "דור" של אובייקטים שלכל אחד מהם "גנטיקה" (מאפיינים) שונים. לאחר ביצוע המשימה, לכל אובייקט ניתן ציון על אופן לפיעולתו ביחס לדרישות המערכת.

לאחר מכן, נוצר דור חדש המבוסס על הדור הקודם המתאים לציון של כל אחד מהאובייקטים מהדור הקודם. כמובן, ככל שצינוו של אובייקט גבוה יותר ביחס לשאר האובייקטים, כך גדל הסיכוי שחלק ממאפייניו יופיעו באובייקטים שיוצרים בדור הבא.

שילוב 2 עקרונות אלו יוצר את מה שנקרא "neural-evolution".

פעולות הלמידה ברשת הנוירונים שלילי מתחלתת ל- חלקים:

1. **Feed-Forward** - העברת המידע דרך כל שכבות הרשת, UIBODO במשקלים ובפונקציות האקטיבציה של הננוירונים עד הגיעו לשכבת הפלט.
2. **Output** - הוצאת התוצאות משכבת הפלט.
3. **Normalization** - חישוב ציון רשת הנוירונים ביחס לשאר הרשות השיווקות לאותו דור.
4. **Pool Selection** - ייצור דור חדש הגדרת כל אחד מהרשותות שלו לפי ציוני רשותות הדור הקודם.

מחלקה הרשות - Net

מחלקה זו מייצגת אובייקט מסווג רשות נוירוניים. היא מכילה מערך "שכבות" של נוירונים, כאשר כל שכבה מייצגת נוירונים בעלי תפקיד אחר.

כמו כן, מחלקת זו אחראית לניהול דורות הרשותות, ולכל פעילות האלגוריתם הגנטי.

אלגוריתם הגנטי שייכים כמה אלמנטים:

- מערך הדור, המציג את כל הרשותות השיכוכות אליו
- מצביע לרשות באמצעות מתבצעת הסימולציה הנוכחית
- העתק של הרשות בעלת הביצועים הטובים ביותר עד כה
- הביצוע הטוב ביותר עד כה

תהליך השימוש ברשות נוירוניים פשוט למדי, והוא כולל אימפלמנטציה של עקרון ה-ForwardFeed. לשם כך ישנה פונקציה FeedForward שבצורה פשוטה "מעבירה" את המידע דרך הרשות, מפעילה עליו את כל המשקלים שברשות, מפעילה פונקציות אקטיבציה, ומציבת את התוצאה בנוירונים שבשכבות הפלט.

קוד המחלוקת

```
class Net
{
public:
    Net() {}
    Net(const vector<unsigned> &topology);

    void Draw(RenderWindow *window);
    void Update(float elapsedTime);

    void Reset();

    void FeedForward(const vector<double> &inputVals);
    [[maybe_unused]] void PrintNet();
    void SetScore(double score) { score_ = score; };
    void GetResults(vector<double> &resultVals) const;
    void Save(const string dir);
    static void Load(const string dir);

    static void NextGeneration();
    static void NormalizeFitness(vector<Net> &oldGen);
    static Net PoolSelection(const vector<Net> &oldGen);
    static vector<Net> Generate(const vector<Net> &oldGen);

    static vector<Net> Generation;
    static unsigned CurrentNetIndex;
    static unsigned GenerationCount;
```

```

static Net *CurrentNet;
static Net BestNet;
static float HighScore;
static const unsigned PopulationSize;

private:
    vector<Layer> layers_; //layers_[layerNum][neuronNum]
    double fitness_;
    double score_;
    double error_;
    double recent_average_error_;

    Vector2f calculate_neuron_position(unsigned layerNum, unsigned layerCount,
                                         unsigned neuronNum, unsigned neuronCount);

    void mutate(float mutationRate);

    void create_weight_vertex_array();
    vector<VertexArray> weight_lines_;
    Vector2f size_;
};


```

פונקציית **NextGeneration**

לאחר סיום פעולה של דור רשותות, על מחלקה זו ליצור דור חדש של רשותות המבוסס על הדור הקודם.

לשם כך קיימת פונקציית **NextGeneration()**, העושה את כל הנדרש על מנת ליצור דור חדש בהתאם לעקרונות הפונקציה הגדנית.

הפונקציה הגדנית במערכת זו פועלת בצורה הבאה:

- בצע נורמליזציה של תוצאות ביצועי הרשותות של הדור הקודם
- צור דור חדש כאשר חלקה של כל רשות בגנטיקת הרשותות בדור הבא תלויות בציון המנורמל שלו.
- הגדר את הדור שנוצר כדור הנוכחי, ומה שאר את ביצוע הסימולציות ושימוש ברשותות שבדור החדש.

קוד המחלקה

```
void Net::NextGeneration() {  
  
    Net::CurrentNet = nullptr;  
  
    // normalize the fitness of all the nets in this gen  
    Net::NormalizeFitness(Net::Generation);  
    // create a new generation of nets  
    vector<Net> newGen = Net::Generate(Net::Generation);  
    // copy the new gen to the Generation array  
    Net::Generation = newGen;  
  
    Net::CurrentNetIndex = 0;  
    Net::GenerationCount++;  
}
```

פונקציית NormalizeFitness

על מנת לבצע נורמליזציה של תוצאות הרשות, משתמש בפונקציית NormalizedFitness, שתתבצע על רשות ציון הנקרא "כושר", שהוא מייצג את ציון הרשות ביחס לסכום ציוני הרשותות בדור.

על מנת לגרום לביצוע של רשות להיות טוב בצורה אקספוננציאלית, כך ש- α נקודות יותר יהיו בעלות השפעה גדולה בהרבה בחלוקת העליון של שדה התוצאות, נעה וראשית את ציוני הרשותות בריבוע, ונסכום את כולם.

הכשר של כל רשות מוגדר כציינו שלו ביחס לסכום הציונים הכלול של אותו דור.

קוד הפונקציה

```
void Net::NormalizeFitness(vector<Net> &oldGen) {
    double sum = 0;
    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        double score = pow(oldGen[i].score_, 2);
        oldGen[i].score_ = score;
        sum += score;
    }
    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        oldGen[i].fitness_ = oldGen[i].score_ / sum;
    }
}
```

פונקציית Generate

פונקציה זו יוצרת דור חדש על בסיס הדור הקודם.

הfonקציה מבצעת 2 דברים:

- יוצרת רשות חדשה המבוססת על הרשותות מהדור הקודם
- מפעילה על אותה הרשות מוטציה כלשהי.

```
vector<Net> Net::Generate(const vector<Net> &oldGen){
    vector<Net> newGen;
    for (unsigned i = 0; i < oldGen.size(); i++)
    {
        newGen.push_back(Net::PoolSelection(oldGen));
        newGen.back().mutate(0.2);
    }
    return newGen;
}
```

פונקציה PoolSelection

פונקציה זו תפקידה לבחור מדור קודם רשות מסוימת, כאשר סיכוי הבחירה ברשות כלשהו מושפעים ישירות מהקשר שלו. כמובן, ככל שהקשר של רשות גדול יותר, כך הסיכויים שפונקציה זו תבחר בה ותבוסס עליה רשות בדור הבא גדולים יותר.

הfonקציה עשויה זאת כך:

```
< הגדר z מספר רנדומלי בין 0 ל -1  
< הגדר 0 = i  
< כל עד 0 > r  
r -= generation[i].fitness <  
i++ <  
    <- i  
< החזר [i]generation
```

אם נבחר מספר רנדומלי בין 0 ל -1, ונוריד ממנו לפי הסדר את הקשר של כל רשות, בסופו של דבר המספר הזה יתאפס לאחר הורדתה של ערך הקשור מסוים. ככל שערכ הקשר של רשות גדול יותר - כך בתורו יורד יותר מהמספר הרנדומלי, וכך הסיכוי שהמספר יתאפס עלתו יגדל.

קוד הפונקציה

```
Net Net::PoolSelection(const vector<Net> &oldGen)  
{  
    unsigned index = 0;  
    double r = rand() / double(RAND_MAX);  
  
    while (r > 0)  
    {  
        r -= oldGen[index].fitness_;  
        index++;  
    }  
    index--;  
    return oldGen[index];  
}
```

ויזואלייזציה של רשת הנוירונים

במערכת רשת הנוירונים האז ישנה גם אפשרות ליצור המחברה של המתרחש בתוך רשת הנוירונים.

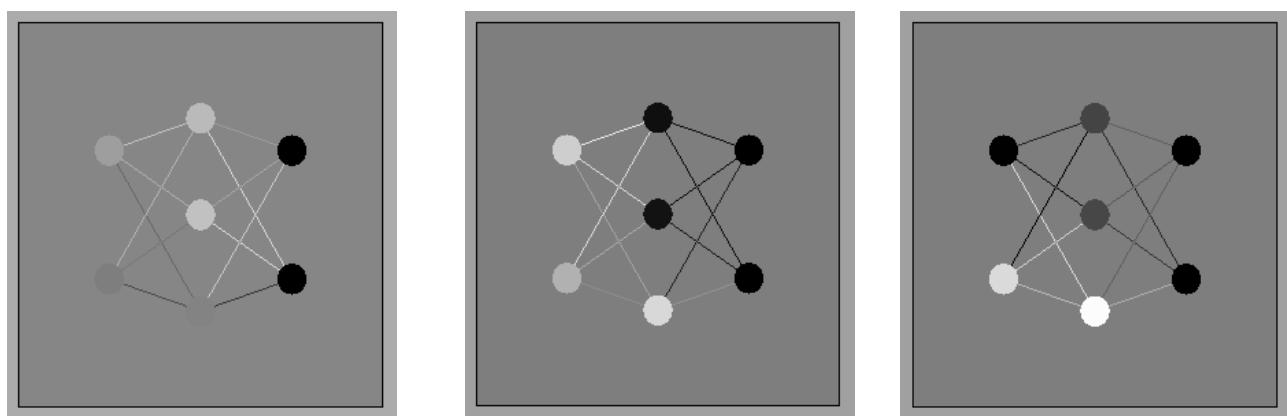
על מנת לעשות זאת, יצורו איזור במרק הסימולציה, שפועל בהתאם הוצאה כמו המיני-מפה (הסביר מלא בפרק מחלקה המפה - Map). באיזור זה, מצוירים כל הנוירונים ברשת הנוירונים, וכך גם כל המשקלים שביניהם.

הנוירונים מיוצגים על ידי עיגולים פשוטים, והמשקלים מיוצגים על ידי קוים. אלמנטים אלו ניתנים לי על ידי ספרית SFML.

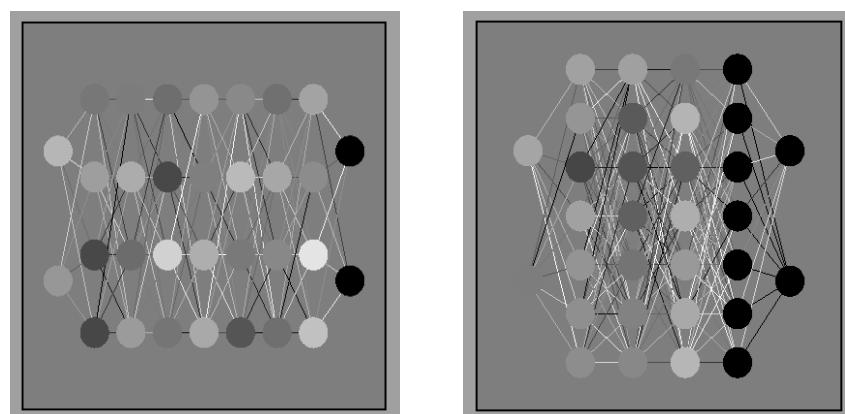
כל האלמנטים בהמחשה צבעם מתאר את מצבם :

- עברו משקלים צבע כהה משמעו משקל נמוך, וצבע בהיר משמעו משקל גבוה.
- עברו נוירונים צבע כהה משמעו סך משקלים יוצאים נמוך, וצבע בהיר משמעו סך משקלים יוצאים גבוה.

להלן דוגמאות לרשת הנוירונים של מערכת זו במצבים שונים:



להלן דוגמאות לרשתות נוירוניים שניתן לבנות עבור מערכות יותר מורכבות:



מחלקה הנירון - Neuron

מחלקה זו מייצגת אובייקט מסווג נירון. לכל נירון משקלים היוצאים ממנו, באמצעותם הוא מעביר מידע אל השכבות הבאות.

תפקידו של כל נירון באופן פשוט מאוד והוא לקבל מידע, להפעיל עליו את המשקלים הנקנסים אליו, להפעיל את פונקציית האקטיבציה, ולהגדיר את ערכו.

קוד המחלקה

```
class Neuron
{
public:
    Neuron(unsigned numOutputs, unsigned myIndex, Vector2f position, float radius);

    void Draw(RenderWindow * window);
    void Update(float elapsedTime, vector<VertexArray> * weight_lines_, int * firstWeightIndex);
    void Reset();

    vector<Connection> GetWeights();
    void SetWeights(vector<Connection> weights);

    void SetOutputValue(double val) { output_value_ = val; }
    double GetOutputValue() const { return output_value_; }
    void FeedForward(const Layer &prevLayer);
    void UpdateInputWeights(Layer &prevLayer);
    Vector2fGetPosition(){return circle_->GetPosition();}
    void Mutate(float mutationRate);

private:
    // randomWeight: 0 - 1
    static double randomize_weight() { return random() / double(RAND_MAX); }
    double output_value_;
    vector<Connection> output_weights_;
    unsigned my_index_;

    CircleShape * circle_;
};
```

פונקציית FeedForward

לשם העברת חישוב ערכו של ניירון בתקהיליק FeedForward של הרשות כולה, משתמש בפונקציית FeedForward במחלקה זו המתרכזת בתקהיליק העברת מידע ועיבודו דרך ניירון בודד.

פונקציה זו מכפילה את ערכו של ניירון כלשהו בשכבה הקודמת במשקל המחבר בין לביין הניירון הנוכחי, ומוסיפה את התוצאה לסכום. סכום זה הוא ערכו של הניירון.

קוד הפונקציה

```
void Neuron::FeedForward(const Layer &prevLayer) {
    // the sum of outputs from previous layer
    // into this node
    float sum = 0.f;

    // for each neuron in previous layer
    for (int n = 0; n < prevLayer.size(); ++n)
    {
        sum += prevLayer[n].GetOutputValue() *
               prevLayer[n].output_weights_[my_index_].weight;
    }
    output_value_ = Neuron::transfer_function(sum);
}
```

פונקציית Mutate

לאחר שרשת נבנתה על בסיס רשת שנבחרה מהדרו הקודם באמצעות פונקציית PoolSelection, נרצה לבצע שינויים רנדומליים קטנים על מנת "להרחיב אופקים" ולהגדיל את תחום הפעולה של הרשותות שלנו. תקהיליק זה נקרא "מוטציה" (Mutation), ובשביל לבצעו משתמש בפונקציית Mutate על כל רשות. פונקציה זו פשוות מבצעת שינויים קטנים באופן רנדומלי על משקלים היוצאים מנירון. על מנת לבצע את השינוי, משתמש בפונקציית עוזר הנקראת mutator.

קוד ההפונקציה

```
double mutator(double value, float rate){
    if((rand() / double(RAND_MAX)) < rate){
        return value + (rand() / double(RAND_MAX)) * 2 - 1;
    }else{
        return value;
    }
}
```

אימפלמנטציה במערכת

פעילות רשת הנוירוניים

האימפלמנטציה (הטמעה) של רשת הנוירונים במערכת נעשית בחלוקת Cycle. מחלוקת Cycle אחראית על מחזורי הפאות.

במערכת שלי יהיו 2 קלטים עبور כל פאה:

1. צפיפות הנתיב המקסימלית מבין כל הנתיבים השبيיכים לפאה
2. אורך התווך המקסימלי מבין כל הנתיבים השבייכים לפאה

1-2 פלטים:

1. נקודות העדיפות של הפאה
2. זמן המחזור של הפאה

המטרה היא לחת את הפלט הראשון, ובאמצעותו למיין את סדר הפעלתן של פאות לפי נקודות העדיפות של כל פאה, ולחת את הפלט השני ובאמצעותו לקבוע את זמן המחזור של כל פאה.

לאחר ביצוע כל סימולציה בשימוש ברשת הנוירונים לקביעת מדיניות הפאות, נבדקת תוצאת הסימולציה.

תוצאת הסימולציה ביא זמן ביצעה עבור מספר מכוניות מסוימים. ככלمر, ככל שהסימולציה עברה בזמן קצר יותר, כך זרימת המכוניות הייתה גבוהה יותר, וכך רשת הנוירונים ביצעה בצורה טוביה יותר.

כאמור, תפקידה של רשת הנוירונים הוא לחשב את נקודות העדיפות של כל פאה וזמן מחזורה בהתאם לנתחיה.

чисוב זה נעשה בפונקציה `(calculate_priority()`

`calculate_priority()`

פונקציה זו מעבירה את הנתונים של כל פאה דרך רשת הנוירונים, ובאמצעות הפלט של הרשת היא קובעת את עדיפות הפאה ואת זמן המחזור שלה.

בכדי לקבל את מערך הנתונים של פאה, נישית קריאה לפונקציית `(GetInputValues()` המ>Returns את צפיפות הנתיב המקסימלית ואת אורך התווך המקסימלי בפה.

```
void Phase::GetInputValues(vector<double> & inputValues)
{
    inputValues[0] = GetMaxLaneDensity();
    inputValues[1] = GetMaxQueueLength() ;
}
```

לאחר קבלת הפלט מרשת הנוירונים, עדיפות הפazaה וזמן מחזורה נקבע. עם עדיפות הפazaה, מתבצע מחזור הפazaות בצורה שתוארה בפרק העוסק בחלוקת המחזור - Cycle.

אם צפיפות פazaה היא 0, אין טעם להعبر מידע זה דרך רשת הנוירונים שכן אוטומטית עדיפות הפazaה היא 0.

```
/// calculate the phases priority using neural network
void Cycle::calculate_priority() {

    for (int p = 0; p < number_of_phases_ - 1; p++)
    {
        // get input values
        phases_[p]->GetInputValues(input_values_);

        if(input_values_[0] > 0)
        {
            Settings::NeuralNetwork.FeedForward(input_values_);
            Settings::NeuralNetwork.GetResults(output_values_);

            phases_[p]->SetPhasePriority(output_values_[0]);
            phases_[p]->SetCycleTime(clamp(float(output_values_[1])) *
Settings::MaxCycleTime, Settings::MinCycleTime, Settings::MaxCycleTime));
        }
        else
        {
            phases_[p]->SetCycleTime(Settings::MinCycleTime);
            phases_[p]->SetPhasePriority(0);
        }
    }
}
```

בסיומה של סימולציה, ניתן להציג. ציון זה, הוא מתאר את זרימת התchapורה בזומת במהלך הסימולציה (מכונית בשניה), שימושה ישרות מדיניות הפazaות שמנוהלת על ידי רשת הנוירונים, וכן ניתן לומר שציון הסימולציה הוא בעצם ציון רשת הנוירונים.

לאחר הריצה של דור, כלומר מספר מוגדר של סימולציות המשמשות ברשותות שונות, נרצה ליצור דור חדש ולמש את עקרונות האלגוריתם הגנטי.

תהליך זה מתבצע בתוך פונקציית העדכון של המערכת, הנמצאת בחלוקת Engine.

```

for (Set *s : sets_) {
    // when an update on a set returns true
    // it means that a simulation has finished
    if (s->Update(elapsedTime)) {
        if (s->IsFinished()) {
            SetFinished();
        } else {
            SimulationFinished();
            // set the new score as result
            float result = s->GetLastSimulationResult();
            if (Settings::RunBestNet){
                Net::BestNet.Update(elapsedTime);
            } else {
                Net::CurrentNet->SetScore(result);
                Net::CurrentNet->Update(elapsedTime);

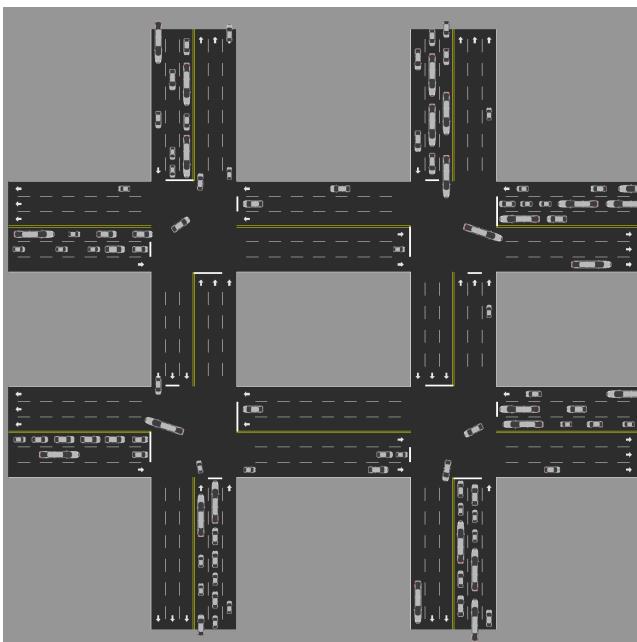
                if (result > Net::HighScore){
                    Net::HighScore = result;
                    Net::BestNet = *(Net::CurrentNet);
                }
                Net::CurrentNetIndex++;
            }
        }
        // check if generation is done
        if (Net::CurrentNetIndex == Net::PopulationSize){
            // if is, create a new generation
            Net::NextGeneration();
        }
    }

    Net::CurrentNet = &(Net::Generation[Net::CurrentNetIndex]);

    if (Settings::DrawNnProgression){
        cout << "Gen no. " << Net::GenerationCount + 1
            << " Net no. " << Net::CurrentNetIndex << "/"
            << Net::PopulationSize
            << " High Score: " << Net::HighScore << endl;
    }
}
}
}

```

תוצאות



עבור מפה בסיסית בצורה הבאה:

בקצב הופעת מכוניות בעלות מסלול רנדומלי של מכוניות בכל 0.9 שניות

לאחר הריצה של כ 200 סימולציות בלבד, אחת אחרי השנייה, השתרפה זרימת התחבורה בזומת זה ממוצע של **0.3** מכוניות לשניה לממוצע של **0.55** מכוניות בשניה.

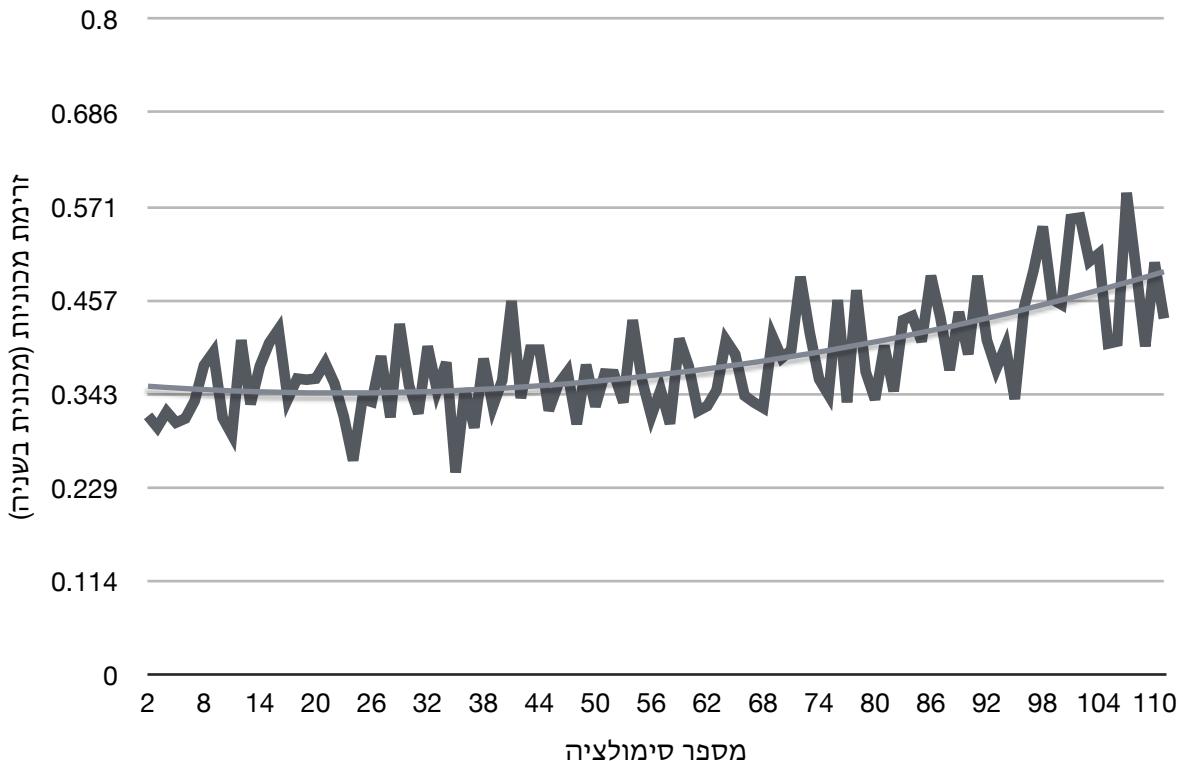
זהו שיפור של כ-80% בזרימת התחבורה בזומת.

המשמעות של שיפור בגודל זה היא:

ביום שלם, במקום כ 100,000 מכוניות שהיו עוברות בזומת ללא המערכת, יעברו כעת 000 180,000 באמצעותה.

להלן גרפ השיפור של רשת הנירוניים:

תרשים זרימת התחבורה



ממשק המשתמש

מבנה הממשק

ממשק המשתמש בנוי מ-2 חלקים: הסימולציה, והכליים. למען בניית ממשק המשתמש, נעשה שימוש בספריה ובכליים של Qt.

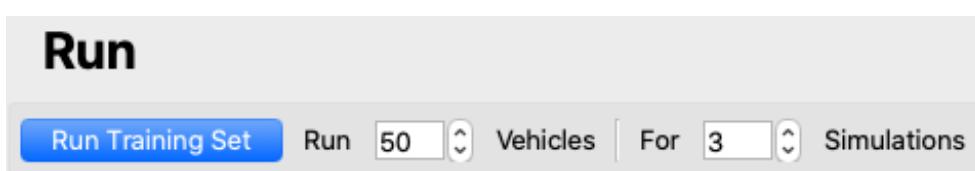
Qt הוא כלי לפיתוח GUI (ממשק משתמש וויזואלים) מהמובילים בעולם. הכלי זהה מורכב מתוכנת עזר הנקראת **QtCreator**, Qt, ומספרית C++.

תוכנת העזר QtCreator היא סביבת תכנון, אך מזינה גם פונקציונליות של Drag-and-Drop. כלומר גיררת אלמנטים מסוימים אל תוך חלון מוגדר מראש ב כדי ליצור ממשק משתמש בצורה קלה ופешטה.

לאחר יצירת ממשק המשתמש באמצעות QtCreator, לכל אובייקט ניתן שם כלשהו, ונוצרת עבורו פונקציה המגיבה לאיורו כלשהו. זהו למעשה תהליך הדומה(Event -> Slot) ב- C++ .

לדוגמה, עם כפתור בשם Button1, נוכל להגיד שבניתנית סימן (Signal) של לחיצה, תקרה הפונקציה (Slot) בשם on_Button1Clicked(). איתה נוכל להגיד כרגע כראינו. כאן נכנסת ספריית Qt, המאפשרת לבנות פונקציות שונות עבור מצבים שונים של כל האלמנטים השונים במשתמש.

דוגמה לכפתור הרצת סט אימונים של רשת הנוירונים:



שם הכפתור הכהול הוגדר בעת יצרתו כ-RunSetButton. לאחר מכן slot חדש בשם on_RunSetButton_clicked() נוצר בשם on_RunSetButton_clicked(). כל מה שנוטר לעשות הוא לכתוב את תוכן槽, שהוא בעצם פונקציה, בהתאם למה שאנו רוצים שכפתור זה יבצע.

במקרה זה, כפתור זה מורה על יצירת סט חדש של סימולציות לפי הנתונים המוזנים ב-2 התיבות שבעצם ימין. כמו כן, גם לתיבות אלו יש שם במערכת וניתן לגשת אל הערך שלהם.

להלן פונקציית ה *Slot* בשם *on_RunSetButton_clicked*

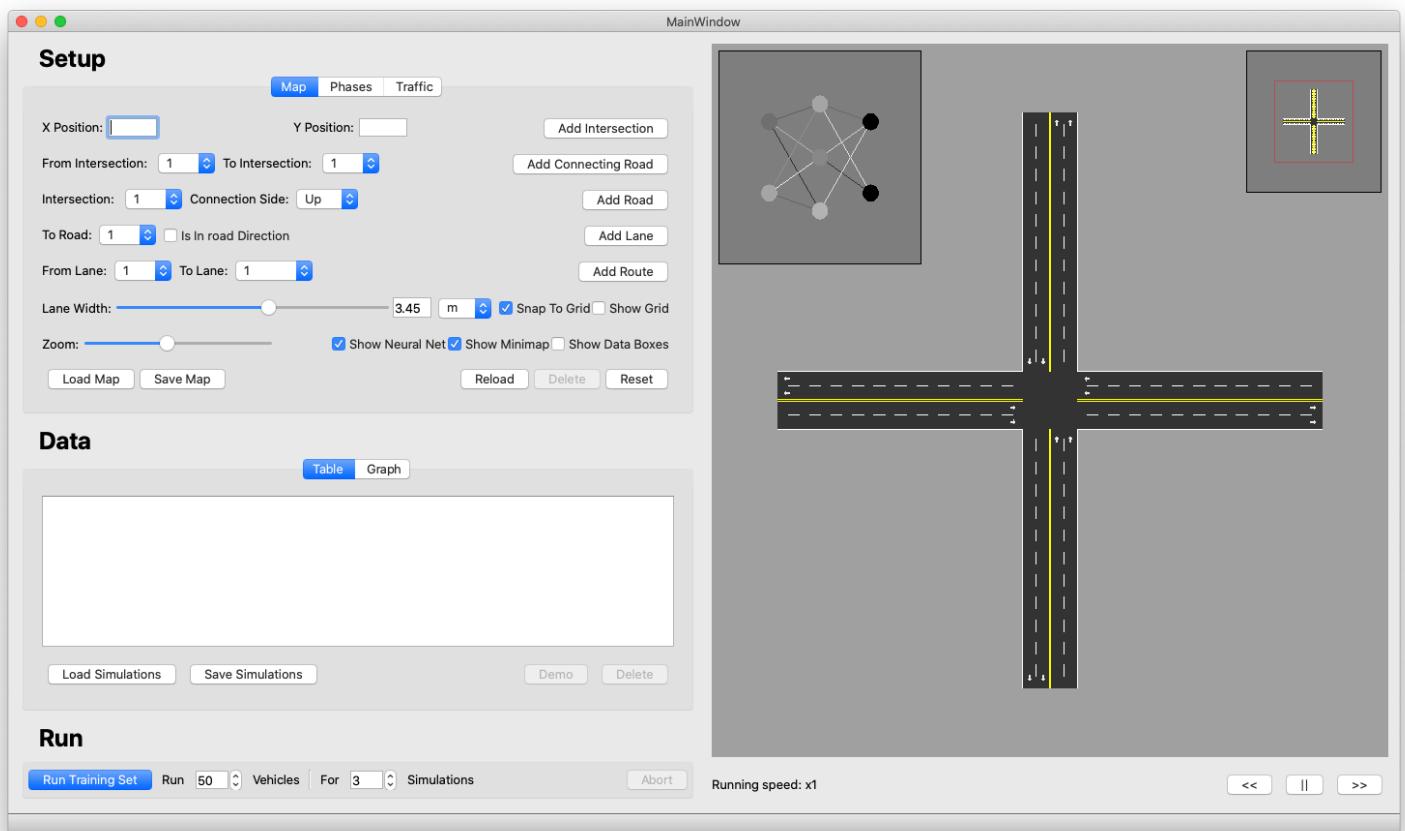
```
void MainWindow::on_RunSetButton_clicked() {  
  
    int vehicleCount = ui->CarCountSpinBox->value();  
    int generations = ui->SimulationCountSpinBox->value();  
  
    if (!Simulation::SimRunning)  
    {  
        SimulatorEngine->RunSet(vehicleCount, generations);  
        ui->AbortButton->setEnabled(true);  
    } else  
    {  
        ui->statusbar->showMessage(tr("Another simulation is  
                currently running, please wait for it to finish"), 5000);  
    }  
}
```

פונקציה זו פשוט מבודד ניגשת את נתוני הרצת הסט שהוזנו, ששםם הוא `CarCountSpinBox` ו-`SimulationCountSpinBox`, ושולחת למנוע הסימולציה `SimulatorEngine` את הבקשה להריץ סט חדש בהתאם להגדרות שניתנו.

אם הרצאה זו נכשלה, לדוגמה במצב בו כבר יש סימולציה במהלך ביצוע, תודפס הודעה הת חיתית | החלוון שנראית כך:

Another simulation is currently running, please wait for it to finish

כפי שנאמר, ממשק המשתמש מחולק ל-2 חלקים עיקריים, והם נראים כך:



הגדרת וניהול נתונים

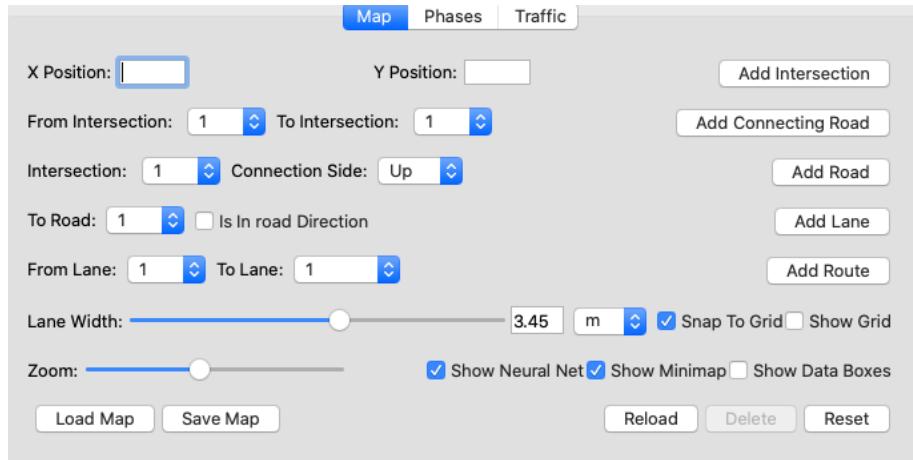
כמו שנitinן לראות, החלק השמאלי שהוא חלק הגדרת והפעלת הסימולציה, מחולק ל-3 תת-חלקים:

Setup .1

בחלק זה מוגדרים ומנווהלים כל הנתונים על הריצת הסימולציה ועל המפה. ניתן ליצור מפה חדשה, לטען ולשמור מפות, ניתן לשולט על המצלמה, להגדיר הגדרות הקשורות לפאות, הרמזהורים, וכלי הרכב.

חלק זה מחולק ל-3 תת-חלקים, כאשר כל חלק אחראי על הגדרת מרכיב אחר של המערכת.

Map 1.1



להלן זה אפשר יצירת מפה חדשה בצורה הבאה:

ראשית, יש להגדיר מקום של צומת על ידי הזנת נתוני מקום או לחיצה כפולה על מקום כלשהו על המפה.

אם זאת מפה עם כמה צמתים, ניתן להוסיף כביש אחד ביןיהם.

כמו כן, ניתן להוסיף רגיל המתחבר לצומת נבחר אחד מ-4 הכוונים האפשריים.

לאחר מכן, ניתן להוסיף כל כביש. כל נתיב שנוסף, יש להגדיר אם הוא בכיוון הכביש או נגדו. להזדהה, כיוון הכביש הוא כלפי חוץ הצומת.

לאחר הוספת כל הנתיבים, יש להוסיף את כל מסלולי התנועה האפשריים על המפה. על מנת להוסיף מסלול חדש, כל שיש לעשות זה לבחור נתיב מקור ונתיב יעד.

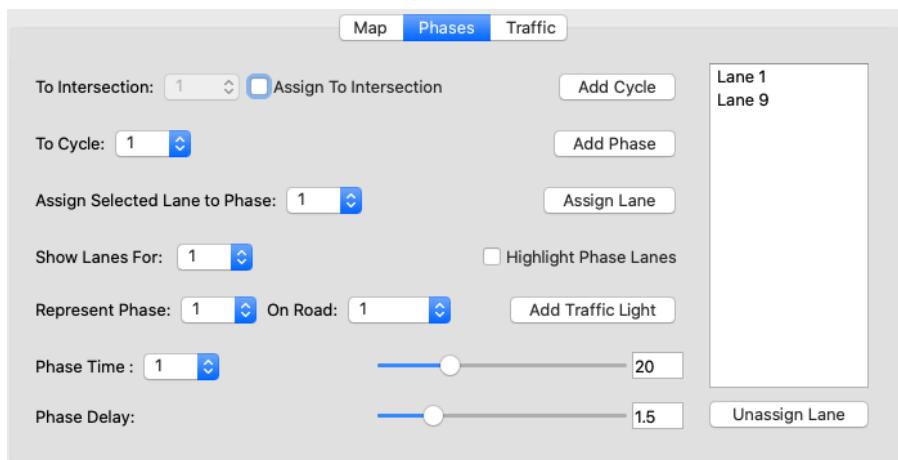
אם לא ידועים מהו מספר הזאות של נתיב מסוים, ולאיזה צומת וכביש הוא שייך, ניתן לחוץ לחיצה כפולה עליו על המפה, ומידע זה ירשם בתחתית החלון.

Selected: Lane {16}, Road {4}, Intersection {1}, Direction {270}

כמו כן, ניתן להגדיר את רוחב הכבישים באמצעות גירות הסלידר. הגדרת היקום של המצלמה נעשית גם כן באמצעות סליידר.

ניתן להראות להסתיר מרכיבים מהסימולציה כמו מיני-מפה, ורשת הנוירונים הויזואלית, באמצעות תיבות הבחירה.

Phases 1.2



להלן זה נตอน שליטה על כל מרכיבי הפאזה ב�פה.

בכדי לבנות מערכת שליטה על �פה שבנוינו, נתחל לבנות מחזור של פאות.

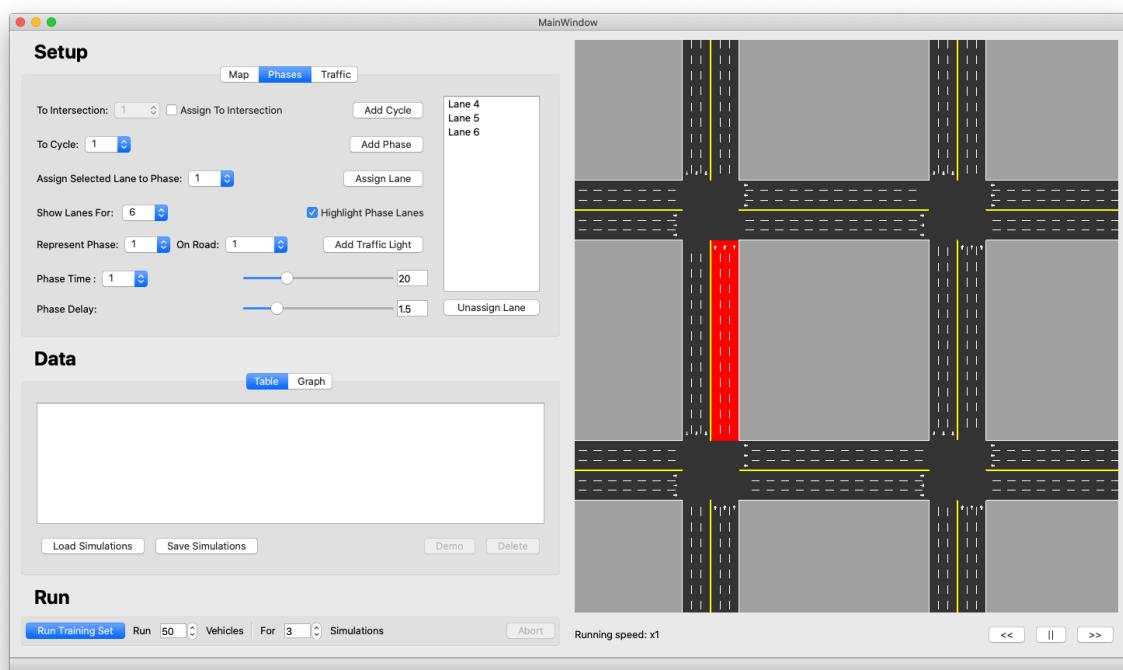
להזכרה, ניתן ליצור כמה מחזורי פאות על מנת ליזור מוחזרים עצמאיים שאינם תלויים אחד בשני.

בכדי ליצור מחזור, נבחר ראשית אם אנחנו רוצים לשיך אותו לצומת מסוים. שיק לצומת מסוים מראה שמחזור זה שולט על צומת אחד. לאחר הבחירה, נלחץ על הכפתור **Add Cycle**.

בכדי להוסיף פאה אל פאה, נחבר את מספר הפאה ונלחץ **Add Phase**.

יש להגדיר את הנתיבים הקשורים לכל פאה, וכן לעשות זאת נבחר נתיב כלשהו על ידי לחיצה כפולה עליו ב�פה, ונבחר לאיזה פאה להוסיף אותו.

ניתן גם להמחייב אילו נתיבים שייכים לאיזו פאה בטבלה שילד, ועל ידי בחירת אפשרות **Highlight Phase Lanes**, ניתן גם לצביע את הנתיבים הללו ב�פה. כך זה נראה:



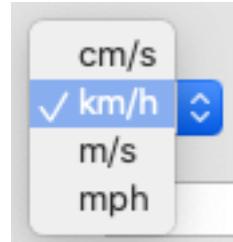
ניתן להוסיף כמו כן רמות נוספים אשר מייצגים פאזה כלשהי, ולהזדקיק אותם לכיבוש מסויים. כמו כן, ניתן לשלוט על זמני הפאזות באופן ידני באמצעות הסליידרים.



Traffic 1.3

בחלק זה נוכל להגדיר את המהירותים המקסימליות של כל סוגי הרכב, ואת תדריות יצירתן.

בפינה הימנית העליונה, ישנה הבחירה של יחידות המידה.



האפשרויות הן:

- ס"מ בשניה
- קילומטר בשעה
- מטר בשניה
- מייל בשעה

בהתאם לבחירה, יופיעו המהירותים המקסימליות ביחידות המתאימות.

את תדריות יצירת כלי הרכב ניתן לשנות באמצעות סליידר. יחידות תדריות יצירת כלי הרכב היא מכונית כל x מיל-שניות.

כמו כן, קיימות מספר תיבות בחירה, שכל אחת מהן שולטת על משווה אחד:

1. **Draw Texture** - לא לצייר את הטקסטורות של המכוניות

2. **Multi Color** - לצייר את המכוניות במגוון צבעים או לא

3. **Follow Selected Car** - כאשר כלי רכב נבחר על ידי לחיצה כפולה, המצלמה יכולה לעקוב אחריו.

4. **Show Routes** - לצייר/לא לצייר את כל המסלולים.

5. **Show Lane Blocks** - להמחיש האם נתיב סגור או לא על ידי צייר קו לבן בקו העצירה.

6. **Density Color** - צביעת הנתיבים בצבע המתאים לצפיפות הנתיב.

Data .2

חלק זה אחראי על הצגת המידע על הסימולציות הרצו, שמירת סימולציה או טעינתם.
ישןן 2 רפרנסנטציות למידע הסימולציות:

Table 2.1

צורת הציגה הראשונה היא טבלה. הטבלה מציגה את הסימולציות ואת המידע שליהן בצורה של טבלה פשוטה.

ID	Start Time	End Time	Simulated Time	Vehicle Count	Score
2	Tue Apr 14 ...	Tue Apr 14 ...	179.199	50	0.27902
3	Tue Apr 14 ...	Tue Apr 14 ...	182.527	50	0.273933
4	Tue Apr 14 ...	Tue Apr 14 ...	174.591	50	0.286384
5	Tue Apr 14 ...	Tue Apr 14 ...	177.663	50	0.281432

Load Simulations Save Simulations Demo Delete

כל שורה מייצגת את נתוניה של סימולציה אחרת.
ישנה האפשרות לבחור סימולציה כלשהיא על ידי בחירת השורה שלה, ולהריץ דמונסטרציה שלה או למחוק אותה מהטט על ידי לחיצה על הכפתור המתאים.

Graph 2.2

צורת הציגה השנייה היא גרפ. הגרף מייצג את ציון הסימולציה, ביחס למספרה. ציר ה-X הוא מספר הסימולציה, וציר ה-Y הוא ציון הסימולציה.
הגרף מביא המכחשה וויזואלית מעולה להתקדמות למידת רשת הנירוניים.



Run .3

חלק זה אחראי על ייצרת סט חדש והרצתו.

ישנם 2 האדרות פשוטות שיש להגדיר לפני שמריצים סט אימוניים:

1. מספר המכוניות שיופיעו בכל סימולציה

2. מספר הסימולציות שיופיעו בסט.

ניתן להפעיל סט, וניתן לוותר עליו באמצעות לחיצה על כפתור abort.

כפתור זה עוצר את פעילות הסט, מוחק את הסימולציה שהייתה באמצע הרצה, אך לא מוחק את כל הסימולציות שכבר בוצעו.



במקרה זה, ירוץו בסט 3 סימולציות ובכל אחת מהן 50 מכוניות.

החלק הימני של חלון המערכת. בחלק זה מוצגת המפה וכל כלי הרכיב, וכליים נוספים כמו מיני-מפה ורשת נירוניים ויזואלית.

חלון זה מנוהל על ידי מחלקת המנוע Engine, אך מחלוקת זו כמו שאר מחלוקות הסימולציה בנויות באמצעות ספריית SFML בעיקר. אז איך נעשה בכך לשלב חלון SFML אל תוך חלון Qt?

אומנם אלו הן 2 ספריות שונות לחהוטין, אך ישנה דרך לשלב את השתיים. הדרך לעשות זאת היא להפוך את מחלקת Engine למחלוקת של SFML בצורה טהורה, למחלוקת משולבת.

לשם כך, ניצור מחלוקת חדשה המייצגת אובייקט המציג חלון המשלב את 2 העולמות. מחלוקת זו נקראת **QSfmlCanvas**, והיא יורשת מ-2 המחלוקות הבאות:

1. **QWidget** - מחלוקת רכיב ממשקי משתמש היכי בסיסית בQt. אובייקט היורש ממחלוקת זו ניתן לשימוש נתונים של Qt ללא בעיה, ונitin להתייחס אליו כמו אל שאר רכיבי Qt.
2. **RenderWindow** - מחלוקת חלון הציור של SFML. במערכת טהורה של SFML, זהו החלון עלייו כל המערכת מצוירת. אל אובייקט שיורש ממחלוקת זו, ניתן להתייחס כמרכיב sfml כשאר המרכיבים, ונitin לצייר כאילו הוא רכיב עצמאי ואינו קשור למשק משתמש בלבדו.

מחלקה זו דורשת ומגדירה מחדש כמה מאפייניה הבסיסיות של **QWidget**, על מנת שנוכל לצייר אותה כחלק ממשק המשתמש.

מחלקה זו יורשת מחלוקת המנוע Engine, האחראית בצורה מלאה על הפעלת הסימולציה.

```
class QSfmlCanvas : public QWidget, public RenderWindow
{
    Q_OBJECT

public:
    QSfmlCanvas(QWidget *Parent, int logicIntervalTime = 500,
                int drawIntervalTime = 60);
    ~QSfmlCanvas() { std::cout << "QSFML Canvas destroyed" << std::endl; }

private:
    virtual void on_init() {}
    virtual void logic_cycle() {}
    virtual void draw_cycle() {}

    QWidget *parent_;

    QPaintEngine *paintEngine() const override;
    void showEvent(QShowEvent *) override;
```

```

void paintEvent(QPaintEvent *) override;

bool is_init_;

private slots:
    void redraw();

protected:
    QTimer logic_timer_;
    QTimer draw_timer_;
};


```

חלון הסימולציה

חלקו של חלון הסימולציה במשק המשתמש הוא האפשרות לבחור נתיבים ולגרור את המצלמה על פני המפה.

מעבר לכך, כל פעילות חלון הסימולציה מתוארת בפרק "הסימולציה".

ביבליוגרפיה

https://doc.qt.io/	דוקומנטציית ספריית Qt
https://www.sfml-dev.org/documentation/2.5.1/	דוקומנטציית ספריית SFML
http://www.cplusplus.com/	אתר cplusplus
https://en.cppreference.com/w/	אתר cppreference
https://stackoverflow.com/	אתר stackexchange
http://gamecodeschool.com/sfml-projects/	אתר gamecodeschool

קוד המערכת

כעת יוצג בקוד של הפרויקט שלו. רק קבצי ה-.cpp יוצגו, מכיוון שרוב קבצי ה-.cpp של המחלקות הראשיות כבר הוצגו בפרויקט, ואין בהם קוד של ממש אלה ורק הצהרות משתנים ופונקציות.

הקוד יחולק לחתתי-קבוצות, לפי היררכיית הפרויקט עצמו, ולפי שמות התקינות.

main.cpp

```
//  
// main.cpp  
// SimulatorSFML  
  
//  
// Created by Samuel Arbibe on 18/11/2019.  
// Copyright © 2019 Samuel Arbibe. All rights reserved.  
  
  
#include <iostream>  
#include <QApplication>  
#include <QWidget>  
#include <QFrame>  
#include "ui/mainwindow.h"  
  
int main(int argc, char **argv) {  
    srand((time(nullptr)));  
    vector<unsigned> topology;  
  
    // input neurons : max lane density, max queue length  
    topology.push_back(2);  
    // hidden neurons  
    topology.push_back(3);  
    // output neurons : priority points, phase time  
    topology.push_back(2);  
  
    for(unsigned i = 0; i < Net::PopulationSize; i++)  
    {  
        Net::Generation.emplace_back(topology);  
        Net::CurrentNet = &(Net::Generation[Net::CurrentNetIndex]);  
    }  
  
    QApplication Application(argc, argv);  
  
    auto *main = new MainWindow();  
    main->show();  
  
    return QApplication::exec();  
}
```

sim / map

Cycle.cpp

```
//  
// Created by Samuel Arbibe on 06/04/2020.  
  
  
#include "Cycle.hpp"  
  
int Cycle::CycleCount = 0;  
  
/// a compare function to compare phases priority  
bool compare_priority(Phase *first, Phase *second) {  
    return (first->GetPriorityScore() < second->GetPriorityScore());  
}  
  
Cycle::Cycle(int cycleNumber, Intersection *intersection) {  
    cycle_number_ = cycleNumber;  
    intersection_ = intersection;  
    number_of_phases_ = 0;  
  
    input_values_ = vector<double>(2, 0);  
    output_values_ = vector<double>(1, 0);  
}  
  
Cycle::~Cycle() {  
    for (Phase *p : phases_)  
    {  
        delete p;  
    }  
  
    cout << "Cycle number " << cycle_number_ << " has been deleted." << endl;  
}  
  
/// update function  
void Cycle::Update(float elapsedTime) {  
    for (Phase *p : phases_)  
    {  
        p->Update(elapsedTime);  
    }  
  
    cycle_phases();  
}  
  
/// reload all phases in this cycle  
void Cycle::ReloadCycle() {  
  
    for (Phase *p : phases_)
```

```

    {
        p->ReloadPhase();
    }
}

/// add a phase to this cycle
Phase *Cycle::AddPhase(int phaseNumber, float cycleTime) {

    Phase *temp;

    if (phaseNumber == 0)
    {
        phaseNumber = Phase::PhaseCount + 1;
    }

    temp = new Phase(phaseNumber, this->cycle_number_, cycleTime);

    phases_.push_back(temp);

    ++number_of_phases_;
    ++Phase::PhaseCount;

    if (Settings::DrawAdded)
        cout << "phase " << phaseNumber << " added" << endl;

    return temp;
}

/// get a phase by phase ID
/*
Phase *Cycle::GetPhase(int phaseNumber) {

    for (Phase *p : phases_)
    {
        if (p->GetPhaseNumber() == phaseNumber)
        {
            return p;
        }
    }

    return nullptr;
}
*/

/// calculate the phases priority using neural network
///////////////////////////////
/// \brief
///
/// Calculates the priority of each lane using the neural
/// network.

```

```

/// Runs the data of each phase as input through the NN,
/// and sets its priority from the NN outputs.
///
///////////
void Cycle::calculate_priority() {

    for (int p = 0; p < number_of_phases_ - 1; p++)
    {
        // get input values
        phases_[p]->GetInputValues(input_values_);

        if (input_values_[0] > 0)
        {
            if(Settings::RunBestNet)
            {
                Net::BestNet.FeedForward(input_values_);
                Net::BestNet.GetResults(output_values_);
            }
            else
            {
                Net::CurrentNet->FeedForward(input_values_);
                Net::CurrentNet->GetResults(output_values_);
            }

            phases_[p]->SetPhasePriority(output_values_[0]);
            phases_[p]->SetCycleTime(clamp(
                float(output_values_[1]) * Settings::MaxCycleTime,
                Settings::MinCycleTime,
                Settings::MaxCycleTime));
        } else
        {
            phases_[p]->SetCycleTime(Settings::MinCycleTime);
            phases_[p]->SetPhasePriority(0);
        }
    }
}

/// cycle the phases by the phase array order.
///////////
/// \brief
///
/// all the phases but the active one are constantly evaluated
/// and sorted by the score they have been given by the NN
/// when phase is finished, it gets swapped with the previous one in the array
/// to advance and start the next phase in order.
///
///////////

void Cycle::cycle_phases() {
    // if cycle has a minimum of 2 phases
}

```

```

if (number_of_phases_ >= 2)
{
    // when current phase is closed, advance to next phase and open it
    if (!phases_.back()->GetIsOpen())
    {
        Phase *backPhase = phases_[number_of_phases_ - 1];
        phases_[number_of_phases_ - 1] = phases_[number_of_phases_ - 2];
        phases_[number_of_phases_ - 2] = backPhase;

        phases_[number_of_phases_ - 1]->Open();

        //Net::NeuralNetwork.printNet();
    }
    // constantly sort the list by their priority score
    else
    {
        // calculate the priority of each phase
        calculate_priority();
        // sort(arr[0:-2])
        partial_sort(phases_.begin(),
                    phases_.end() - 1,
                    phases_.end() - 1,
                    compare_priority);
    }
}

/// draw function
void Cycle::Draw(RenderWindow *window) {

    for (Phase *p : phases_)
    {
        p->Draw(window);
    }
}

```

Intersection.cpp

```
//  
//  Intersection.cpp  
//  SimulatorSFML  
//  
//  Created by Samuel Arbibe on 23/11/2019.  
//  Copyright © 2019 Samuel Arbibe. All rights reserved.  
//  
  
#include "Intersection.hpp"  
  
int Intersection::IntersectionCount = 0;  
  
Intersection::Intersection(Vector2f position, int intersectionNumber)  
    : RectangleShape() {  
  
    intersection_number_ = intersectionNumber;  
    position_ = position;  
    width_ = 0;  
    height_ = 0;  
    current_vehicle_count_ = 0;  
    total_vehicle_count_ = 0;  
    number_of_roads_ = 0;  
  
    this->setOrigin(width_ / 2.f, height_ / 2.f);  
    this->setPosition(position_);  
    this->setOutlineColor(WhiteColor);  
    this->setFillColor(LaneColor);  
    this->setOutlineThickness(0.1f);  
    this->setSize(Vector2f(width_, height_));  
}  
  
Intersection::~Intersection() {  
    // every road belongs to 2 intersections  
    // to avoid deleting them both, we make sure we delete it once  
    for (Road *road : roads_)  
    {  
        if (road->GetIntersectionNumber(0) == this->intersection_number_)  
        {  
            delete road;  
        }  
    }  
  
    if (Settings::DrawDelete)  
        cout << "Intersection " << intersection_number_ << " deleted" << endl;  
}  
  
/// add a road to an intersection  
Road *Intersection::AddRoad(int roadNumber, int connectionSide, float length) {
```

```

if (!roadNumber)
{
    roadNumber = Road::RoadCount + 1;
}

roads_.push_back(new Road(roadNumber,
                         intersection_number_,
                         connectionSide,
                         GetPositionByConnectionSide(connectionSide),
                         length,
                         float(connectionSide - 1) * 90.f));

number_of_roads_++;
Road::RoadCount++;

if (Settings::DrawAdded)
    std::cout << "Road " << roadNumber << " added" << endl;

return roads_[number_of_roads_ - 1];
}

///////////////////////////////
/// \brief Overload of binary operator !=
///
/// This function adds a new road connecting between 2 intersection.
/// The first intersection is this intersection, the second intersection is
/// passed through to this function.
///
/// \param roadNumber (int) - the ID of the road
/// \param connectionSide1 (int) - connection side to intersection 1
/// \param connectionSide2 (int) - connection side to intersection 2
/// \param connectedIntersection (* Intersection) - pointer to the connected
intersection
///
///
/// \return a pointer to the added connecting road
///
/////////////////////////////
Road *Intersection::AddConnectingRoad(int roadNumber,
                                       int connectionSide1,
                                       int connectionSide2,
                                       Intersection *connectedIntersection) {
if (!roadNumber)
{
    roadNumber = Road::RoadCount + 1;
}

roads_.push_back(new Road(roadNumber,
                          this->intersection_number_,
                          connectedIntersection->intersection_number_,

```

```

        connectionSide1,
        connectionSide2,
        this->GetPositionByConnectionSide(connectionSide1),
        connectedIntersection
            ->GetPositionByConnectionSide(connectionSide2),
            float(connectionSide1 - 1) * 90));
    }

connectedIntersection->roads_.push_back(roads_[number_of_roads_]);
connectedIntersection->number_of_roads_++;

number_of_roads_++;
Road::RoadCount++;

if (Settings::DrawAdded)
    std::cout << "Connecting Road " << roadNumber
        << " added between intersections "
        << this->intersection_number_ << " <--> "
        << connectedIntersection->intersection_number_ << "" << endl;

    return roads_[number_of_roads_ - 1];
}

/// get a lane by laneNumber
Lane *Intersection::GetLane(int laneNumber) {
    Lane *temp;

    for (int i = 0; i < number_of_roads_; i++)
    {
        if ((temp = roads_[i]->GetLane(laneNumber)) != nullptr)
        {
            return temp;
        }
    }

    return nullptr;
}

/// get a road by roadNUmber
Road *Intersection::GetRoad(int roadNumber) {
    Road *temp;
    for (int i = 0; i < number_of_roads_; i++)
    {
        temp = roads_[i];
        if ((temp->GetRoadNumber()) == roadNumber)
        {
            return temp;
        }
    }

    return nullptr;
}

```

```

}

/// get road connected to an intersection by its connection side
Road *Intersection::GetRoadByConnectionSide(int connectionSide) {
    for (int i = 0; i < number_of_roads_; i++)
    {
        if (roads_[i]->GetConnectionSide(0) == connectionSide
            || (roads_[i]->GetIsConnecting()
                && roads_[i]->GetConnectionSide(1) == connectionSide))
        {
            return roads_[i];
        }
    }

    return nullptr;
}

/// add a regular lane to an intersection
Lane *Intersection::AddLane(int laneNumber,
                            int roadNumber,
                            bool isInRoadDirection) {
    Road *temp = GetRoad(roadNumber);
    Lane *l;

    l = temp->AddLane(laneNumber, isInRoadDirection);

    return l;
}

/// get the position of an intersection connection by side
Vector2f Intersection::GetPositionByConnectionSide(int connectionSide) {
    switch (connectionSide)
    {
        case UP: return Vector2f(position_.x, position_.y - height_ / 2.f);

        case RIGHT: return Vector2f(position_.x + width_ / 2.f, position_.y);

        case DOWN: return Vector2f(position_.x, position_.y + height_ / 2.f);

        case LEFT: return Vector2f(position_.x - width_ / 2.f, position_.y);

        default: return position_;
    }
}

///////////
/// \brief
///
/// This function re-assigns road position if their structure has been changed.

```

```

/// When a lane is added or removed for example, the whole intersection's roads
/// need to be relocated
///
///////////////////////////////
void Intersection::ReAssignRoadPositions() {
    for (int i = 0; i < number_of_roads_; i++)
    {
        // if road starts from this intersection, adjust start position
        if (roads_[i]->GetIntersectionNumber(0) == this->intersection_number_)
        {
            roads_[i]->UpdateStartPosition(GetPositionByConnectionSide(roads_[i]
                ->GetConnectionSide(
                    0)));
        } else
        {
            if (roads_[i]->GetIsConnecting()
                && roads_[i]->GetIntersectionNumber(1)
                == this->intersection_number_)
            {
                // if road connects this intersection to another
                roads_[i]
                    ->UpdateEndPosition(GetPositionByConnectionSide(roads_[i]
                        ->GetConnectionSide(
                            1)));
            }
        }
    }
}

/// Reload the intersection after any change
void Intersection::ReloadIntersection() {
    for (Road *r : roads_)
    {
        r->ReloadRoadDimensions();
    }

    // update intersection dimensions
    Road *r1 = GetRoadByConnectionSide(1);
    Road *r2 = GetRoadByConnectionSide(2);
    Road *r3 = GetRoadByConnectionSide(3);
    Road *r4 = GetRoadByConnectionSide(4);

    if (r1 != nullptr)
        width_ = r1->GetWidth();
    else if (r3 != nullptr)
        width_ = r3->GetWidth();
    if (r2 != nullptr)

```

```

        height_ = r2->GetWidth();
    else if (r4 != nullptr)
        height_ = r4->GetWidth();

    if (r3 != nullptr && r3->GetWidth() > width_)
    {
        width_ = r3->GetWidth();
    }

    if (r4 != nullptr && r4->GetWidth() > height_)
    {
        height_ = r4->GetWidth();
    }

    this->setSize(Vector2f(width_, height_));
    this->setOrigin(width_ / 2, height_ / 2);

    ReAssignRoadPositions();
}

/// check if a road in this intersection has been selected
Lane *Intersection::CheckSelection(Vector2f position) {
    // for each intersection in map
    Lane *temp;
    for (Road *road : roads_)
    {
        // if selection found
        temp = road->CheckSelection(position);
        if (temp != nullptr)
            return temp;
    }
    return nullptr;
}

/// return the lane count in this intersection
int Intersection::GetLaneCount() {
    int sum = 0;

    for (Road *road : roads_)
    {
        sum += road->GetLaneCount();
    }

    return sum;
}

/// update
void Intersection::Update(float elapsedTime) {
    current_vehicle_count_ = 0;
    total_vehicle_count_ = 0;
}

```

```

for (Road *r : roads_)
{
    r->Update(elapsedTime);
    current_vehicle_count_ += r->GetCurrentVehicleCount();
    total_vehicle_count_ += r->GetTotalVehicleCount();
}
}

///////////
/// \brief
///
/// This deletes a given lane by its ID from its intersection.
/// function sends down the order to delete a lane to its road,
/// and if that road is empty after the deletion,
/// delete that road from the intersection
///
/// \param laneNumber (int) - ID of the lane to delete
/// \param otherIntersection (* Intersection) - if lane is connecting between 2
/// intersections, a pointer to the other intersection is passed here.
///
/// \return true if successfully deleted, else false.
///
/////////
bool Intersection::DeleteLane(int laneNumber, Intersection *otherIntersection) {
    Lane *targetLane = this->GetLane(laneNumber);
    if (targetLane != nullptr)
    {
        Road *targetRoad = this->GetRoad(targetLane->GetRoadNumber());
        if (targetRoad != nullptr)
        {
            targetRoad->DeleteLane(laneNumber);
            // check if no lanes are left in road.
            // if so, delete road as well
            if (targetRoad->GetLaneCount() == 0)
            {
                auto it = find(roads_.begin(), roads_.end(), targetRoad);
                this->roads_.erase(it);

                if (otherIntersection != nullptr)
                {
                    auto xt = find(otherIntersection->roads_.begin(),
                                   otherIntersection->roads_.end(),
                                   targetRoad);
                    otherIntersection->roads_.erase(xt);
                    otherIntersection->number_of_roads_--;
                }
            }

            delete targetRoad;
            number_of_roads_--;
        }
    }
}

```

```

        }
        return true;
    }
}
return false;
}

/// draw the intersection and everything that belongs to it
void Intersection::Draw(RenderWindow *window) {
    (*window).draw(*this);

    for (int i = 0; i < number_of_roads_; i++)
    {
        roads_[i]->Draw(window);
    }
}

```

Lane.cpp

```

//
// Lane.cpp
// SimulatorSFML
//
// Created by Samuel Arbibe on 03/12/2019.
// Copyright © 2019 Samuel Arbibe. All rights reserved.
//

#include "Lane.hpp"

int Lane::LaneCount = 0;

Lane::Lane(int laneNumber,
           int roadNumber,
           int intersectionNumber,
           Vector2f startPosition,
           float length,
           float direction,
           bool isInRoadDirection) {

    is_in_road_direction_ = isInRoadDirection;
    lane_number_ = laneNumber;
    road_number_ = roadNumber;
    intersection_number_ = intersectionNumber;
    start_pos_ = startPosition;
    width_ = Settings::LaneWidth;
    length_ = length;
}

```

```

direction_ = fmod(direction, 360.f);
is_blocked_ = false;
total_vehicle_count_ = 0;
phase_number_ = 0;
density_ = 0;
selected_ = false;
queue_length_ = 0;

// calculate end position:
Vector2f lengthVec;

Transform t;
t.rotate(direction);

lengthVec = t.transformPoint(Vector2f(0.f, -1.f)) * length;

end_pos_ = lengthVec + start_pos_;

// init rectangle shape
this->setOrigin(width_ / 2.f, 0.f);
this->setPosition(start_pos_);
this->setRotation(direction_ + 180);
this->setSize(Vector2f(width_, length_));
if (Settings::LaneDensityColorRamping)
{
    ColorRamp();
} else
{
    this->setFillColor(LaneColor);
}

// create direction arrow shape
create_arrow_shape(t);

// create lane block rectangle shape
create_block_shape();

data_box_ = new DataBase(end_pos_);
data_box_->AddData("ID", float(lane_number_));
data_box_->AddData("Dens", 0);
data_box_->AddData("Qlen", 0);
}

Lane::~Lane() {
    if (Settings::DrawDelete)
        cout << "Lane " << lane_number_ << " deleted" << endl;
}

/// create the arrow shapes indicating the lane's direction
void Lane::create_arrow_shape(Transform t) {

```

```

arrow_shape_.setPointCount(7);

t.scale(width_ / 4, width_ / 4);
arrow_shape_.setPoint(0, end_pos_ - t.transformPoint(Vector2f(0.f, -2.f)));

t.rotate(-45);
arrow_shape_.setPoint(1,
                     arrow_shape_.getPoint(0)
                     - t.transformPoint(Vector2f(0.f, -1.f)));

t.rotate(90);
arrow_shape_.setPoint(6,
                     arrow_shape_.getPoint(0)
                     - t.transformPoint(Vector2f(0.f, -1.f)));

t.rotate(45);
arrow_shape_.setPoint(2,
                     arrow_shape_.getPoint(1)
                     - t.transformPoint(Vector2f(0.f, -1.0f)));

t.rotate(180);
arrow_shape_.setPoint(5,
                     arrow_shape_.getPoint(6)
                     - t.transformPoint(Vector2f(0.f, -1.f)));

t.rotate(90);
arrow_shape_.setPoint(3,
                     arrow_shape_.getPoint(2)
                     - t.transformPoint(Vector2f(0.f, -1.f)));

arrow_shape_.setPoint(4,
                     arrow_shape_.getPoint(5)
                     - t.transformPoint(Vector2f(0.f, -1.f)));

arrow_shape_.setFillColor(WhiteColor);
}

/// create the block shape shown in a blocked lane
void Lane::create_block_shape() {
    lane_block_shape_ = RectangleShape();

    float block_height = 15.f;

    lane_block_shape_.setOrigin(Vector2f(this->width_ / 2, block_height));
    lane_block_shape_.setPosition(this->end_pos_);
    lane_block_shape_.setSize(Vector2f(this->width_, block_height));
    lane_block_shape_.rotate(this->getRotation());
    lane_block_shape_.setFillColor(Color::White);
}

```

```

/// update
void Lane::Update(float elapsedTime) {

    density_ = vehicles_in_lane_.size() / Settings::ConvertSize(PX, M, length_);

    if (Settings::DrawRoadDataBoxes)
    {
        data_box_->SetData("Qlen", queue_length_);
        data_box_->SetData("Dens", density_ * 100);
    }

    // disable lane coloring if needed
    if (selected_)
    {
        this->setFillColor(Color::Red);
    } else if (Settings::LaneDensityColorRamping)
    {
        this->ColorRamp();
    } else
    {
        this->setFillColor(LaneColor);
    }
}

///////////////////////////////
/// \brief
///

/// Colors the lane in a color that corresponds to its density.
/// The higher the lane density, the red-er its color is
/// The lower the lane density, the blue-er its color is
///

void Lane::ColorRamp() {

    // normalizing the lane density.
    // density is vehicle-per-meter

    float value = (density_ / Settings::MaxDensity);
    if (value > 1.f)
        value = 1.f;
    float r, g, b;

    Settings::GetHeatMapColor(value, &r, &g, &b);

    this->setFillColor(Color(r, g, b, 255));
}

/// try to set the current queue length in this lane
void Lane::SetQueueLength(float queueLength) {
    if (queueLength > queue_length_)

```

```

    {
        queue_length_ = queueLength;
    }
}

/// set this lane as selected
void Lane::Select() {
    selected_ = true;
}

/// set this lane as unselected
void Lane::Unselect() {
    selected_ = false;
}

/// draw the road
void Lane::Draw(RenderWindow *window) {
    window->draw(*this);
    window->draw(arrow_shape_);

    if (Settings::DrawLaneBlock && is_blocked_)
    {
        window->draw(lane_block_shape_);
    }

    if (Settings::DrawRoadDataBoxes)
        data_box_->Draw(window);
}

```

Light.cpp

```
//  
// Created by Samuel Arbibe on 20/02/2020.  
  
  
#include "Light.hpp"  
  
int Light::LightCount = 0;  
  
Light::Light(int lightNumber, int phaseNumber, Road *parentRoad) {  
    parent_road_ = parentRoad;  
    phase_number_ = phaseNumber;  
    light_number_ = lightNumber;  
    state_ = RED;  
  
    circles_.push_back(new CircleShape());  
    circles_.push_back(new CircleShape());  
    circles_.push_back(new CircleShape());  
  
    this->setOutlineColor(Color(169, 169, 169, 255));  
    this->setOutlineThickness(4.f);  
  
    UpdatePosition();  
  
    data_box_ = new DataBox(this->getPosition());  
    data_box_->AddData("ID", light_number_);  
    data_box_->AddData("Phase", phase_number_);  
    data_box_->AddData("State", state_);  
}  
  
Light::~Light() {  
    delete data_box_;  
  
    if (Settings::DrawDelete)  
        cout << "Light " << light_number_ << " deleted" << endl;  
}  
  
///////////\brief  
///  
/// Function makes sure a light is always on the side of it's parent road.  
/// when a road changes position, this light is moved accordingly  
///  
///////////  
void Light::UpdatePosition() {  
    Vector2f roadPos = parent_road_->GetStartPosition();  
    float direction = parent_road_->GetRoadDirection();
```

```

Transform t;
t.rotate(direction - 90);
float margin = parent_road_->GetWidth() / 2 + 30;
t.scale(margin, margin);
Vector2f sideVector = t.transformPoint(Settings::BaseVec);

t = Transform();
t.rotate(direction);
t.scale(20, 20);
Vector2f marginVector = t.transformPoint(Settings::BaseVec);
Vector2f position = roadPos + sideVector + marginVector;

direction -= 180;

this->setSize(Vector2f(40, 100));
this->setOrigin(getSize().x / 2, 0);
this->setPosition(position);
this->setFillColor(Color::Black);
this->setRotation(direction);

float radius = this->getSize().x / 2 - 7;

Vector2f basePos = this->getPosition();

t = Transform();
t.rotate(direction - 180);
// set axis relative to basePos and direction
Vector2f yDir = t.transformPoint(Settings::BaseVec);

// set location relative to new axis
Vector2f yMargin = Vector2f(radius * 2 + 7, radius * 2 + 7);
t = Transform();
t.scale(yMargin);
yMargin = t.transformPoint(yDir);

t = Transform();
t.scale(0.5, 0.5);

Vector2f first = basePos + t.transformPoint(yMargin);

circles_[0]->setOrigin(radius, radius);
circles_[0]->setFillColor(Color::Black);
circles_[0]->setPosition(first);
circles_[0]->setRadius(radius);

circles_[1]->setOrigin(radius, radius);
circles_[1]->setFillColor(Color::Black);
circles_[1]->setPosition(first + yMargin);
circles_[1]->setRadius(radius);

```

```

        circles_[2]->setOrigin(radius, radius);
        circles_[2]->setFillColor(Color::Black);
        circles_[2]->setPosition(first + yMargin + yMargin);
        circles_[2]->setRadius(radius);
    }

    /// update
    void Light::Update(float elapsedTime) {
        circles_[0]->setFillColor(Color::Black);
        circles_[1]->setFillColor(Color::Black);
        circles_[2]->setFillColor(Color::Black);

        switch (state_)
        {
        case RED:circles_[0]->setFillColor(Color::Red);
            break;
        case ORANGE:circles_[1]->setFillColor(Color::Yellow);
            break;
        case GREEN:circles_[2]->setFillColor(Color::Green);
            break;
        }

        if (Settings::DrawLightDataBoxes)
            data_box_->SetData("State", state_);
    }

    /// draw
    void Light::Draw(RenderWindow *window) {
        window->draw(*this);

        for (int i = 0; i < 3; i++)
        {
            window->draw(*circles_[i]);
        }

        if (Settings::DrawLightDataBoxes)
            data_box_->Draw(window);
    }
}

```

Map.cpp

```
//  
// Created by Samuel Arbibe on 28/12/2019.  
  
  
#include <QStringList>  
#include "Map.hpp"  
  
int Map::MapCount = 0;  
  
Map::Map(int mapNumber, int width, int height) {  
    if (mapNumber == 0)  
    {  
        mapNumber = ++MapCount;  
    }  
    map_number_ = mapNumber;  
    width_ = width;  
    height_ = height;  
    SelectedLane = nullptr;  
    current_phase_index_ = 0;  
    number_of_cycles_ = 0;  
    number_of_intersections_ = 0;  
}  
  
Map::~Map() {  
    for (Intersection *inter : intersections_)  
    {  
        delete inter;  
    }  
  
    for (Route *route : routes_)  
    {  
        delete route;  
    }  
  
    for (Cycle *cycle : cycles_)  
    {  
        delete cycle;  
    }  
  
    Route::RouteCount = 0;  
    Lane::LaneCount = 0;  
    Road::RoadCount = 0;  
    Phase::PhaseCount = 0;  
    Cycle::CycleCount = 0;  
    Intersection::IntersectionCount = 0;
```

```

if (Settings::DrawDelete)
    cout << "map " << map_number_ << " deleted" << endl;
}

/// add an intersection to the map
Intersection *Map::AddIntersection(int intersectionNumber, Vector2f position) {
    SelectedLane = nullptr;

    if (!intersectionNumber)
    {
        intersectionNumber = Intersection::IntersectionCount + 1;
    }

    intersections_.push_back(new Intersection(position, intersectionNumber));

    number_of_intersections_++;
    Intersection::IntersectionCount++;

    if (Settings::DrawAdded)
        std::cout << "Intersection " << intersectionNumber << " added" << endl;

    return intersections_[number_of_intersections_ - 1];
}

/// add a road to a specific intersection
Road *Map::AddRoad(int roadNumber,
                    int intersectionNumber,
                    int connectionSide,
                    float length) {
    SelectedLane = nullptr;

    Intersection *temp = GetIntersection(intersectionNumber);
    Road *tempRoad = nullptr;

    if (temp)
    {
        tempRoad = temp->AddRoad(roadNumber, connectionSide, length);
    }

    this->ReloadMap();

    return tempRoad;
}

/// add a lane to a specific road
Lane *Map::AddLane(int laneNumber, int roadNumber, bool isInRoadDirection) {
    SelectedLane = nullptr;

    Road *tempRoad = GetRoad(roadNumber);
    Intersection *temp = nullptr;
}

```

```

Lane *tempLane = nullptr;

if (tempRoad)
{
    temp = GetIntersection(tempRoad->GetIntersectionNumber());
}

if (temp)
{
    tempLane = temp->AddLane(laneNumber, roadNumber, isInRoadDirection);
}

this->ReloadMap();

return tempLane;
}

/// add a cycle of phases and attach it to an intersecion
Cycle *Map::AddCycle(int cycleNumber, int intersectionNumber) {
    if (cycleNumber == 0)
    {
        cycleNumber = Cycle::CycleCount + 1;
    }

    Intersection *inter = nullptr;

    if (intersectionNumber != 0)
    {
        inter = GetIntersection(intersectionNumber);
    }

    Cycle *temp = new Cycle(cycleNumber, inter);
    cycles_.push_back(temp);

    ++Cycle::CycleCount;
    ++number_of_cycles_;

    cout << "Cycle number " << cycleNumber << " added successfully" << endl;

    return temp;

    cout << "could not add cycle as intersection " << intersectionNumber
        << " wasnt found." << endl;

    return nullptr;
}

///////////////////////////////
/// \brief
///

```

```

/// Creates a new road connecting between 2 given intersections
///
/// \param roadNumber (int) - the ID of the road to be added.
/// \param intersectionNumber1 (int) - ID of the first intersection
/// \param intersectionNumber2 (int) - ID of the second intersection
///
/// \return the connecting road added
///
///////////
Road *Map::AddConnectingRoad(int roadNumber,
                               int intersectionNumber1,
                               int intersectionNumber2) {
    SelectedLane = nullptr;

    Intersection *inter1 = GetIntersection(intersectionNumber1);
    Intersection *inter2 = GetIntersection(intersectionNumber2);

    if (inter1 == nullptr || inter2 == nullptr)
    {
        cerr << "one of the given intersections was not found..." << endl;
        return nullptr;
    }

    // if intersection do not align on one of the axis, return error
    if ((int(inter1->getPosition().x) != int(inter2->getPosition().x) &&
         int(inter1->getPosition().y) != int(inter2->getPosition().y)) ||
        ((int(inter1->getPosition().x) == int(inter2->getPosition().x) &&
         int(inter1->getPosition().y) == int(inter2->getPosition().y)))
    {
        cerr << "the intersections must align on one of the axis" << endl;
        return nullptr;
    }

    pair<ConnectionSides, ConnectionSides> connections;
    connections =
        AssignConnectionSides(inter1->getPosition(), inter2->getPosition());

    Road *temp = inter1->AddConnectingRoad(roadNumber,
                                             connections.first,
                                             connections.second,
                                             inter2);

    this->ReloadMap();
    return temp;
}

///////////
/// \brief
///
/// Creates a new route between 2 given lanes

```

```

/// 
/// \param from (int) - the source lane
/// \param to (int) - the target lane
///
/// \return a pointer to the created route
///
///////////////////////////////
Route *Map::AddRoute(int from, int to) {
    Lane *fromLane = GetLane(from);
    Lane *toLane = GetLane(to);

    if (fromLane != nullptr && toLane != nullptr)
    {
        Route *r = new Route(fromLane, toLane);
        routes_.emplace_back(r);

        if (Settings::DrawAdded)
            cout << "Route added from " << r->FromLane->GetLaneNumber()
                << " to " << r->ToLane->GetLaneNumber() << endl;
        return r;
    } else
    {
        cout
            << "could not create possible route, as one of the roads was not found"
            << endl;
        return nullptr;
    }
}

/// add a phase to the mac
Phase *Map::AddPhase(int phaseNumber, int cycleNumber, float cycleTime) {

    Cycle *cycle = GetCycle(cycleNumber);
    Phase *temp = nullptr;

    if (cycle != nullptr)
    {
        if ((temp = cycle->AddPhase(phaseNumber, cycleTime)) != nullptr)
        {
            return temp;
        }
    }

    cout << "Could not add phase as the given cycle doesnt exist." << endl;

    return nullptr;
}

/// assign a light to a phase to attach to road
Light *Map::AddLight(int lightNumber, int phaseNumber, int parentRoadNumber) {

```

```

Light *temp = nullptr;
Phase *myPhase = GetPhase(phaseNumber);
Road *parentRoad = GetRoad(parentRoadNumber);

// assign position relative to parent road
if (myPhase != nullptr && parentRoad != nullptr)
{
    temp = myPhase->AddLight(lightNumber, parentRoad);
    if (Settings::DrawAdded)
        cout << "light " << temp->GetLightNumber() << " added to phase "
        << phaseNumber << endl;
    return temp;
}

cout << "could not add light as phase or parent road don't exist" << endl;
return temp;
}

/// set the phase time of a given phase
bool Map::SetPhaseTime(int phaseNumber, float phaseTime) {
    Phase *phase = GetPhase(phaseNumber);

    if (phase != nullptr)
    {
        phase->SetCycleTime(phaseTime);
        cout << "Phase time of phase " << phaseNumber << " successfully set."
        << endl;
        return true;
    }

    cout << "Could not set phase time as phase doesnt exist." << endl;
    return false;
}

///////////////////////////////
/// \brief
///
/// Assign a lane to a phase, to make the lane depend on
/// the phase status.
///
/// \param phaseNumber (int) - the phase ID
/// \param laneNumber (int) - the lane ID to assign to the phase
///
/// \return true if successful, else false
///
/////////////////////////////
bool Map::AssignLaneToPhase(int phaseNumber, int laneNumber) {
    Phase *temp = GetPhase(phaseNumber);
    Lane *lane = GetLane(laneNumber);
}

```

```

if (temp != nullptr && lane != nullptr)
{
    temp->AddLane(lane);
    if (Settings::DrawAdded)
        cout << "lane " << lane->GetLaneNumber()
            << " added to phase " << temp->GetPhaseNumber() << endl;
    return true;
}
cout << "could not add lane to phase as phase or lane don't exist." << endl;
return false;
}

///////////////////////////////
/// \brief
///
/// Removes a route by a lane number/
/// If the given lane is a part of sum routes, as source or as target,
/// delete it from the routes array.
/// This function is when a lane is deleted
///
/// \param laneNumber (int) - the lane to remove the routes including it
///
/// \return true if sucessfull, else false
///
/////////////////////////////
bool Map::RemoveRouteByLaneNumber(int laneNumber) {
    Lane *laneToRemove = GetLane(laneNumber);

    if (laneToRemove != nullptr)
    {
        auto it = routes_.begin();

        // while there are cars to delete;
        while (it != routes_.end())
        {
            // if is to be deleted
            if ((*it)->FromLane->GetLaneNumber() == laneNumber ||
                (*it)->ToLane->GetLaneNumber() == laneNumber)
            {
                Route *temp = (*it);
                int routeNumber = temp->GetRouteNumber();
                it = routes_.erase(it);

                delete temp;

                if (Settings::DrawActive)
                    cout << "route " << routeNumber << " deleted." << endl;
            } else
            {
                it++;
            }
        }
    }
}

```

```

        }
    }

    return true;
} else
{
    //cout << "could not delete this route. the deleted lane could not be
found." << endl;
    return false;
}

}

/// remove a lane from the control of a phase
bool Map::UnassignLaneFromPhase(int laneNumber) {

    Lane *lane = GetLane(laneNumber);

    if (lane->GetPhaseNumber() != 0)
    {
        if (lane != nullptr)
        {
            Phase *phase = GetPhase(lane->GetPhaseNumber());

            if (phase->UnassignLane(lane))
            {
                return true;
            }
        }
        else
        {
            cout
                << "could not Unassign lane from phase, as lane does not belong to
any phase"
                << endl;
        }
    }
    return false;
}

///////////
/// \brief
///
/// Finds all the starting lanes and assign them to the starting_lane array.
/// A starting lane is one that its road is not connecting,
/// and it is against the road direction.
///
/////////
void Map::FindStartingLanes() {
    // for a lane to be a starting lane, it has to be in a non-connecting road,
}

```

```

// and it has to have isInRoadDirection = false;
starting_lanes_.clear();

for (Intersection *inter : intersections_)
{
    for (Road *road : *inter->GetRoads())
    {
        if (!road->GetIsConnecting())
        {
            for (Lane *lane : *road->GetLanes())
            {
                if (!lane->GetIsInRoadDirection())
                {
                    starting_lanes_.push_back(lane);
                }
            }
        }
    }
}

/// returns a possible starting lane
Lane *Map::GetPossibleStartingLane() {
    if (starting_lanes_.empty())
        return nullptr;
    int randomIndex = rand() % starting_lanes_.size();
    return starting_lanes_[randomIndex];
}

///////////
/// \brief
///
/// Selects all the routes where a vehicle is about to pass through
///
/// \param instructionSet (list<Lane *>) - a list of the lanes the vehicle is
/// going to pass through
///
/////////
void Map::SelectRoutesByVehicle(list<Lane *> *instructionSet) {
    Route *r = nullptr;
    std::list<Lane *>::const_iterator to = instructionSet->begin();
    std::list<Lane *>::const_iterator from = to;
    to++;
    for (; to != instructionSet->end(); ++to)
    {
        r = GetRouteByStartEnd((*from)->GetLaneNumber(),
                               (*to)->GetLaneNumber());
        if (r != nullptr)
        {
            selected_routes_.push_back(r);
        }
    }
}

```

```

        r->SetSelected(true);
    }
    from = to;
}
}

///////////
/// \brief
///
/// Creates a random track composed of a chain of routes
///
/// \return a list of lane, that will act as an instruction
/// set for a vehicle
///
/////////
list<Lane *> *Map::GenerateRandomTrack() {
    // find a random starting point
    Lane *l = GetPossibleStartingLane();
    if (l == nullptr)
    {
        cout << "no starting lanes available." << endl;
        return nullptr;
    }
    // find a starting route from starting lane
    Route *r = GetPossibleRoute(l->GetLaneNumber());

    if (r == nullptr)
    {
        cout << "no routes available. please add them to the map" << endl;
        return nullptr;
    }

    // while new routes to append are available
    // new routes will be searched starting from the previous route end
    list<Lane *> *track = new list<Lane *>();
    Lane *lastLane = nullptr;

    while (r != nullptr)
    {
        track->push_back(r->FromLane);
        lastLane = r->ToLane;
        r = GetPossibleRoute(r->ToLane->GetLaneNumber());
    }
    if (lastLane != nullptr)
    {
        track->push_back(lastLane);
    }

    return track;
}

```

```

///////////
/// \brief
///
/// Return a random possible route that sources from a given lane
///
/// \param fromLane (int) - source lane ID
///
/// \return pointer to a randomly chosen possible route
///
/////////
Route *Map::GetPossibleRoute(int fromLane) {
    Lane *myLane = GetLane(fromLane);
    vector<Route *> possibleRoutes;

    if (myLane != nullptr)
    {
        for (Route *r : routes_)
        {
            if (r->FromLane->GetLaneNumber() == myLane->GetLaneNumber())
            {
                possibleRoutes.push_back(r);
            }
        }
    }
    if (possibleRoutes.empty())
    {
        return nullptr;
    }
    int randomIndex = rand() % possibleRoutes.size();
    return possibleRoutes[randomIndex];
}

///////////
/// \brief
///
/// return a route that starts and ends from given lanes
///
/// \param from (int) - source lane ID
/// \param to (int) - target lane ID
///
/// \return a pointer to a corresponding route
///
/////////
Route *Map::GetRouteByStartEnd(int from, int to) {
    for (Route *r : routes_)
    {
        if (r->FromLane->GetLaneNumber() == from &&
            r->ToLane->GetLaneNumber() == to)
        {

```

```

        return r;
    }
}
return nullptr;
}

/// get intersection by intersectionNumber
Intersection *Map::GetIntersection(int intersectionNumber) {
    Intersection *temp;

    for (Intersection *inter : intersections_)
    {
        if ((temp = inter)->GetIntersectionNumber() == intersectionNumber)
        {
            return temp;
        }
    }

    cout << "error : intersection not found in map..." << endl;

    return nullptr;
}

/// get intersection by lane number
vector<Intersection *> Map::GetIntersectionByLaneNumber(int laneNumber) {
    Lane *l = this->GetLane(laneNumber);
    Road *r = this->GetRoad(l->GetRoadNumber());

    vector<Intersection *> retVal;

    if (r->GetIsConnecting())
    {
        retVal.push_back(this->GetIntersection(r->GetIntersectionNumber(1)));
    }
    retVal.push_back(this->GetIntersection(r->GetIntersectionNumber(0)));
    return retVal;
}

/// get road by roadNumber
Road *Map::GetRoad(int roadNumber) {
    Road *temp;

    for (Intersection *inter : intersections_)
    {
        if ((temp = inter->GetRoad(roadNumber)) != nullptr)
        {
            return temp;
        }
    }
}

```

```

cout << "error : road not found in map..." << endl;

return nullptr;
}

/// get lane by lane number
Lane *Map::GetLane(int laneNumber) {
    Lane *temp;

    for (Intersection *inter : intersections_)
    {
        if ((temp = inter->GetLane(laneNumber)) != nullptr)
        {
            return temp;
        }
    }

    cout << "error : lane not found in map..." << endl;

    return nullptr;
}

// get cycle by cycle number
Cycle *Map::GetCycle(int cycleNumber) {

    for (Cycle *cycle : cycles_)
    {
        if (cycle->GetCycleNumber() == cycleNumber)
        {
            return cycle;
        }
    }

    cout << "error : lane not found in map..." << endl;

    return nullptr;
}

/// get phase by phase number
Phase *Map::GetPhase(int phaseNumber) {

    for (Cycle *c : cycles_)
    {
        for (Phase *p : *c->GetPhases())
        {
            if (p->GetPhaseNumber() == phaseNumber)
            {
                return p;
            }
        }
    }
}

```

```

}

cout << "error : phase not found in map..." << endl;
return nullptr;
}

///////////////////////////////
/// \brief
///
/// return the according connection sides to the given intersection positions.
///
/// \param pos1 (int) - intersection 1 position
/// \param pos2 (int) - intersection 2 position
///
/// \return a pair of connection sides
///
/////////////////////////////
pair<ConnectionSides, ConnectionSides> Map::AssignConnectionSides(Vector2f pos1,
Vector2f pos2) {
    ConnectionSides con1, con2;
    if (pos1.x > pos2.x)
    {
        con1 = ConnectionSides::LEFT;
        con2 = ConnectionSides::RIGHT;
    } else if (pos1.x < pos2.x)
    {
        con1 = ConnectionSides::RIGHT;
        con2 = ConnectionSides::LEFT;
    } else if (pos1.y > pos2.y)
    {
        con1 = ConnectionSides::UP;
        con2 = ConnectionSides::DOWN;
    } else if (pos1.y < pos2.y)
    {
        con1 = ConnectionSides::DOWN;
        con2 = ConnectionSides::UP;
    }

    pair<ConnectionSides, ConnectionSides> connections(con1, con2);
    return connections;
}

///////////////////////////////
/// \brief
///
/// check if given position is inside the bounds of a lane object
///
/// \param position (int) - the mouse position
///
/// \return a pointer to a selected lane. nullptr if none selected

```

```

/// 
///////////////////////////////
Lane *Map::CheckSelection(Vector2f position) {
    // for each intersection in map
    Lane *temp;
    for (Intersection *inter : intersections_) {
        // if selection found
        temp = inter->CheckSelection(position);
        if (temp != nullptr)
            return temp;
    }
    return nullptr;
}

/// Reload all intersection in this map
void Map::ReloadMap() {

    // unselect all the selected lanes
    UnselectAll();

    for (Intersection *i : intersections_)
    {
        i->ReloadIntersection();
    }

    for (Cycle *c : cycles_)
    {
        c->ReloadCycle();
    }

    for (Route *r : routes_)
    {
        r->ReloadRoute();
    }

    FindStartingLanes();
}

/// update, for future use
void Map::Update(float elapsedTime) {
    for (Intersection *i : intersections_)
    {
        i->Update(elapsedTime);
    }

    for (Cycle *c : cycles_)
    {
        c->Update(elapsedTime);
    }
}

```

```

}

/// select all the lanes that were assigned to a given phase
void Map::SelectLanesByPhase(int phaseNumber) {
    Phase *p = GetPhase(phaseNumber);

    if (p != nullptr)
    {
        UnselectAll();

        for (Lane *l : *p->GetAssignedLanes())
        {
            selected_lanes_.push_back(l);
            l->Select();
        }
    }
}

/// unselect al selected items
void Map::UnselectAll() {
    // unselect selected lane
    if (SelectedLane != nullptr)
    {
        SelectedLane->Unselect();
        SelectedLane = nullptr;
    }

    for (Lane *l : selected_lanes_)
    {
        l->Unselect();
    }

    for (Route *r : selected_routes_)
    {
        r->SetSelected(false);
    }

    selected_lanes_.clear();
    selected_routes_.clear();
}

/// return road count in this map
int Map::GetRoadCount() {
    int sum = 0;

    for (Intersection *inter : intersections_)
    {
        sum += inter->GetRoadCount();
    }
}

```

```

        return sum;
    }

/// return lane count int this map
int Map::GetLaneCount() {
    int sum = 0;

    for (Intersection *inter : intersections_)
    {
        sum += inter->GetLaneCount();
    }

    return sum;
}

/// return a vector of all the existing lanes
vector<Phase *> *Map::GetPhases() {
    vector<Phase *> *phases = new vector<Phase *>();

    for (Cycle *cycle : cycles_)
    {
        for (Phase *p : *cycle->GetPhases())
        {
            phases->push_back(p);
        }
    }

    return phases;
}

/// return a vector of all the existing lanes
vector<Lane *> *Map::GetLanes() {
    vector<Lane *> *lanes = new vector<Lane *>();

    for (Intersection *inter : intersections_)
    {
        for (Road *road : *inter->GetRoads())
        {
            for (Lane *lane : *road->GetLanes())
            {
                lanes->push_back(lane);
            }
        }
    }

    return lanes;
}

/// return a vector of all the existing lights
vector<Light *> *Map::GetLights() {

```

```

vector<Light *> *lights = new vector<Light *>();

for (Phase *p : *GetPhases())
{
    for (Light *l : *p->GetLights())
    {
        lights->push_back(l);
    }
}

return lights;
}

///////////
/// \brief
///
/// Delete a lane from the map.
/// this function has to make sure everything that uses this lane, is deleted as
well.
///
/// \param laneNumber (int) - the ID of the lane to be deleted
///
/// \return true if successful, else false
///////////

bool Map::DeleteLane(int laneNumber) {

    vector<Intersection *>
        targetIntersections = GetIntersectionByLaneNumber(laneNumber);
    Lane *lane = GetLane(laneNumber);

    if (!targetIntersections.empty())
    {
        // delete all routes that go through this lane
        RemoveRouteByLaneNumber(laneNumber);

        // delete this lane from all phases it belongs to
        UnassignLaneFromPhase(laneNumber);
        // delete the given lane
        // if lane's road is connecting, send other intersection as well to handle
deletion
        if (targetIntersections.size() > 1)
        {
            targetIntersections[0]
                ->DeleteLane(laneNumber, targetIntersections[1]);
        } else
        {
            targetIntersections[0]->DeleteLane(laneNumber);
        }
    }
}

```

```

// if intersection has no roads left, delete it as well
if (targetIntersections[0]->GetRoadCount() == 0)
{
    auto it = find(intersections_.begin(),
                   intersections_.end(),
                   targetIntersections[0]);
    it = intersections_.erase(it);
    delete (*it);
    number_of_intersections--;
}

// if road was connecting, check if the connected intersection needs to be
deleted as well
if (targetIntersections.size() > 1)
{
    if (targetIntersections[1]->GetRoadCount() == 0)
    {
        auto it = find(intersections_.begin(),
                       intersections_.end(),
                       targetIntersections[1]);
        int intersection_number = (*it)->GetIntersectionNumber();
        intersections_.erase(it);
        delete (*it);
        // if exists, delete a cycle that was attached to this
intersection;
        for (Cycle *c : cycles_)
        {
            if (c->GetIntersection()->GetIntersectionNumber()
                == intersection_number)
            {
                auto it2 = find(cycles_.begin(), cycles_.end(), c);
                cycles_.erase(it2);
                delete (*it);
                number_of_cycles--;
            }
        }
        number_of_intersections--;
    }
}

// set selected as nullptr
this->SelectedLane = nullptr;

// reload the map to display the changes
this->ReloadMap();

return true;
}
return false;
}

```

```

/// draw the map, and all of its belongings
void Map::Draw(RenderWindow *window) {
    // Draw all intersections
    for (Intersection *inter : intersections_)
    {
        inter->Draw(window);
    }

    // draw all routes

    for (auto &route : routes_)
    {
        route->Draw(window);
    }

    for (Cycle *c : cycles_)
    {
        c->Draw(window);
    }
}

/// return a list of all the lanes' id's
set<QString> Map::GetLaneIdList(int phaseNumber) {
    set<QString> idList = set<QString>();

    if (phaseNumber != 0)
    {
        Phase *p = GetPhase(phaseNumber);
        for (Lane *lane : *p->GetAssignedLanes())
        {
            idList.insert(QString::number(lane->GetLaneNumber()));
        }
    } else
    {
        for (Intersection *inter : intersections_)
        {
            for (Road *road : *inter->GetRoads())
            {
                for (Lane *lane : *road->GetLanes())
                {

                    idList.insert(QString::number(lane->GetLaneNumber()));
                }
            }
        }
    }

    return idList;
}

```

```

/// return a list of all the roads' id's
set<QString> Map::GetRoadIdList() {
    set<QString> idList = set<QString>();
    for (Intersection *inter : intersections_)
    {
        for (Road *road : *inter->GetRoads())
        {
            idList.insert(QString::number(road->GetRoadNumber()));
        }
    }

    return idList;
}

/// return a list of all the intersections' id's
set<QString> Map::GetIntersectionIdList() {
    set<QString> idList = set<QString>();
    for (Intersection *inter : intersections_)
    {
        idList.insert(QString::number(inter->GetIntersectionNumber()));
    }

    return idList;
}

/// return a list of all the intersections' id's
set<QString> Map::GetPhaseIdList() {
    set<QString> idList = set<QString>();
    for (Phase *p : *GetPhases())
    {
        idList.insert(QString::number(p->GetPhaseNumber()));
    }

    return idList;
}

/// return a list of all the cycles' id's
set<QString> Map::GetCycleIdList() {
    set<QString> idList = set<QString>();
    for (Cycle *c : cycles_)
    {
        idList.insert(QString::number(c->GetCycleNumber()));
    }

    return idList;
}

/// return a list of all the intersections' id's
set<QString> Map::GetLightIdList() {

```

```

set<QString> idList = set<QString>();
for (Phase *p : *GetPhases())
{
    for (Light *l : *p->GetLights())
    {
        idList.insert(QString::number(l->GetLightNumber()));
    }
    idList.insert(QString::number(p->GetPhaseNumber()));
}

return idList;
}

```

Phase.cpp

```

//  

// Created by Samuel Arbibe on 20/02/2020.  

//  

#include "Phase.hpp"  

int Phase::PhaseCount = 0;  

Phase::Phase(int phaseNumber, int cycleNumber, float cycleTime) {
    number_of_lights_ = 0;
    phase_number_ = phaseNumber;
    cycle_number_ = cycleNumber;
    cycle_time_ = cycleTime;
    open_ = false;
    open_time_ = 0;
    state_ = RED;
    priority_ = phaseNumber;
}  

Phase::~Phase() {
    for (Light *light : lights_)
    {
        delete light;
    }

    if (Settings::DrawDelete)
        cout << "Phase " << phase_number_ << " deleted." << endl;
}

/// add a light to this phase and attach it to a road

```

```

Light *Phase::AddLight(int lightNumber, Road *parentRoad) {
    if (lightNumber == 0)
    {
        lightNumber = Light::LightCount + 1;
    }

    Light *temp = new Light(lightNumber, phase_number_, parentRoad);
    lights_.push_back(temp);

    ++Light::LightCount;
    ++number_of_lights_;

    return temp;
}

/// assign a lane to this phase
void Phase::AddLane(Lane *lane) {
    lanes_.push_back(lane);
    lane->SetPhaseNumber(phase_number_);
}

/// reload all the lights in this phase
void Phase::ReloadPhase() {
    for (Light *l : lights_)
    {
        l->UpdatePosition();
    }
}

///////////
/// \brief
///
/// Unassign a given lane from this phase. delete it from the lanes_ array.
///
/// \param lane (Lane *) - a pointer to the lane to unassign
///
/// \return true if succesfull, else false
///
/////////
bool Phase::UnassignLane(Lane *lane) {

    if (lane != nullptr)
    {
        auto it = lanes_.begin();

        while (it != lanes_.end())
        {
            if ((*it)->GetLaneNumber() == lane->GetLaneNumber())
            {
                int laneNumber = lane->GetLaneNumber();

```

```

        it = lanes_.erase(it);

        if (Settings::DrawActive)
            cout << "lane " << laneNumber << " removed from phase "
            << phase_number_ << endl;
        return true;
    } else
    {
        it++;
    }
}

return false;
}

/// update
void Phase::Update(float elapsedTime) {
    if (open_)
    {
        open_time_ += elapsedTime * Settings::Speed;

        if (open_time_ > cycle_time_)
        {
            open_time_ = 0;
            open_ = false;
            state_ = RED;
        } else if (open_time_ < Settings::OrangeDelay)
        {
            state_ = ORANGE;
        } else
        {
            state_ = GREEN;
        }
    }
    for (Light *l : lights_)
    {
        l->SetState(state_);
        l->Update(elapsedTime);
    }
    for (Lane *l : lanes_)
    {
        if (state_ == GREEN)
        {
            l->SetIsBlocked(false);
        } else
        {
            l->SetIsBlocked(true);
        }
    }
}

```

```

///////////
/// \brief
///
/// Create input values for the neural network
///
/// \param inputValues (vector<double> &) - a reference to the input array
///
/////////
void Phase::GetInputValues(vector<double> &inputValues) {
    inputValues[0] = GetMaxLaneDensity();
    inputValues[1] = GetMaxQueueLength();
}
/// return the largest lane queue length
float Phase::GetMaxQueueLength() {
    float max = 0;
    float temp = 0;

    for (Lane *l : lanes_)
    {
        if ((temp = l->GetQueueLength()) > max)
        {
            max = temp;
        }
    }

    return max / 2500.f;
}
/// return the largest lane density
float Phase::GetMaxLaneDensity() {
    float max = 0;
    float temp = 0;

    for (Lane *l : lanes_)
    {
        if ((temp = l->GetNormalizedDensity()) > max)
        {
            max = temp;
        }
    }

    return max;
}
/// draw
void Phase::Draw(RenderWindow *window) {
    for (Light *l : lights_)
    {
        l->Draw(window);
    }
}

```

Road.cpp

```
//  
//  Road.cpp  
//  SimulatorSFML  
//  
//  Created by Samuel Arbibe on 03/12/2019.  
//  Copyright © 2019 Samuel Arbibe. All rights reserved.  
//  
  
#include "Road.hpp"  
  
int Road::RoadCount = 0;  
  
/// ctor for a normal road  
Road::Road(int roadNumber,  
           int intersectionNumber,  
           int connectionSide,  
           Vector2f startPosition,  
           float length,  
           float direction) {  
  
    is_connecting_ = false;  
    road_number_ = roadNumber;  
    intersection_number_[0] = intersectionNumber;  
    intersection_number_[1] = intersectionNumber;  
    connection_side_[0] = connectionSide;  
    start_pos_ = startPosition;  
    length_ = length;  
    direction_ = direction;  
    number_of_lanes_ = 0;  
    width_ = 0;  
    current_vehicle_count_ = 0;  
    total_vehicle_count_ = 0;  
  
    // calculate end position:  
    Vector2f lengthVec;  
  
    Transform t;  
    t.rotate(direction + 180);  
  
    lengthVec = t.transformPoint(Vector2f(0.f, -1.f)) * length;  
    end_pos_ = start_pos_ - lengthVec;  
  
    // init rectangle shape  
    this->setOrigin(width_ / 2, 0.f);  
    this->setPosition(start_pos_);  
    this->setFillColor(Color::Transparent);
```

```

this->setRotation(direction_ + 180);
this->setSize(Vector2f(width_, length_));

data_box_ = new DataBox(end_pos_);
data_box_->AddData("ID", road_number_);
data_box_->AddData("Count", 0);
}

/// ctor for a connecting road
Road::Road(int roadNumber,
           int intersectionNumber1,
           int intersectionNumber2,
           int connectionSide1,
           int connectionSide2,
           Vector2f conPosition1,
           Vector2f conPosition2,
           float direction) {

    is_connecting_ = true;
    road_number_ = roadNumber;
    intersection_number_[0] = intersectionNumber1;
    intersection_number_[1] = intersectionNumber2;
    connection_side_[0] = connectionSide1;
    connection_side_[1] = connectionSide2;
    start_pos_ = conPosition1;
    end_pos_ = conPosition2;
    direction_ = direction;
    number_of_lanes_ = 0;
    width_ = 0;
    length_ = Settings::CalculateDistance(start_pos_, end_pos_);

    // calculate end position:

    // init rectangle shape
    this->setOrigin(width_ / 2, 0.f);
    this->setPosition(start_pos_);
    this->setFillColor(Color::Transparent);
    this->setRotation(direction_ + 180);
    this->setSize(Vector2f(width_, length_));

    data_box_ = new DataBox(end_pos_);
    data_box_->AddData("ID", road_number_);
    data_box_->AddData("Count", 0);
}

Road::~Road() {
    for (Lane *lane : lanes_)
    {
        delete lane;
    }
}

```

```

if (Settings::DrawDelete)
    cout << "Road " << road_number_ << " deleted" << endl;

delete data_box_;
}

/// add a lane to a road
Lane *Road::AddLane(int laneNumber, bool isInRoadDirection) {
    if (!laneNumber)
    {
        laneNumber = Lane::LaneCount + 1;
    }

    if (isInRoadDirection)
    {
        lanes_.push_back(
            new Lane(laneNumber,
                      road_number_,
                      intersection_number_[isInRoadDirection],
                      start_pos_,
                      length_,
                      direction_,
                      isInRoadDirection));
    } else
    {
        lanes_.push_back(new Lane(laneNumber,
                                  road_number_,
                                  intersection_number_[isInRoadDirection],
                                  end_pos_,
                                  length_,
                                  (direction_ + 180.f),
                                  isInRoadDirection));
    }

    number_of_lanes_++;
    Lane::LaneCount++;

    // adjust road size
    width_ = number_of_lanes_ * Settings::LaneWidth;
    this->setSize(Vector2f(width_, length_));
    this->setOrigin(width_ / 2, 0.f);

    if (Settings::DrawAdded)
        std::cout << "lane " << lanes_[number_of_lanes_ - 1]->GetLaneNumber()
                  << " added to road " << road_number_
                  << std::endl;
    return lanes_[number_of_lanes_ - 1];
}

```

```

/// get lane by laneNumber
Lane *Road::GetLane(int laneNumber) {
    for (int i = 0; i < number_of_lanes_; i++)
    {
        if (lanes_[i]->GetLaneNumber() == laneNumber)
        {
            return lanes_[i];
        }
    }

    return nullptr;
}

///////////////////////////////
/// \brief
///
/// Re-assigns the lane positions of all the lanes in this road.
/// Every time a lane is added or removed, a position re-assignment
/// is needed.
///
/////////////////////////////
void Road::ReAssignLanePositions() {

    Vector2f firstLanePoint;
    Vector2f firstLaneDifference;
    Vector2f laneDifference;
    Vector2f lengthVec;

    // temporarily deactivate Deletion drawing
    bool prevState = Settings::DrawDelete;
    Settings::DrawDelete = false;

    Transform t, x;

    t.rotate(direction_ + 90);
    laneDifference = t.transformPoint(0.f, -1.f) * Settings::LaneWidth;

    (number_of_lanes_ % 2) ?
    x.scale(number_of_lanes_ / 2, number_of_lanes_ / 2) :
    x.scale((number_of_lanes_ - 1) / 2 + 0.5, (number_of_lanes_ - 1) / 2 + 0.5);

    firstLaneDifference = x.transformPoint(laneDifference);

    firstLanePoint = start_pos_ - firstLaneDifference;

    for (int i = 0; i < number_of_lanes_; i++)
    {

        Transform z, y;

```

```

z.scale(i, i);

int tempLaneNumber = lanes_[i]->GetLaneNumber();
float tempLaneDirection = lanes_[i]->GetDirection();

//if lane is in road direction
if (tempLaneDirection == direction_)
{
    // send calculated starting point
    *lanes_[i] = Lane(tempLaneNumber,
                       road_number_,
                       intersection_number_[1],
                       firstLanePoint + z.transformPoint(laneDifference),
                       Settings::CalculateDistance(start_pos_, end_pos_),
                       direction_, true);
}
else
{
    // send starting point + length vector
    y.rotate(direction_);
    lengthVec = y.transformPoint(Vector2f(0.f, -1.f)) * length_;

    *lanes_[i] = Lane(tempLaneNumber,
                       road_number_,
                       intersection_number_[0],
                       firstLanePoint + z.transformPoint(laneDifference)
                           + lengthVec,
                       Settings::CalculateDistance(start_pos_, end_pos_),
                       (direction_ + 180.f), false);
}
}

Settings::DrawDelete = prevState;

BuildLaneLines();
}

/// update the road's start position
void Road::UpdateStartPosition(Vector2f position) {
    start_pos_ = position;
    length_ = Settings::CalculateDistance(end_pos_, start_pos_);
    this->setOrigin(width_ / 2, 0.f);
    this->setPosition(start_pos_);
    this->setSize(Vector2f(width_, length_));

    ReAssignLanePositions();
}

/// update the road's end position
void Road::UpdateEndPosition(Vector2f position) {
    end_pos_ = position;
}

```

```

length_ = Settings::CalculateDistance(end_pos_, start_pos_);
this->setSize(Vector2f(width_, length_));

ReAssignLanePositions();
}

/// check if a road has been selected
Lane *Road::CheckSelection(Vector2f position) {
    // for each intersection in map
    for (Lane *lane : lanes_)
    {
        // if selection found
        if (lane->getGlobalBounds().contains(position))
        {
            return lane;
        }
    }
    return nullptr;
}

/// update, for databoxes ect.
void Road::Update(float elapsedTime) {

    current_vehicle_count_ = 0;
    total_vehicle_count_ = 0;

    for (Lane *l : lanes_)
    {
        l->Update(elapsedTime);
        current_vehicle_count_ += l->GetCurrentVehicleCount();
        total_vehicle_count_ += l->GetTotalVehicleCount();
    }

    if (Settings::DrawRoadDataBoxes)
    {
        data_box_->SetData("Count", current_vehicle_count_);
    }
}

/// delete a given lane in this road
bool Road::DeleteLane(int laneNumber) {
    Lane *targetLane = this->GetLane(laneNumber);

    // if lane was found
    if (targetLane != nullptr)
    {
        // remove the targetLane from the list by iterator
        auto it = lanes_.begin();
        while (it != lanes_.end())
        {

```

```

        if ((*it)->GetLaneNumber() == laneNumber)
    {
        it = lanes_.erase(it);
        delete targetLane;
        number_of_lanes_--;
        return true;
    } else
    {
        it++;
    }
}
return false;
}

/// create all the lines rendered on the road
void Road::BuildLaneLines() {

    if (number_of_lanes_ > 0)
    {
        // clear prev vector
        lane_lines_.clear();

        int lineCount = number_of_lanes_ + 1;

        bool firstLaneInDir = lanes_[0]->GetIsInRoadDirection();

        int directionSwitchIndex = 0;

        for (Lane *lane : lanes_)
        {
            if (lane->GetIsInRoadDirection() != firstLaneInDir)
            {
                break;
            }
            directionSwitchIndex++;
        }

        if (directionSwitchIndex == 0 || directionSwitchIndex == lineCount)
        {
            directionSwitchIndex = -1;
        }

        // first and last lines will be thick white lines;
        // the indexed line will be a double line / thick white line
        // everything else will be regular dashed lines

        // pre-made direction vectors for dashed lines:
        Transform t, d, s;
        t.rotate(this->direction_);
    }
}

```

```

d.scale(Settings::DashLineLength, Settings::DashLineLength);
s.scale(Settings::DashLineSpace, Settings::DashLineSpace);

Vector2f dashVec = t.transformPoint(Settings::BaseVec);
dashVec = d.transformPoint(dashVec);

Vector2f spacerVec = t.transformPoint(Settings::BaseVec);
spacerVec = s.transformPoint(spacerVec);

//pre-made direction vectors for lines starting points:
Transform h, v;

h.scale(this->length_, this->length_);
h.rotate(this->direction_);

v.scale(Settings::LaneWidth, Settings::LaneWidth);
v.rotate(this->direction_ - 90);

Vector2f lengthVec = h.transformPoint(Settings::BaseVec);
Vector2f laneWidthVec = v.transformPoint(Settings::BaseVec);

Vector2f upperLeftCorner =
    start_pos_ - laneWidthVec * float(number_of_lanes_) / 2.f;

Vector2f startPos, endPos;
LaneLine temp;

for (int i = 0; i < lineCount; i++)
{
    temp = LaneLine();
    startPos = upperLeftCorner + laneWidthVec * float(i);
    endPos = startPos + lengthVec;

    // border lines
    if (i == 0 || i == lineCount - 1)
    {
        VertexArray line = VertexArray(Lines, 2);

        line[0].position = startPos;
        line[1].position = endPos;

        temp.push_back(line);
    }
    // separator line
    else if (i == directionSwitchIndex)
    {
        if (Settings::DoubleSeparatorLine)
        {
            VertexArray line1 = VertexArray(Lines, 2);

```

```

        line1[0].position = startPos + laneWidthVec * 0.05f;
        line1[0].color = Color::Yellow;
        line1[1].position = endPos + laneWidthVec * 0.05f;
        line1[1].color = Color::Yellow;

        temp.push_back(line1);

        VertexArray line2 = VertexArray(Lines, 2);

        line2[0].position = startPos - laneWidthVec * 0.05f;
        line2[0].color = Color::Yellow;
        line2[1].position = endPos - laneWidthVec * 0.05f;
        line2[1].color = Color::Yellow;

        temp.push_back(line2);
    } else
    {
        VertexArray line = VertexArray(Lines, 2);

        line[0].position = startPos;
        line[0].color = Color::Yellow;
        line[1].position = endPos;
        line[1].color = Color::Yellow;

        temp.push_back(line);
    }
}
// dashed lines
else
{
    Vector2f tempPos = startPos;

    tempPos += spacerVec;

    while (Settings::CalculateDistance(startPos, tempPos + dashVec)
        < length_)
    {
        VertexArray line = VertexArray(Lines, 2);

        line[0].position = tempPos;
        tempPos += dashVec;
        line[1].position = tempPos;
        tempPos += spacerVec;

        temp.push_back(line);
    }

    if (Settings::CalculateDistance(tempPos, endPos)
        > Settings::CalculateDistance(Vector2f(0, 0), dashVec))

```

```

    {
        VertexArray line = VertexArray(Lines, 2);

        line[0].position = tempPos;
        line[1].position = endPos;

        temp.push_back(line);
    }
}
lane_lines_.push_back(temp);
}

}

/// reload the road dimensions
void Road::ReloadRoadDimensions() {
    width_ = Settings::LaneWidth * number_of_lanes_;
    length_ = Settings::CalculateDistance(start_pos_, end_pos_);
}

/// draw the road and al of its lanes
void Road::Draw(RenderWindow *window) {
    for (int i = 0; i < number_of_lanes_; i++)
    {
        lanes_[i]->Draw(window);
    }

    window->draw(*this);

    for (LaneLine laneLine : lane_lines_)
    {
        for (VertexArray va : laneLine)
        {
            window->draw(va);
        }
    }

    if (Settings::DrawRoadDataBoxes)
        data_box_->Draw(window);
}

```

Route.cpp

```
//  
// Created by Samuel Arbibe on 16/02/2020.  
  
  
#include "Route.hpp"  
  
int Route::RouteCount{0};  
  
Route::Route(Lane *from, Lane *to)  
{  
    route_number_ = ++RouteCount;  
    selected_ = false;  
  
    FromLane = from;  
    FromLane = from;  
    ToLane = to;  
  
    lines_.push_back(new Vertex[2]);  
    lines_.push_back(new Vertex[2]);  
  
    ReloadRoute();  
}  
  
Route::~Route()  
{  
    lines_.clear();  
}  
  
/// build the green lane lines  
void Route::BuildLaneLines()  
{  
  
    lines_[0][0] = Vertex(FromLane->GetStartPosition());  
    lines_[0][0].color = Color::Green;  
    lines_[0][1] = sf::Vertex(FromLane->GetEndPosition());  
    lines_[0][1].color = Color::Green;  
  
    lines_[1][0] = Vertex(ToLane->GetStartPosition());  
    lines_[1][0].color = Color::Green;  
    lines_[1][1] = sf::Vertex(ToLane->GetEndPosition());  
    lines_[1][1].color = Color::Green;  
  
}  
  
/// build the green radius lines in intersections  
void Route::BuildRadiusLine()
```

```

{
    radius_line_.clear();

    Vector2f startPos = FromLane->GetEndPosition();
    Vector2f endPos = ToLane->GetStartPosition();

    float distanceSourceTarget = Settings::CalculateDistance(startPos, endPos);
    float angle = Settings::CalculateAngle(FromLane->GetDirection(), ToLane-
>GetDirection());
    float radius = (distanceSourceTarget / 2) / (sin(angle * M_PI / 360.f));

    Transform t;

    t.rotate(FromLane->GetDirection() + 90);
    t.scale(radius, radius);

    // if straight line
    if(angle < 1 && angle > -1)
    {
        radius_line_ = VertexArray(LinesStrip, 2);
        radius_line_[0].position = startPos;
        radius_line_[0].color = Color::Green;
        radius_line_[1].position = endPos;
        radius_line_[1].color = Color::Green;
    }
    else
    {
        Vector2f radiusVec = t.transformPoint(Settings::BaseVec);
        Vector2f circleCenter = startPos + radiusVec;

        // a strip [alpha] of lines, making a quarter of a circle
        radius_line_ = VertexArray(LinesStrip, abs(angle));

        Transform cycle;
        cycle.rotate(angle / abs(angle));

        Vector2f newPos;

        for(int i = 0; i < int(abs(angle)); i++)
        {
            newPos = circleCenter - radiusVec;
            radius_line_[i].position = newPos;
            radius_line_[i].color = Color::Green;
            radiusVec = cycle.transformPoint(radiusVec);
        }
    }
}

```

```
/// reload this route
void Route::ReloadRoute()
{
    BuildLaneLines();
    BuildRadiusLine();
}

/// draw
void Route::Draw(RenderWindow *window)
{
    if(Settings::DrawRoutes || selected_)
    {
        window->draw(radius_line_);

        for (Vertex *l : lines_)
        {
            window->draw(l, 2, Lines);
        }
    }
}
```

sim / neural_network

NeuralNet.cpp

```
//  
// Created by Samuel Arbibe on 08/04/2020.  
  
  
#include "NeuralNet.hpp"  
  
Net Net::BestNet = Net();  
Net *Net::CurrentNet = nullptr;  
const unsigned Net::PopulationSize = 10;  
unsigned Net::GenerationCount = 0;  
unsigned Net::CurrentNetIndex = 0;  
float Net::HighScore = 0;  
vector<Net> Net::Generation = vector<Net>();  
  
///////////  
/// \brief Overload of binary operator !=  
///  
/// Creates a new generation of neural nets, based on the  
/// previous generation.  
///  
///////////  
void Net::NextGeneration() {  
  
    Net::CurrentNet = nullptr;  
  
    // normalize the fitness of all the nets in this gen  
    Net::NormalizeFitness(Net::Generation);  
    // create a new generation of nets  
    vector<Net> newGen = Net::Generate(Net::Generation);  
    // copy the new gen to the Generation array  
    Net::Generation = newGen;  
  
    Net::CurrentNetIndex = 0;  
    Net::GenerationCount++;  
}  
  
///////////  
/// \brief  
///  
/// Create a new array of neural nets based on an old generation  
///  
/// \param oldGen (vector<Net>) - the previous gen of NN's  
///  
/// \return new array of nets
```

```

///////////
vector<Net> Net::Generate(const vector<Net> &oldGen) {
    vector<Net> newGen;
    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        newGen.push_back(Net::PoolSelection(oldGen));
        newGen.back().mutate(0.2);
    }
    return newGen;
}

/////////
/// \brief
///
/// Normalizes the fitness of all the nets to a value
/// between 0 and 1
///
/////////
void Net::NormalizeFitness(vector<Net> &oldGen) {
    double sum = 0;
    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        double score = pow(oldGen[i].score_, 2);
        oldGen[i].score_ = score;
        sum += score;
    }

    for(unsigned i = 0; i < oldGen.size(); i++)
    {
        oldGen[i].fitness_ = oldGen[i].score_ / sum;
    }
}

/////////
/// \brief
///
/// Create a copy of a net, selected from the old net.
/// The probability of a net being selected from the pool
/// is according to its fitness.
///
/// \param oldGen (vector<Net>) - The previous generation
///
/// \return a copied Net
/////////
Net Net::PoolSelection(const vector<Net> &oldGen) {
    unsigned index = 0;
    double r = rand() / double(RAND_MAX);

    while(r > 0)
    {

```

```

        r -= oldGen[index].fitness_;
        index++;
    }
    index--;
    return oldGen[index];
}

Net::Net(const vector<unsigned> &topology) {

    size_ = Vector2f(Settings::DefaultMapWidth, Settings::DefaultMapHeight);

    unsigned layerCount = topology.size();
    for (unsigned layerNum = 0; layerNum < layerCount; ++layerNum)
    {
        layers_.push_back(Layer());

        unsigned numOutputs =
            layerNum == topology.size() - 1 ? 0 : topology[layerNum + 1];

        // We have made a new Layer, fill it ith neurons
        unsigned neuronCount = topology[layerNum];
        for (unsigned neuronNum = 0; neuronNum < neuronCount;
             ++neuronNum)
        {
            Vector2f position = calculate_neuron_position(layerNum,
                layerCount,
                neuronNum,
                neuronCount);

            layers_.back().push_back(Neuron(numOutputs,
                neuronNum,
                position,
                size_.x / 25.f));
        }
    }

    create_weight_vertex_array();
    Update(0.f);
}

///////////
/// \brief
///
/// Mutates the weights in a neural net by a given mutation rate.
///
/// \param mutationRate (float) - the mutation range
/////////
void Net::mutate(float mutationRate)
{
    // for all layer but the output layer
    for(unsigned i = 0; i < layers_.size() - 1; i++)

```

```

{
    Layer & layer = layers_[i];
    for(unsigned j = 0; j < layer.size(); j++)
    {
        layer[j].Mutate(mutationRate);
    }
}

///////////
/// \brief
///
/// Saves the neural net in a JSON file.
///
/// \param dir (string) - the directory in which the file will be saved
///
///////////

void Net::Save(const string dir) {

    json j;

    unsigned layerCount = layers_.size();

    for (unsigned layerNum = 0; layerNum < layerCount; ++layerNum)
    {
        unsigned neuronCount = layers_[layerNum].size();

        j["layers"].push_back(
        {
            {"ID", layerNum},
            {"neuron_count", neuronCount}
        });
    }

    for (unsigned neuronNum = 0; neuronNum < neuronCount; ++neuronNum)
    {
        vector<Connection>
        weights = layers_[layerNum][neuronNum].GetWeights();
        unsigned weightCount = weights.size();

        for (unsigned weightNum = 0; weightNum < weightCount; ++weightNum)
        {
            j["weights"].push_back(
            {
                {"delta_weight", weights[weightNum].deltaWeight},
                {"weight", weights[weightNum].weight}
            });
        };
    }
}

```

```

}

// write to file
ofstream o(dir);
o << setw(4) << j << endl;
o.close();

cout << "Set saved to '" << dir << "' successfully." << endl;

}

///////////
/// \brief
///
/// Loads a given JSON file and creates a new NeuralNet
/// object with it, and sets it as 'bestNet'
///
/// \param dir (string) - the directory of the JSON file
/////////
void Net::Load(const string dir) {
    try
    {
        json j;
        // open the given file, read it to a json variable
        ifstream i(dir);
        i >> j;

        vector<unsigned> topology;
        for (auto data : j["layers"])
        {
            topology.push_back(unsigned(data["neuron_count"]));
        }

        Net::BestNet = Net(topology);

        unsigned layerCount = Net::BestNet.layers_.size();

        for (unsigned layerNum = 0; layerNum < layerCount; ++layerNum)
        {
            unsigned neuronCount = Net::BestNet.layers_[layerNum].size();

            for (unsigned neuronNum = 0; neuronNum < neuronCount; ++neuronNum)
            {
                vector<Connection> weights;
                unsigned weightCount =
                    Net::BestNet.layers_[layerNum][neuronNum].GetWeights()
                    .size();

                for (unsigned weightNum = 0; weightNum < weightCount;
                     ++weightNum)

```

```

    {
        Connection con = Connection();
        con.deltaWeight = j["weights"][weightNum]["delta_weight"];
        con.weight = j["weights"][weightNum]["weight"];
        weights.push_back(con);
    }

    Net::BestNet.layers_[layerNum][neuronNum]
        .SetWeights(weights);
}
}

catch (const std::exception &e)
{
    cout << "Could not load Neural Network from this directory." << endl;
    cout << e.what() << endl;
}
}

/// creates the visual element making the visual net
void Net::create_weight_vertex_array() {

    unsigned layerCount = layers_.size();

    for (unsigned layerNum = 1; layerNum < layerCount; ++layerNum)
    {
        Layer &prevLayer = layers_[layerNum - 1];
        Layer &currLayer = layers_[layerNum];
        unsigned NeuronCount = layers_[layerNum].size();
        unsigned PrevNeuronCount = prevLayer.size();

        // previous layer
        for (unsigned i = 0; i < PrevNeuronCount; ++i)
        {
            Neuron *prevNeuron = &prevLayer[i];
            // current layer
            for (unsigned j = 0; j < NeuronCount; ++j)
            {
                Neuron *currNeuron = &currLayer[j];

                VertexArray line = VertexArray(Lines, 2);

                line[0].position = prevNeuron->GetPosition();
                line[1].position = currNeuron->GetPosition();

                weight_lines_.push_back(line);
            }
        }
    }
}

```

```

void Net::Update(float elapsedTime) {

    int weights_index = 0;
    for (int l = 0; l < layers_.size(); l++)
    {
        for (int n = 0; n < layers_[l].size(); n++)
        {
            layers_[l][n].Update(elapsedTime, &weight_lines_, &weights_index);
        }
    }
}

void Net::Draw(RenderWindow *window) {

    for (VertexArray va : weight_lines_)
    {
        window->draw(va);
    }

    for (Layer &l : layers_)
    {
        for (Neuron n : l)
        {
            n.Draw(window);
        }
    }
}

void Net::Reset() {

    for (int l = 0; l < layers_.size() - 1; l++)
    {
        Layer &layer = layers_[l];
        for (int n = 0; n < layer.size(); n++)
        {
            layer[n].Reset();
        }
    }
    Update(0.f);
}

void Net::GetResults(vector<double> &resultVals) const {
    resultVals.clear();

    for (unsigned n = 0; n < layers_.back().size(); ++n)
    {
        resultVals.push_back(layers_.back()[n].GetOutputValue());
    }
}

```

```

Vector2f Net::calculate_neuron_position(unsigned layerNum,
                                         unsigned layerCount,
                                         unsigned neuronNum,
                                         unsigned neuronCount) {

    Vector2f pos;

    pos.x = size_.x / float(layerCount + 1) * float(layerNum + 1);
    pos.y = size_.y / float(neuronCount + 1) * float(neuronNum + 1);

    return pos;
}

[[maybe_unused]] void Net::PrintNet() {

    for (unsigned l = 0; l < layers_.size(); l++)
    {
        for (unsigned n = 0; n < layers_[l].size(); n++)
        {
            cout << setprecision(6) << layers_[l][n].GetOutputValue() << " ";
        }
        cout << endl;
    }
}

void Net::FeedForward(const vector<double> &inputVals) {
    // Check the num of inputVals equal to neuronnum expect bias
    assert(inputVals.size() == layers_[0].size());

    // Assign {latch} the input values into the input neurons
    for (unsigned i = 0; i < inputVals.size(); ++i)
    {
        layers_[0][i].SetOutputValue(inputVals[i]);
    }

    // Forward propagate
    for (unsigned layerNum = 1; layerNum < layers_.size(); ++layerNum)
    {
        Layer &prevLayer = layers_[layerNum - 1];
        for (unsigned n = 0; n < layers_[layerNum].size(); ++n)
        {
            layers_[layerNum][n].FeedForward(prevLayer);
        }
    }
}

```

Neuron.cpp

```
//  
// Created by Samuel Arbibe on 08/04/2020.  
  
  
#include "Neuron.hpp"  
  
  
double mutator(double value, float rate){  
    if((rand() / double(RAND_MAX)) < rate){  
        return value + (rand() / double(RAND_MAX)) * 2 - 1;  
    }else{  
        return value;  
    }  
}  
  
Neuron::Neuron(unsigned numOutputs,  
               unsigned myIndex,  
               Vector2f position,  
               float radius) {  
  
    // create random output weights  
    for (int c = 0; c < numOutputs; c++)  
    {  
        // push a random into the output weights  
        output_weights_.push_back(Connection());  
        output_weights_.back().weight = randomize_weight();  
    }  
  
    my_index_ = myIndex;  
  
    circle_ = new CircleShape();  
    circle_->setOrigin(radius, radius);  
    circle_->setPosition(position);  
    circle_->setRadius(radius);  
    circle_->setFillColor(Color::Black);  
  
}  
  
void Neuron::Update(float elapsedTime,  
                    vector<VertexArray> *weight_lines_,  
                    int *firstWeightIndex) {  
  
    int outputCount = output_weights_.size();  
    int sum = 0;  
    int value = 0, max = 0;  
    Color col;
```

```

        for (int c = 0; c < outputCount; c++)
    {
        value = 255 * output_weights_[c].weight;
        col = Color(value, value, value);
        (*weight_lines_)[(firstWeightIndex)][0].color = col;
        (*weight_lines_)[(firstWeightIndex)++][1].color = col;

        sum += value;
    }
    if (sum > 0)
    {
        value = sum / outputCount;
    } else
    {
        value = 0;
    }
    circle_->setFillColor(Color(value, value, value));
}

void Neuron::Draw(RenderWindow *window) {
    window->draw(*circle_);
}

void Neuron::Reset() {

    for (unsigned i = 0; i < output_weights_.size(); i++)
    {
        output_weights_[i].weight = randomize_weight();
        output_weights_[i].deltaWeight = 0;
    }

    output_value_ = 0;
}

vector<Connection> Neuron::GetWeights() {
    vector<Connection> temp;

    for (unsigned w = 0; w < output_weights_.size(); w++)
    {
        temp.push_back(output_weights_[w]);
    }

    return temp;
}

void Neuron::SetWeights(vector<Connection> weights) {

    for (unsigned w = 0; w < output_weights_.size(); w++)
    {

```

```

        output_weights_[w] = weights[w];
    }
}

/// sum up all the weights that go into this neuron
// apply a transfer function
// and set it as this neuron's output value
void Neuron::FeedForward(const Layer &prevLayer) {

    // the sum of outputs from previous layer
    // into this node
    float sum = 0.f;

    // for each neuron in previous layer
    for (int n = 0; n < prevLayer.size(); ++n)
    {
        sum += prevLayer[n].GetOutputValue() *
               prevLayer[n].output_weights_[my_index_].weight;
    }

    output_value_ = Neuron::transfer_function(sum);
}

/// a simple transfer function
// which outputs in range [-1.0 .. 1.0]
double Neuron::transfer_function(double x) {

    return 1 / (1 + exp(-x));
}

void Neuron::Mutate(float mutationRate) {

    for(unsigned w = 0; w < output_weights_.size(); w++)
    {
        Connection & con = output_weights_[w];
        con.weight = mutator(con.weight, mutationRate);
    }
}

```

sim / simulator

DataBox.cpp

```
//  
// Created by Samuel Arbibe on 07/02/2020.  
  
  
#include "DataBox.hpp"  
  
Font DataBox::font_{};  
bool DataBox::font_loaded_ = false;  
  
DataBox::DataBox(Vector2f position) : RectangleShape() {  
    // will be displayed [offset] pixels above target  
    max_data_items_ = 3;  
    data_count_ = 0;  
    offset_ = Vector2f(-0.f, -0.f);  
    this->setPosition(Vector2f(position.x, position.y) + offset_);  
    this->setSize(Vector2f(150.f, 50.f));  
    this->setFillColor(Color::White);  
    this->setOutlineColor(Color::Blue);  
    this->setOutlineThickness(4.f);  
  
    // load font if needed  
    if (!font_loaded_)  
    {  
        if (!font_.loadFromFile("../resources/fonts/Roboto/Roboto-Bold.ttf"))  
        {  
            cout << "ERROR: Could not load font from the given file." << endl;  
        }  
    }  
}  
  
/// update  
void DataBox::Update(Vector2f position) {  
    this->setPosition(Vector2f(position.x, position.y) + offset_);  
}  
  
/// draw  
void DataBox::Draw(RenderWindow *window) {  
    window->draw(*this);  
    int count = 0;  
    float space = 35;  
    this->setSize(Vector2f(this->getSize().x, space * data_count_ + 5.f));
```

```

Text temp;
string s;
temp.setCharacterSize(space - 5);
temp.setFillColor(Color::Red);
temp.setFont(font_);

for (auto &data_item : data_.items())
{
    s = data_item.key() + ":" + to_string(int(data_item.value()));
    temp.setString(s);
    temp.setPosition(Vector2f(this->getPosition().x + 10.f, this-
>getPosition().y + space * count));
    //temp.setPosition(Vector2f(this->getPosition().x + 10.f, this-
>getPosition().y - 30.f));

    window->draw(temp);
    count++;
}
}

/// add data to this databox
bool DataBase::AddData(string valueName, float value) {
    if (data_.size() < max_data_items_)
    {
        data_[valueName] = value;
        data_count_++;
        return true;
    }
    cout << "Could not add another data item into data box, max items reached." <<
endl;
    return false;
}

/// set the data of an element in this databox
bool DataBase::SetData(string valueName, float value) {
    if (data_[valueName] != nullptr)
    {
        data_[valueName] = value;
        return true;
    }

    cout << "Could not set databox data item, as data name was not found." << endl;
    return false;
}

/// remove a data item
bool DataBase::RemoveData(string valueName) {
    if (data_[valueName] != nullptr)
    {
        data_.erase("valueName");
    }
}

```

```

        return true;
    }

    cout << "Could not remove databox data item, as data name was not found." <<
endl;
    return false;
}

```

Engine.cpp

```

//  

// Engine.cpp  

// SimulatorSFML  

//  

// Created by Samuel Arbibe on 22/11/2019.  

// Copyright © 2019 Samuel Arbibe. All rights reserved.  

//  

#include "Engine.hpp"

Engine::Engine(QWidget *Parent) : QSFMLCanvas(Parent,
                                              1000 / Settings::Interval,
                                              1000 / Settings::Fps) {

    cout << "Setting Up map..." << endl;
    map = new Map(0, Settings::DefaultMapWidth, Settings::DefaultMapHeight);

    cout << "Setting Up Camera..." << endl;
    snap_to_grid_ = true;
    view_pos_ = Vector2f(0, 0);
    temp_view_pos_ = Vector2f(0, 0);
    number_of_sets_ = 0;
    set_view();
    set_minimap(Vector2f(Settings::MinimapWidth, Settings::MinimapHeight),
                Settings::MinimapMargin);
    set_visual_net(Vector2f(Settings::VisualNetWidth,
                           Settings::VisualNetHeight),
                  Settings::VisualNetMargin);
    this->setView(view_);

    cout << "Setting up snap grid..." << endl;
    BuildGrid(Settings::GridRows, Settings::GridColumns);
}

/// set up the map according to the selected presets
void Engine::on_init() {

    map->AddIntersection(0, map->GetSize() / 2.f);
}

```

```

map->AddRoad(0, 1, UP, Settings::DefaultLaneLength);
map->AddRoad(0, 1, RIGHT, Settings::DefaultLaneLength);
map->AddRoad(0, 1, DOWN, Settings::DefaultLaneLength);
map->AddRoad(0, 1, LEFT, Settings::DefaultLaneLength);

map->AddLane(0, 1, false);
map->AddLane(0, 1, false);
map->AddLane(0, 1, true);
map->AddLane(0, 1, true);
map->AddLane(0, 2, false);
map->AddLane(0, 2, false);
map->AddLane(0, 2, true);
map->AddLane(0, 2, true);
map->AddLane(0, 3, false);
map->AddLane(0, 3, false);
map->AddLane(0, 3, true);
map->AddLane(0, 3, true);
map->AddLane(0, 4, false);
map->AddLane(0, 4, false);
map->AddLane(0, 4, true);
map->AddLane(0, 4, true);

map->AddRoute(1, 16);
map->AddRoute(1, 12);
map->AddRoute(2, 7);
map->AddRoute(5, 4);
map->AddRoute(5, 16);
map->AddRoute(6, 11);
map->AddRoute(9, 8);
map->AddRoute(9, 4);
map->AddRoute(10, 15);
map->AddRoute(13, 12);
map->AddRoute(13, 8);
map->AddRoute(14, 3);

map->AddCycle(0, 1);

map->AddPhase(0, 1, 20);
map->AddPhase(0, 1, 20);
map->AddPhase(0, 1, 20);
map->AddPhase(0, 1, 20);

map->AssignLaneToPhase(1, 1);
map->AssignLaneToPhase(1, 9);
map->AssignLaneToPhase(2, 2);
map->AssignLaneToPhase(2, 10);
map->AssignLaneToPhase(3, 5);
map->AssignLaneToPhase(3, 13);
map->AssignLaneToPhase(4, 6);

```

```

map->AssignLaneToPhase(4, 14);

}

/// resize the sfml window
void Engine::ResizeFrame(QSize size) {
    resize(size);
    setSize(sf::Vector2u(size.width(), size.height()));

    set_view();
    //set_minimap(Settings::MinimapSize, Settings::MinimapMargin);
    this->setView(view_);
}

/// delete a simulation
bool Engine::DeleteSimulation(int simulationNumber) {

    Simulation *s = GetSimulation(simulationNumber);
    if (s != nullptr)
    {
        Set *set = GetSet(s->GetSetNumber());

        if (set != nullptr)
        {
            return set->DeleteSimulation(simulationNumber);
        }
    }
    cout << "Could not delete simulation as it wasnt found. " << endl;
    return false;
}

/// deletes the current set. The current set is either:
// the set that is currently running
// the last set that ran
bool Engine::DeleteCurrentSet() {
    auto it = sets_.begin();
    while (it != sets_.end())
    {
        if ((*it)->GetSetNumber() == Set::CurrentSet)
        {
            Set::CurrentSet = 0;
            (*it)->StopSet();
            it = sets_.erase(it);
            number_of_sets_--;

            delete (*it);

            return true;
        } else
        {

```

```

        it++;
    }

}

return false;
}

/// re-run a simualtion by sim-number, without calculating it as a simulation
bool Engine::RunDemo(int simulationNumber) {

if (!Simulation::DemoRunning && !Set::SetRunning)
{
    Simulation *s = GetSimulation(simulationNumber);
    if (s != nullptr)
    {
        Set *set = GetSet(s->GetSetNumber());

        if (set != nullptr)
        {
            return set->DemoSimulation(simulationNumber);
        }
    }
    cout << "Could not demo simulation as it wasn't found. " << endl;
}
cout << "Cannot run demo as a set is currently running." << endl;
return false;
}

/// Trains the neural network for a set amount of generation.
// at the end of a training, it saves all the data in a simulation file.
bool Engine::RunSet(int vehicleCount, int generations) {
if (!Set::SetRunning)
{
    ClearMap();

    if (Settings::ResetNeuralNet)
    {
        cout << "Creating a new neural network..." << endl;
        if (Net::CurrentNet != nullptr)
            Net::CurrentNet->Reset();
    }

    Set *s = AddSet(0, vehicleCount, generations);

    s->RunSet();

    cout << "Set number " << s->GetSetNumber() << " has started running"
        << endl;
    return true;
} else

```

```

    {
        cout << "Cannot run set as another set is already running." << endl;
        return false;
    }
}

/// set the viewport for the camera
void Engine::set_view() {
    // view setup

    view_.reset(sf::FloatRect(temp_view_pos_.x,
                              temp_view_pos_.y,
                              Settings::DefaultMapWidth,
                              Settings::DefaultMapHeight));

    view_.zoom(Settings::Zoom);
    // update the shown area index rectangle
    update_shown_area();
}

/// build the minimap
void Engine::set_minimap(Vector2f size, float margin) {
    // minimap viewPort setup
    minimap_.reset(sf::FloatRect(0,
                                 0,
                                 Settings::DefaultMapWidth,
                                 Settings::DefaultMapHeight));

    minimap_.setViewport(sf::FloatRect(
        1 - size.x - margin,
        margin,
        size.x,
        size.y
    ));

    float outlineThickness = Settings::DefaultMapWidth / 150.f;

    // background setup
    minimap_bg_ =
        RectangleShape(Vector2f(
            Settings::DefaultMapWidth - outlineThickness * 2,
            Settings::DefaultMapHeight - outlineThickness * 2));
    minimap_bg_.setPosition(outlineThickness, outlineThickness);
    minimap_bg_.setFillColor(Color(110, 110, 110, 220));
    minimap_bg_.setOutlineColor(Color::Black);
    minimap_bg_.setOutlineThickness(outlineThickness);

    shown_area_index_ = RectangleShape(Vector2f(0, 0));
    shown_area_index_.setOutlineColor(Color(255, 0, 0, 100));
    shown_area_index_.setOutlineThickness(outlineThickness);
    shown_area_index_.setFillColor(Color::Transparent);
}

```

```

        update_shown_area();
    }

    /// build the visual net
    void Engine::set_visual_net(Vector2f size, float margin) {
        // visual_net viewPort setup

        visual_net_.reset(sf::FloatRect(0,
            0,
            Settings::DefaultMapWidth,
            Settings::DefaultMapHeight));
        visual_net_.setViewport(sf::FloatRect(
            margin,
            margin,
            size.x,
            size.y));

        /*
        float zoomFactor = Settings::DefaultMapWidth / size;
        visual_net_.zoom(zoomFactor);
        */

        float outlineThickness = Settings::DefaultMapWidth / 200.f;

        // background setup
        visual_net_bg_ =
            RectangleShape(Vector2f(
                Settings::DefaultMapWidth - outlineThickness * 2.f,
                Settings::DefaultMapHeight - outlineThickness * 2.f));
        visual_net_bg_.setPosition(outlineThickness, outlineThickness);
        visual_net_bg_.setFillColor(Color(110, 110, 110, 220));
        visual_net_bg_.setOutlineColor(Color::Black);
        visual_net_bg_.setOutlineThickness(outlineThickness);

    }

    /// update camera after movement, and enforce overflow
    void Engine::UpdateView(Vector2f posDelta, float newZoom) {
        // view_pos_ is position before dragging
        // t_view_pos_ is the current view position
        temp_view_pos_ =
            view_pos_ + posDelta * Settings::Zoom * Settings::DragFactor;

        // enforce overflow blocking
        if (!Settings::MapOverflow && map != nullptr)
        {
            if (abs(temp_view_pos_.x)
                > map->GetSize().x / 2 - shown_area_index_.GetSize().x / 2)
            {

```

```

        temp_view_pos_.x =
            (map->GetSize().x / 2 - shown_area_index_.GetSize().x / 2)
                * temp_view_pos_.x / abs(temp_view_pos_.x);
    }
    if (abs(temp_view_pos_.y)
        > map->GetSize().y / 2 - shown_area_index_.GetSize().y / 2)
    {
        temp_view_pos_.y =
            (map->GetSize().y / 2 - shown_area_index_.GetSize().y / 2)
                * temp_view_pos_.y / abs(temp_view_pos_.y);
    }
}

if (newZoom != 0)
{
    Settings::Zoom = newZoom;
}
// set view
set_view();
}

/// update the shown area index in minimap
void Engine::update_shown_area() {
    // calculate the actual position of the shown area
    Vector2f position = view_.getCenter();

    // to calculate shown area dimensions,
    // we multiply the map size by the zoom
    // view_size = map_size * zoom
    Vector2f size = view_.GetSize();

    this->shown_area_index_.setSize(size);
    this->shown_area_index_.setOrigin(size / 2.f);
    this->shown_area_index_.setPosition(position);
}

/// build the snap grid
void Engine::BuildGrid(int rows, int cols) {

    snap_grid_.Lines.clear(); // clear the old lines list
    snap_grid_.Rows = rows;
    snap_grid_.Columns = cols;
    snap_grid_.ColumnWidth = int(map->GetSize().x) / snap_grid_.Columns;
    snap_grid_.RowHeight = int(map->GetSize().y) / snap_grid_.Rows;

    // create all vertical lines of the grid
    for (int i = 1; i < snap_grid_.Columns; i++)
    {
        Vertex *line = new Vertex[2];

```

```

        line[0] = sf::Vertex(sf::Vector2f(i * snap_grid_.ColumnWidth, 0));
        line[1] = sf::Vertex(sf::Vector2f(i * snap_grid_.ColumnWidth,
                                         map->GetSize().y));

        snap_grid_.Lines.push_back(line);
    }
    // create all horizontal lines of snap grid
    for (int i = 1; i < snap_grid_.Rows; i++)
    {
        Vertex *line;
        line = new Vertex[2];

        line[0] = sf::Vertex(sf::Vector2f(0, i * snap_grid_.RowHeight));
        line[1] = sf::Vertex(sf::Vector2f(map->GetSize().x,
                                         i * snap_grid_.RowHeight));

        snap_grid_.Lines.push_back(line);
    }
}

///////////////////////////////
/// \brief
///
/// Draws a point in the clicked position on the canvas. if the snap_to_grid option
is enabled,
/// it will snap the point to the grid.
///
/// \param position (Vector2f) - the click point
///
/// \return the snapped point if option enabled, else the point
///
/////////////////////////////
Vector2f Engine::DrawPoint(Vector2f position) {
    // convert it to units according to screen pixel to display ratio
    position *= float(Settings::SFMLRatio);
    // convert it to world coordinates
    position = this->mapPixelToCoords(Vector2i(position), view_);

    // check if a lane exists in the choosen location
    check_selection(position);

    Vector2f temp;
    if (snap_to_grid_)
    {
        temp = GetSnappedPoint(position);
    } else
    {
        temp = position;
    }
}

```

```

        click_point_ = CircleShape(snap_grid_.ColumnWidth / 5);
        click_point_.setOrigin(click_point_.getRadius(), click_point_.getRadius());
        click_point_.setFillColor(Color::Red);
        click_point_.setPosition(temp.x, temp.y);

        return temp;
    }

/// get an array of all the simulations
/*
vector<Simulation *> *Engine::GetSimulations() {

    vector<Simulation *> *sims;
    sims = new vector<Simulation *>();

    for (Set *set : sets_)
    {
        for (Simulation *sim : *set->GetSimulations())
        {
            sims->push_back(sim);
        }
    }

    return sims;
}
*/

```

```

/// get simualtion by simulation number
Simulation *Engine::GetSimulation(int simulationNumber) {
    Simulation *temp = nullptr;

    for (Set *s : sets_)
    {
        if ((temp = s->GetSimulation(simulationNumber)) != nullptr)
        {
            return temp;
        }
    }

    return nullptr;
}

/// get set by set number
Set *Engine::GetSet(int setNumber) {
    for (Set *s: sets_)
    {
        if (s->GetSetNumber() == setNumber)
        {
            return s;
        }
    }
}

```

```

    }

    return nullptr;
}

/// generate a grid-snapped point with a given point
Vector2f Engine::GetSnappedPoint(Vector2f point) {
    float x = 0, y = 0;

    if (int(point.x) % snap_grid_.ColumnWidth > snap_grid_.ColumnWidth / 2)
    {
        x = ceil(point.x / snap_grid_.ColumnWidth)
            * snap_grid_.ColumnWidth;
    } else
    {
        x = floor(point.x / snap_grid_.ColumnWidth)
            * snap_grid_.ColumnWidth;
    }

    if (int(point.y) % snap_grid_.RowHeight > snap_grid_.RowHeight / 2)
    {
        y = ceil(point.y / snap_grid_.RowHeight) * snap_grid_.RowHeight;
    } else
    {
        y = floor(point.y / snap_grid_.RowHeight) * snap_grid_.RowHeight;
    }

    return Vector2f(x, y);
}

/// check if a road was selected
void Engine::check_selection(Vector2f position) {
    // unselect current selection
    map->UnselectAll();

    if (Vehicle::SelectedVehicle != nullptr)
    {
        Vehicle::SelectedVehicle->Unselect();
    }
    Vehicle::SelectedVehicle = nullptr;

    // only check for lane selection if vehicle hasn't been selected
    if (Vehicle::CheckSelection(position) == nullptr)
    {
        Lane *temp = map->CheckSelection(position);

        if (temp != nullptr)
        {

            map->SelectedLane = temp;
        }
    }
}

```

```

        map->SelectedLane->Select();
    }
} else // if vehicle has been selected, select its routes as well
{
    list<Lane *> *ptr = Vehicle::SelectedVehicle->GetInstructionSet();
    Lane *currentLane;
    if ((currentLane = Vehicle::SelectedVehicle->GetCurrentLane())
        != nullptr)
    {
        ptr->push_front(currentLane);
        map->SelectRoutesByVehicle(ptr);
        ptr->pop_front();
    }
}

/// get user input, and make changes accordingly
void Engine::input() {
    // implementation of mouse dragging
    static bool dragging = false;
    static Vector2i startPos = Vector2i(0.0f, 0.0f);

    QPoint clickPoint = this->mapFromGlobal(QCursor::pos());
    Vector2i mousePos = Vector2i(clickPoint.x(), clickPoint.y());

    if (Mouse::isButtonPressed(sf::Mouse::Left)
        && this->getViewport(view_).contains(mousePos))
    {
        if (!dragging)
            startPos = mousePos;
        dragging = true;
    } else
    {
        if (dragging)
        {
            view_pos_ = temp_view_pos_;
        }
        dragging = false;
    }

    if (dragging)
    {
        UpdateView(Vector2f(startPos.x - mousePos.x, startPos.y - mousePos.y));
    }
}

/// add a new set into the sim engine
Set *Engine::AddSet(int setNumber, int vehicleCount, int generations) {
    if (setNumber == 0)
    {

```

```

        setNumber = Set::SetCount + 1;
    }

    Set *set = new Set(setNumber, generations, vehicleCount);
    sets_.push_back(set);

    Set::SetCount++;
    number_of_sets_++;

    if (Settings::DrawAdded)
    {
        cout << "Set number " << setNumber << " added." << endl;
    }

    return set;
}

/// build a map using instructions from a given json file
void Engine::LoadMap(const string &loadDirectory) {
    // first, delete the old map.
    ResetMap();

    try
    {
        json j;
        // open the given file, read it to a json variable
        ifstream i(loadDirectory);
        i >> j;

        // build intersections
        for (auto data : j["intersections"])
        {
            map->AddIntersection(data["id"],
                                   Vector2f(data["position"][0],
                                             data["position"][1]));
        }

        // build connecting roads
        for (auto data : j["connecting_roads"])
        {
            map->AddConnectingRoad(data["id"],
                                     data["intersection_number"][0],
                                     data["intersection_number"][1]);
        }

        // build roads
        for (auto data : j["roads"])
        {
            map->AddRoad(data["id"],
                          data["intersection_number"]);
        }
    }
}

```

```

        data[ "connection_side" ],
        Settings::DefaultLaneLength);
}

for (auto data : j[ "lanes" ])
{
    map->AddLane(data[ "id" ],
                    data[ "road_number" ],
                    data[ "is_in_road_direction" ]);
}

for (auto data : j[ "routes" ])
{
    map->AddRoute(data[ "from" ], data[ "to" ]);
}

for (auto data : j[ "cycles" ])
{
    int interId = data[ "attached_intersection_id" ];
    map->AddCycle(data[ "id" ], interId);
}

for (auto data : j[ "phases" ])
{
    map->AddPhase(data[ "id" ], data[ "cycle_id" ], data[ "cycle_time" ]);
}

for (auto data : j[ "assigned_lanes" ])
{
    map->AssignLaneToPhase(data[ "phase_number" ], data[ "lane_number" ]);
}

for (auto data : j[ "lights" ])
{
    map->AddLight(data[ "id" ],
                    data[ "phase_number" ],
                    data[ "parent_road_number" ]);
}

cout << "map has been successfully loaded from '" << loadDirectory
     << " . "
     << endl;
}

catch (const std::exception &e)
{
    cout << "Could not load map from this directory." << endl;
    cout << e.what() << endl;
}
}

```

```

/// save the current map to a json file
void Engine::SaveMap(const string &saveDirectory) {
    // first save intersections, then save connecting roads, then save roads, then
    save lanes
    json j;

    for (Intersection *inter : *map->GetIntersections())
    {
        j["intersections"].push_back(
        {
            {"id", inter->GetIntersectionNumber()},
            {"position", {inter->GetPosition().x, inter->GetPosition().y}}
        });
        for (Road *road : *inter->GetRoads())
        {
            // check if road is a connecting road
            if (!road->GetIsConnecting())
            {
                j["roads"].push_back(
                {
                    {"id", road->GetRoadNumber()},
                    {"intersection_number", road->GetIntersectionNumber()},
                    {"connection_side", road->GetConnectionSide()}
                });
            }

            for (Lane *lane : *road->GetLanes())
            {
                j["lanes"].push_back(
                {
                    {"id", lane->GetLaneNumber()},
                    {"road_number", lane->GetRoadNumber()},
                    {"is_in_road_direction",
                     lane->GetIsInRoadDirection()}
                });
            }
        } else
        {
            // only save the connecting road once for the connected
            intersection
            if (inter->GetIntersectionNumber()
                == road->GetIntersectionNumber(0))
            {
                j["connecting_roads"].push_back(
                {
                    {"id", road->GetRoadNumber()},
                    {"intersection_number",
                     {road->GetIntersectionNumber(
                         0), road->GetIntersectionNumber(1)}}
                });
            }
        }
    }
}

```

```

        for (Lane *lane : *road->GetLanes())
        {
            j[ "lanes" ].push_back(
            {
                {"id", lane->GetLaneNumber()},
                {"road_number", lane->GetRoadNumber()},
                {"is_in_road_direction",
                 lane->GetIsInRoadDirection()}
            });
        }
    }

    for (Route *route : *map->GetRoutes())
    {
        j[ "routes" ].push_back(
        {
            {"from", route->FromLane->GetLaneNumber()},
            {"to", route->ToLane->GetLaneNumber()}
        });
    }

    for (Cycle *cycle : *map->GetCycles())
    {
        j[ "cycles" ].push_back(
        {
            {"id", cycle->GetCycleNumber()},
            {"attached_intersection_id",
             (cycle->GetIntersection() != nullptr) ? cycle
                ->GetIntersection()
                ->GetIntersectionNumber() : 0}
        });
        for (Phase *phase : *cycle->GetPhases())
        {
            j[ "phases" ].push_back(
            {
                {"cycle_id", phase->GetCycleNumber()},
                {"id", phase->GetPhaseNumber()},
                {"cycle_time", phase->GetCycleTime()}
            });
        }

        for (Lane *lane : *phase->GetAssignedLanes())
        {
            j[ "assigned_lanes" ].push_back(
            {

```

```

        {"phase_number", phase->GetPhaseNumber()},
        {"lane_number", lane->GetLaneNumber()}
    }
)
}

for (Light *light : *phase->GetLights())
{
    j["lights"].push_back(
    {
        {"id", light->GetLightNumber()},
        {"phase_number", light->GetPhaseNumber()},
        {"parent_road_number",
         light->GetParentRoad()->GetRoadNumber()}
    }
);
}

// write to file
ofstream o(saveDirectory);
o << setw(4) << j << endl;
o.close();

cout << "map saved to '" << saveDirectory << "' successfully." << endl;
}

/// save the neural net in a given directory in JSON file
void Engine::SaveNet(const string &saveDirectory) {
    if (Net::CurrentNet != nullptr)
        Net::BestNet.Save(saveDirectory);
}

/// load a neural network from a given JSON file
void Engine::LoadNet(const string &saveDirectory) {
    Net::Load(saveDirectory);
}

/// save the recent simulations to a file
void Engine::SaveSets(const string &saveDirectory) {

    json j;
    for (Set *set : sets_)
    {
        j["sets"].push_back(
        {
            {"id", set->GetSetNumber()},
            {"generation_simulated", set->GetGenerationsSimulated()},
            {"generation_count", set->GetGenerationsCount()},
        }
    }
}
}

```

```

        {"vehicle_count", set->GetVehicleCount()},
        {"start_time", static_cast<long int>(*set->GetStartTime())},
        {"end_time", static_cast<long int>(*set->GetEndTime())},
        {"progress", set->GetProgress()},
        {"finished", set->IsFinished()}

    }

);

for (Simulation *sim : *set->GetSimulations())
{
    j["simulations"].push_back(
    {
        {"id", sim->GetSimulationNumber()},
        {"set_id", sim->GetSetNumber()},
        {"vehicle_count", sim->GetVehicleCount()},
        {"start_time", static_cast<long int>(*sim->GetStartTime())},
        {"end_time", static_cast<long int>(*sim->GetEndTime())},
        {"simulated_time", sim->GetElapsedTime()},
        {"result", sim->GetResult()},

    }
);
}

// write to file
ofstream o(saveDirectory);
o << setw(4) << j << endl;
o.close();

cout << "Set saved to '" << saveDirectory << "' successfully." << endl;
}

/// load simulations from a file
void Engine::LoadSets(const string &loadDirectory) {

try
{
    json j;
    // open the given file, read it to a json variable
    ifstream i(loadDirectory);
    i >> j;

    Set *s;
    for (auto data : j["sets"])
    {
        s = new Set(data["id"],
                    data["generation_count"],
                    data["vehicle_count"]);
    }
}

```

```

        s->SetStartTime(time_t(data["start_time"]));
        s->SetEndTime(time_t(data["end_time"]));
        s->SetGenerationsSimulated(data["generation_simulated"]);
        s->SetProgress(data["progress"]);
        s->SetFinished(data["finished"]);
        sets_.push_back(s);
    }

    // build intersections
    Simulation *sim;
    for (auto data : j["simulations"])
    {
        s = GetSet(data["set_id"]);
        sim = s->AddSimulation(data["id"], data["vehicle_count"]);
        sim->SetStartTime(time_t(data["start_time"]));
        sim->SetEndTime(time_t(data["end_time"]));
        sim->SetSimulationTime(data["simulated_time"]);
        sim->SetFinished(true);
    }

    if (!sets_.empty())
    {
        cout << "sets have been successfully loaded from '" << loadDirectory
            << "'." << endl;
    } else
    {
        throw std::exception();
    }
}
catch (const std::exception &e)
{
    cout << "Could not load simulations from this directory." << endl;
    cout << e.what() << endl;
}
}

/// reset the whole map, delete everything
void Engine::ResetMap() {

    ClearMap();

    cout << "Resetting the Neural Network..." << endl;
    if (Net::CurrentNet != nullptr)
        Net::CurrentNet->Reset();

    cout << "Resetting map..." << endl;
    delete map;
    map = new Map(0, Settings::DefaultMapWidth, Settings::DefaultMapWidth);
}

```

```

        cout << "===== map has been reset ====="
        << endl;
    }

/// stop the current simulation, and clear all vehicles
void Engine::ClearMap() {

    // clear all lanes;
    for (Lane *l : *map->GetLanes())
    {
        l->ClearLane();
    }

    for (Set *s : sets_)
    {
        s->StopSet();
    }

    cout << "Deleting Vehicles..." << endl;
    Vehicle::DeleteAllVehicles();

    cout << "===== map has been cleared ====="
        << endl;
}

/// do the game cycle (input->update)
/// draw and display are separate for different fps
/// this allows running logic cycle in high rate -> better accuracy
// and running draw cycle in low rate -> better performance
void Engine::logic_cycle() {

    input();
    update((float(logic_timer_.interval()) / 1000.f));
}

/// do the rest of the game cycle independently
void Engine::draw_cycle() {
    render();
    display();
}

/// update all the engine's objects
void Engine::update(float elapsedTime) {

    map->Update(elapsedTime);

    // deploy vehicles if needed
    if (Vehicle::VehiclesToDeploy > 0)
    {
        add_vehicles_with_delay(elapsedTime);
    }
}

```

```

}

for (Vehicle *v : Vehicle::ActiveVehicles)
{
    v->Update(elapsedTime);
}

//clear all cars to be deleted
Vehicle::ClearVehicles();

if (Settings::DrawFps)
    cout << "FPS : " << 1000.f / elapsedTime << endl;

// follow the selected car
if (Settings::FollowSelectedVehicle && Vehicle::SelectedVehicle != nullptr)
{
    view_pos_ = Vehicle::SelectedVehicle->getPosition()
        - Vector2f(map->GetSize().x / 2, map->GetSize().y / 2);
    temp_view_pos_ = view_pos_;
    set_view();
}

for (Set *s : sets_)
{
    // when an update on a set returns true
    // it means that a simulation has finished
    if (s->Update(elapsedTime))
    {
        if (s->IsFinished())
        {
            SetFinished();
        } else
        {
            SimulationFinished();

            // set the new score as result
            float result = s->GetLastSimulationResult();

            if (Settings::RunBestNet)
            {
                Net::BestNet.Update(elapsedTime);
            } else
            {

                Net::CurrentNet->SetScore(result);
                Net::CurrentNet->Update(elapsedTime);

                if (result > Net::HighScore)
                {

```



```

int randomIndex = 0;

if (Settings::MultiTypeVehicle)
    randomIndex = random() % 4;

return (Vehicle::AddVehicle(track,
                            this->map,
                            static_cast<VehicleTypeOptions>(randomIndex))
        != nullptr);
} else
{
    cout << "Could not add a new vehicle as tracks could not be generated."
        << endl;
    return false;
}
}

/// render the engine's objects
void Engine::render() {
    // Clean out the last frame
    clear(BackgroundColor);

    // Draw the map
    this->map->Draw(this);

    // Draw all vehicles
    for (Vehicle *v : Vehicle::ActiveVehicles)
    {
        // only draw active vehicles; stacked vehicles wont be rendered
        if (v->GetIsActive())
        {
            v->Draw(this);
        }
    }

    // Draw the click index
    if (Settings::DrawClickPoint)
        this->draw(this->click_point_);

    // Draw the grid
    if (Settings::DrawGrid)
    {
        for (Vertex *l : this->snap_grid_.Lines)
        {
            this->draw(l, 2, Lines);
        }
    }

    // draw the minimap
    if (Settings::DrawMinimap)

```

```

{
    this->setView(minimap_); // switch to minimap for rendering
    render_minimap(); // render minimap
}

// draw the visual net
if (Settings::DrawVisualNet)
{
    this->setView(visual_net_); // switch to minimap for rendering
    render_visual_net(); // render minimap
}

this->setView(view_); // switch back to main view
}

/// drawing the minimap is drawing everything but the vehicles and the grid, on a
smaller scale
void Engine::render_minimap() {
    // Draw the minimap's background
    this->draw(minimap_bg_);

    // Draw the map
    this->map->Draw(this);

    // Draw the click index
    if (Settings::DrawClickPoint)
        this->draw(this->click_point_);

    // Draw the shown area index
    this->draw(shown_area_index_);
}

/// render the visual net
void Engine::render_visual_net() {

    this->draw(visual_net_bg_);

    if (Settings::RunBestNet)
    {
        Net::BestNet.Draw(this);
    } else if (Net::CurrentNet != nullptr)
    {
        Net::CurrentNet->Draw(this);
    }
}

```

Set.cpp

```
//  
// Created by Samuel Arbibe on 06/04/2020.  
  
  
#include "Set.hpp"  
  
int Set::SetCount = 0;  
int Set::CurrentSet = 0;  
bool Set::SetRunning = false;  
  
Set::Set(int setNumber, int generationCount, int vehicleCount) {  
  
    set_number_ = setNumber;  
    generations_count_ = generationCount;  
    generations_simulated_ = 0;  
    vehicle_count_ = vehicleCount;  
    running_simulation_ = nullptr;  
    running_demo_ = nullptr;  
    number_of_simulations_ = 0;  
    last_simulation_result_ = 0;  
    progress_ = 0;  
    running_ = false;  
    finished_ = false;  
  
    start_time_ = 0;  
    end_time_ = 0;  
}  
  
Set::~Set() {  
  
    for(Simulation *s : simulations_)  
    {  
        delete s;  
    }  
    simulations_.clear();  
  
    if (Settings::DrawDelete)  
        cout << "Set number " << set_number_  
            << " has been deleted. " << endl;  
}  
  
/// get the simulations of this set  
Simulation *Set::GetSimulation(int simulationNumber) {  
  
    for (Simulation *s : simulations_)  
    {
```

```

        if (s->GetSimulationNumber() == simulationNumber)
        {
            return s;
        }
    }
    return nullptr;
}

/// update
bool Set::Update(float elapsedTime) {

    if (running_)
    {
        //update set progress
        progress_ = float(generations_simulated_) / float(generations_count_);

        // if set is not finished
        if (generations_simulated_ < generations_count_)
        {
            // is there a simulation running ?
            if (running_simulation_ != nullptr)
            {
                running_simulation_->Update(elapsedTime);

                // check if simulation has finished
                if (running_simulation_->IsFinished())
                {
                    last_simulation_result_ = running_simulation_->GetResult();

                    running_simulation_ = nullptr;
                    if(!Settings::RunBestNet)
                        ++generations_simulated_;
                    // re-update set progress
                    progress_ = float(generations_simulated_) /
float(generations_count_);
                    return true;
                }
            } else
            {
                // start a new simulation and run it
                running_simulation_ = StartNewSimulation();
            }
        } else
        {
            running_ = false;
            finished_ = true;
            SetRunning = false;
            end_time_ = time(nullptr);
            return true;
        }
    }
}

```

```

    }

    if (running_demo_ != nullptr)
    {
        if (running_demo_->Update(elapsedTime))
        {
            running_demo_ = nullptr;
            SetRunning = false;
            return true;
        }
    }

    return false;
}

/// stop this set
void Set::StopSet() {

    running_ = false;
    SetRunning = false;

    if (running_simulation_ != nullptr)
    {
        running_simulation_->StopSimulation();
        DeleteSimulation(running_simulation_->GetSimulationNumber());
        running_simulation_ = nullptr;
    }

    if (running_demo_ != nullptr)
    {
        running_demo_->StopDemo();
        running_demo_ = nullptr;
    }
}

/// delete a given simulation by id
bool Set::DeleteSimulation(int simulationNumber) {
    auto it = simulations_.begin();
    while (it != simulations_.end())
    {
        if ((*it)->GetSimulationNumber() == simulationNumber)
        {
            it = simulations_.erase(it);
            if ((*it)->IsFinished())
                generations_simulated_--;
            number_of_simulations_--;
            generations_count_--;

            delete (*it);
        }
    }
}

```

```

        return true;
    } else
    {
        it++;
    }
}

cout << "Could not delete simulation as it wasn't found. " << endl;
return false;
}

/// start a new simulation, and run it
Simulation *Set::StartNewSimulation() {

    // start a new simulation
    Simulation *s;
    s = new Simulation(0, set_number_, vehicle_count_);

    simulations_.push_back(s);
    number_of_simulations_++;

    s->Run();

    return s;
}

/// add a simulation to this set (without running)
Simulation *Set::AddSimulation(int simulationNumber, int vehicleCount) {
    if (simulationNumber == 0)
    {
        simulationNumber = Simulation::SimulationCount + 1;
    }

    Simulation *s;
    s = new Simulation(simulationNumber, set_number_, vehicleCount);
    simulations_.push_back(s);

    number_of_simulations_++;
    Simulation::SimulationCount++;

    if (Settings::DrawAdded)
    {
        cout << "Simulation " << s->GetSimulationNumber() << " added." << endl;
    }

    return s;
}

/// demo a given simulation
bool Set::DemoSimulation(int simulationNumber) {

```

```

if (!SetRunning)
{
    Simulation *s = GetSimulation(simulationNumber);

    if (s != nullptr)
    {
        running_demo_ = s;
        s->Demo();
        return true;
    }
}
cout << "Cannot run demo as set is currently running." << endl;
return false;
}

/// stop a simulation demo
bool Set::StopDemo() {
    if (running_demo_ != nullptr)
    {
        running_demo_->StopDemo();
        running_demo_ = nullptr;
        return true;
    }
    return false;
}

```

Settings.cpp

```
//  
// Created by Samuel Arbibe on 21/01/2020.  
  
  
#include "Settings.hpp"  
  
const Vector2f Settings::BaseVec = Vector2f(0.f, -1.f);  
  
bool Settings::DrawFps = false;  
bool Settings::DrawActive = false;  
bool Settings::DrawDelete = false;  
bool Settings::DrawAdded = false;  
  
bool Settings::DrawVehicleDataBoxes = false;  
bool Settings::DrawRoadDataBoxes = false;  
bool Settings::DrawLightDataBoxes = false;  
bool Settings::DrawRoutes = false;  
bool Settings::DrawGrid = false;  
bool Settings::DrawLaneBlock = false;  
bool Settings::DrawTextures = true;  
bool Settings::DrawClickPoint = true;  
bool Settings::DrawMinimap = false;  
bool Settings::DrawVisualNet = false;  
bool Settings::FollowSelectedVehicle = true;  
bool Settings::LaneDensityColorRamping = false;  
bool Settings::ShowSelectedPhaseLanes = false;  
bool Settings::PrintSimulationLog = false;  
bool Settings::DrawNnProgression = true;  
bool Settings::DrawCurrentSetOnly = false;  
bool Settings::RunBestNet = false;  
  
int Settings::Interval = 1000; // max is 1000  
int Settings::Fps = 30;  
int Settings::AntiAliasing = 0;  
bool Settings::MultiColor = true;  
bool Settings::MultiTypeVehicle = true;  
float Settings::MinDistanceFromNextCar = 55;  
float Settings::MinDistanceFromStop = 50;  
bool Settings::AccWhileTurning = true;  
  
float Settings::MinLaneWidth = 83;  
float Settings::LaneWidth = 115; // lane width in px.  
float Settings::MaxLaneWidth = 140;  
float Settings::DashLineLength = 100;  
float Settings::DashLineSpace = 80;  
float Settings::Scale = 3; // 1 px / [scale] = 1 cm
```

```

float Settings::Speed = 1; // running speed
bool Settings::DoubleSeparatorLine = true;
bool Settings::ResetNeuralNet = false;
float Settings::VehicleSpawnRate = 0.9f;
float Settings::MaxDensity = 0.20f;

float Settings::DefaultLaneLength = 2300; // lane length in px

float Settings::DefaultMapWidth = 10000;
float Settings::DefaultMapHeight = 10000;

int Settings::GridColumns = 50;
int Settings::GridRows = 50;

int Settings::SFMLRatio = 1;

bool Settings::MapOverflow = false;

// camera setting
float Settings::Zoom = 0.1f;
float Settings::DragFactor = 5.f;

// minimap Settings
float Settings::MinimapWidth = 0.2f;
float Settings::MinimapHeight = 0.2f;
float Settings::MinimapMargin = 0.01f;

// visual net settings

float Settings::VisualNetWidth = 0.3f;
float Settings::VisualNetHeight = 0.3f;
float Settings::VisualNetMargin = 0.01f;

// [LaneWidth in px] * Scale * unitScale = laneWidth in Unit
// M, CM, Feet, Inch
float Settings::DistanceUnitScales[5]
{
    0.01, 1, 0.0328, 0.3937, 1 / Scale
};

// [Speed in px/s] * Scale * unitScale = Speed in Unit
// that means that [speed in cm/s] * unitScale = Speed in Unit
// CM/S, KM/H, M/S, MPH, PX/S
float Settings::VelocityUnitScales[5]
{
    1, 0.036, 0.01, 0.022, 1 / Scale
};

// [LaneWidth in px] * Scale * unitScale = laneWidth in Unit
float Settings::GetLaneWidthAs(DistanceUnits unit) {

```

```

// base
float len = Settings::LaneWidth * Settings::Scale;

return len * DistanceUnitScales[int(unit)];
}

// convert a given value from a unit to another unit
float Settings::ConvertSize(DistanceUnits fromUnit,
                             DistanceUnits toUnit,
                             float value) {
    // first, convert value to px
    // LaneWidth in px = laneWidth in unit / Scale / unitScale
    float valueInPx = value / Scale / DistanceUnitScales[int(fromUnit)];

    // convert it to the target unit
    return valueInPx * Scale * DistanceUnitScales[int(toUnit)];
}

// convert a velocity from a given unit to a given unit
float Settings::ConvertVelocity(VelocityUnits fromUnit,
                                 VelocityUnits toUnit,
                                 float value) {
    // first, convert value to px/s
    float valueInPx = value / Scale / VelocityUnitScales[int(fromUnit)];

    // convert it to the target unit
    return valueInPx * Scale * VelocityUnitScales[int(toUnit)];
}

// max speed for all the cars in px/s
float Settings::MaxSpeeds[4]
{
    ConvertVelocity(KMH, PXS, 50.f),
    ConvertVelocity(KMH, PXS, 50.f),
    ConvertVelocity(KMH, PXS, 50.f),
    ConvertVelocity(KMH, PXS, 50.f)
};

// convert m/ss to px/ss
float Settings::Acceleration[4]
{
    ConvertVelocity(MS, PXS, 2.f),
    ConvertVelocity(MS, PXS, 2.f),
    ConvertVelocity(MS, PXS, 2.f),
    ConvertVelocity(MS, PXS, 2.f)
};

float Settings::Deceleration[4]
{
    ConvertVelocity(MS, PXS, -4.5f),
    ConvertVelocity(MS, PXS, -4.5f),
    ConvertVelocity(MS, PXS, -4.5f),
}

```

```

        ConvertVelocity(MS, PXS, -4.5f)
    };

float Settings::GetMaxSpeedAs(VehicleTypeOptions vehicleType,
                               VelocityUnits unit) {
    return Settings::ConvertVelocity(PXS,
                                      unit,
                                      Settings::MaxSpeeds[vehicleType]);
}

/// calculate distance between 2 vectors
float Settings::CalculateDistance(Vector2f a, Vector2f b) {
    float xDist = abs(a.x - b.x);
    float yDist = abs(a.y - b.y);

    return sqrt(xDist * xDist + yDist * yDist);
}

float Settings::CalculateAngle(float a, float b) {

    if (a == 0)
        a += 360;
    if (b == 0)
        b += 360;

    float temp = -(a - b);

    if (temp < -180)
    {
        temp += 360;
    }

    if (temp > 180)
    {
        temp -= 360;
    }

    return temp;
}

/// convert a normalized value into a corresponding value
void Settings::GetHeatMapColor(float value,
                               float *red,
                               float *green,
                               float *blue) {
    int aR = 0;
    int aG = 0;
    int aB = 255; // RGB for our 1st color blue
    int bR = 255;
    int bG = 0;
}

```

```
int bB = 0;      // RGB for our 2nd color red

*red = (float) (bR - aR) * value + aR;      // 255 * value.
*green = (float) (bG - aG) * value + aG;      // 0.
*blue = (float) (bB - aB) * value + aB;      // 255 * (1 - value).
}

float Settings::OrangeDelay = 3.f;
float Settings::DefaultCycleTime = 20.f;
float Settings::MaxCycleTime = 60.f;
float Settings::MinCycleTime = 5.f;
float Settings::PhaseDelay = 1.5f;
```

Simulation.cpp

```
//  
// Created by Samuel Arbibe on 03/03/2020.  
  
  
#include "Simulation.hpp"  
  
int Simulation::SimulationCount = 0;  
bool Simulation::SimRunning = false;  
bool Simulation::DemoRunning = false;  
  
Simulation::Simulation(int simulationNumber, int setNumber, int vehicleCount) {  
  
    simulation_number_ =  
        (simulationNumber != 0) ? simulationNumber : SimulationCount + 1;  
    vehicle_count_ = vehicleCount;  
  
    current_vehicle_count_ = 0;  
    finished_ = false;  
    running_ = false;  
    set_number_ = setNumber;  
    result_ = 0;  
  
    // start timer  
    start_time_ = 0;  
    end_time_ = 0;  
  
    elapsed_time_ = 0;  
    ++SimulationCount;  
}  
  
Simulation::~Simulation() {  
    if (Settings::DrawDelete)  
        cout << "Simulation number " << simulation_number_  
            << " has been deleted. " << endl;  
}  
  
/// update  
/// returns true if simulation has ended  
bool Simulation::Update(float elapsedTime) {  
  
    if (running_)  
    {  
        elapsed_time_ += elapsedTime * Settings::Speed;  
  
        current_vehicle_count_ = Vehicle::GetActiveVehicleCount();  
    }  
}
```

```

    if (current_vehicle_count_ == 0 && Vehicle::VehiclesToDeploy == 0)
    {
        running_ = false;
        finished_ = true;
        Simulation::SimRunning = false;
        Simulation::DemoRunning = false;

        // get simulation end time
        end_time_ = time(nullptr);

        result_ = float(vehicle_count_) / elapsed_time_;

        if (Settings::PrintSimulationLog)
        {
            PrintSimulationLog();
        }

        return true;
    }
}

return false;
}

/// print a sim log
void Simulation::PrintSimulationLog() {

    cout << "-----"
        << endl;
    cout << "Set " << set_number_ << endl;
    cout << "Simulation " << simulation_number_ << " ended at ";
    cout << ctime(&this->end_time_) << endl;
    cout << "Simulation Type :";
    cout << " Vehicle Count" << endl;
    cout << "Results:" << endl;
    cout << "    Vehicles Simulated: " << vehicle_count_ << endl;
    cout << "    Simulation Time: " << elapsed_time_ << " seconds" << endl;
    cout << "    Score: " << result_ << endl;
    cout << "-----"
        << endl;
}

```

Vehicle.cpp

```
//  
// MovableObject.cpp  
// SimulatorSFML  
//  
// Created by Samuel Arbibe on 22/11/2019.  
// Copyright © 2019 Samuel Arbibe. All rights reserved.  
//  
  
#include "Vehicle.hpp"  
  
int Vehicle::to_be_deleted_ = 0;  
int Vehicle::ActiveVehiclesCount = 0;  
int Vehicle::VehicleCount = 0;  
int Vehicle::VehiclesToDeploy = 0;  
list<Vehicle *> Vehicle::ActiveVehicles;  
Vehicle *Vehicle::SelectedVehicle = nullptr;  
  
VehicleType Vehicle::SmallCar{  
    SMALL_CAR,  
    "SmallCar",  
    "../../resources/cars/small/small_",  
    5,  
    Vector2f(1.6 * 100 / Settings::Scale, 3 * 100 / Settings::Scale)  
};  
VehicleType Vehicle::MediumCar{  
    MEDIUM_CAR,  
    "MediumCar",  
    "../../resources/cars/medium/medium_",  
    4,  
    Vector2f(1.8 * 100 / Settings::Scale, 4 * 100 / Settings::Scale)  
};  
VehicleType Vehicle::LongCar{  
    LONG_CAR,  
    "LongCar",  
    "../../resources/cars/large/large_",  
    2,  
    Vector2f(2.4 * 100 / Settings::Scale, 5 * 100 / Settings::Scale)  
};  
  
Vehicle::Vehicle(VehicleTypeOptions vehicleType,  
                 int vehicleNumber,  
                 list<Lane *> *instructionSet,  
                 Map *map) {  
    // set initial values for the movable object  
    vehicle_type_ = GetVehicleTypeByOption(vehicleType);  
    vehicle_number_ = vehicleNumber;
```

```

max_speed_ = Settings::MaxSpeeds[vehicle_type_->Type];
acceleration = Settings::Acceleration[vehicle_type_->Type];
deceleration = Settings::Deceleration[vehicle_type_->Type];
speed_ = 0;
acc_ = 0;
state_ = DRIVE;
curr_map_ = map;
instruction_set_ = instructionSet;
source_lane_ = instruction_set_->front();
instruction_set_->pop_front();
dest_lane_ = instruction_set_->front();
active_ = false;

// get a pointer to the current intersection
// current intersection is the intersection that the lane leads to
curr_intersection_ =
    map->GetIntersection(source_lane_->GetIntersectionNumber());
// the previous intersection, or the intersection of the source lane
prev_intersection_ = nullptr;

angular_vel_ = 0;
turning_ = false;
vehicle_in_front_ = nullptr;

this->setSize(vehicle_type_->Size);
this->setRotation(source_lane_->GetDirection());
this->setPosition(source_lane_->GetStartPosition());
this->setOutlineColor(Color::Blue);
this->setOrigin(this->getSize().x / 2, this->getSize().y / 2);

// if vehicle texture hasn't been loaded yet, load it
if (Settings::DrawTextures && Vehicle::LoadVehicleTextures(vehicle_type_))
{
    // set up sprite
    int textureNumber;
    if (Settings::MultiColor)
    {
        textureNumber = (vehicle_number_ % vehicle_type_->ImageCount);
    } else
    {
        textureNumber = 1;
    }

    this->setTexture(&(vehicle_type_->Textures->at(textureNumber)));
} else
{
    this->setOutlineThickness(10.f);
    this->setOutlineColor(Color::Blue);
    this->setFillColor(Color::White);
}

```

```

}

    data_box_ = new DataBox(this->getPosition());
    data_box_->AddData("Speed", speed_);
    data_box_->AddData("ID", vehicle_number_);
}

Vehicle::~Vehicle() {
    if (Vehicle::SelectedVehicle == this)
    {
        Vehicle::SelectedVehicle = nullptr;
    }
    if (Settings::DrawDelete)
        cout << "Vehicle " << vehicle_number_ << " deleted" << endl;
}

/// delete all active vehicles
void Vehicle::DeleteAllVehicles() {
    for (Vehicle *v : Vehicle::ActiveVehicles)
    {
        v->state_ = DELETE;
        to_be_deleted_++;
    }
    VehicleCount = 0;
    VehiclesToDeploy = 0;
    ClearVehicles();
}

/// clear the 'to be deleted' vehicles
void Vehicle::ClearVehicles() {

    auto it = ActiveVehicles.begin();

    // while there are cars to delete;
    while (to_be_deleted_ != 0 && it != ActiveVehicles.end())
    {
        // if is to be deleted
        if ((*it)->GetState() == DELETE)
        {
            Vehicle *temp = (*it);
            it = ActiveVehicles.erase(it);

            delete temp;

            to_be_deleted_--;
            ActiveVehiclesCount = ActiveVehicles.size();
            if (Settings::DrawActive)
                cout << "active vehicles : " << ActiveVehiclesCount << endl;
        } else
        {

```

```

        it++;
    }
}

/// add a vehicle with an instruction set
Vehicle *Vehicle::AddVehicle(list<Lane *> *instructionSet,
                           Map *map,
                           VehicleTypeOptions vehicleType,
                           int vehicleNumber) {
    auto *temp = new Vehicle(vehicleType, vehicleNumber, instructionSet, map);
    ActiveVehicles.push_back(temp);

    temp->vehicle_in_front_ =
        (temp->source_lane_->GetBackVehicleId()) ? GetVehicle(temp->source_lane_-
->GetBackVehicleId())
                                                : nullptr;

    //set this car as the last car that entered the lane
    temp->source_lane_->PushVehicleInLane(vehicleNumber);
    ActiveVehiclesCount++;
    VehicleCount++;

    if (Settings::DrawAdded)
        cout << "car " << vehicleNumber << " added to lane "
            << temp->source_lane_->GetLaneNumber() << endl;
}

return temp;
}

/// check if the click point is on a vehicle
Vehicle *Vehicle::CheckSelection(Vector2f position) {

    for (Vehicle *v : Vehicle::ActiveVehicles)
    {
        // only check for active vehicles
        if (v->active_)
        {
            if (v->getGlobalBounds().contains(position))
            {
                Vehicle::SelectedVehicle = v;
                Vehicle::SelectedVehicle->Select();
                return v;
            }
        }
    }
    return nullptr;
}

```

```

/// select a vehicle
void Vehicle::Select() {
    this->setOutlineColor(Color::Red);
    this->setFillColor(Color::Red);
}

/// unselect a vehicle
void Vehicle::Unselect() {
    this->setOutlineColor(Color::Blue);

    this->setFillColor(Color::White);
}

/// load textures as required
bool Vehicle::LoadVehicleTextures(VehicleType *vehicleType) {

    if (vehicleType->Textures == nullptr)
    {
        vehicleType->Textures = new vector<Texture>();

        string directory;
        Texture tempTexture;
        for (int i = 1; i <= vehicleType->ImageCount; ++i)
        {
            directory = vehicleType->ImageDir + to_string(i) + ".png";

            if (tempTexture.loadFromFile(directory))
            {
                tempTexture.setSmooth(true);
                vehicleType->Textures->push_back(tempTexture);
            } else
            {
                cerr << "loading texture no." << i << " for "
                    << vehicleType->VehicleTypeName << " failed" << endl;
            }
        }

        cout
            << "-----"
            << endl;
        cout << vehicleType->Textures->size() << "/" << vehicleType->ImageCount
            << " Textures successfully added for "
            << vehicleType->VehicleTypeName
            << endl;
        cout
            << "-----"
            << endl;
        vehicleType->ImageCount = vehicleType->Textures->size();
    }
    return bool(vehicleType->ImageCount > 0);
}

```

```

}

/// convert vehicleTypeOption to VehicleType struct
VehicleType *Vehicle::GetVehicleTypeByOption(VehicleTypeOptions vehicleTypeOptions)
{
    switch (vehicleTypeOptions)
    {
        case MEDIUM_CAR:return &(Vehicle::MediumCar);
        case LONG_CAR:return &(Vehicle::LongCar);
        case SMALL_CAR:return &(Vehicle::SmallCar);
        default:return &(Vehicle::MediumCar);
    }
}

/// get vehicle by vehicleNumber
Vehicle *Vehicle::GetVehicle(int vehicleNumber) {
    for (Vehicle *v : ActiveVehicles)
    {
        if (v->vehicle_number_ == vehicleNumber)
        {
            return v;
        }
    }

    return nullptr;
}

/// transfer a vehicle from a lane to another lane
void Vehicle::transfer_vehicle(Lane *toLane) {

    this->source_lane_ = toLane;
    this->setRotation(this->source_lane_->GetDirection());
    this->angular_vel_ = 0;
    this->setPosition(this->source_lane_->GetStartPosition());
    this->curr_intersection_ =
        this->curr_map_
            ->GetIntersection(this->source_lane_->GetIntersectionNumber());
    this->vehicle_in_front_ =
        (this->source_lane_->GetBackVehicleId()) ? GetVehicle(this->source_lane_-
->GetBackVehicleId())
                                                : nullptr;
    this->source_lane_->PushVehicleInLane(this->vehicle_number_);

    this->instruction_set_->pop_front();
    // if there are instructions left, transfer them to this
    if (!this->instruction_set_->empty())
    {
        this->dest_lane_ = this->instruction_set_->front();
    } else
}

```

```

    {
        this->dest_lane_ = nullptr;
    }
}

/// do drive cycle
State Vehicle::drive() {
    // upon creation, all cars are stacked on each other.
    // while cars dont have a min distance, they wont start driving

    // check for distance with car in front
    if (vehicle_in_front_ != nullptr && vehicle_in_front_->state_ != DELETE
        && dest_lane_ != nullptr)
    {
        float distanceFromNextCar =
            Settings::CalculateDistance(this->getPosition(),
                                         vehicle_in_front_
                                         ->getPosition())
            - this->getSize().y / 2 - vehicle_in_front_->getSize().y / 2;
        float brakingDistance = -(speed_ * speed_) / (2 * deceleration);

        if (distanceFromNextCar
            < brakingDistance + Settings::MinDistanceFromNextCar ||
            distanceFromNextCar < Settings::MinDistanceFromNextCar)
        {
            turning_ = false;
            state_ = STOP;
            acc_ = deceleration;

            // if the lane is blocked, send the stopline-distance
            // to the lane and try to set the queue length
            if (source_lane_ != nullptr && source_lane_->GetIsBlocked()
                && speed_ == 0
                && active_)
            {
                float distanceFromStop =
                    Settings::CalculateDistance(this->getPosition(),
                                                source_lane_
                                                ->GetEndPosition());
                // if this is the last car with STOP state in lane,
                // the queue length is the distance from this vehicle
                // to the end of the lane;
                source_lane_->SetQueueLength(distanceFromStop);
            }

            return STOP;
        }
    }

    // check if car is in between lanes (inside an intersection) and turning
}

```

```

if (curr_intersection_->getGlobalBounds().contains(this->getPosition()) &&
    source_lane_ != nullptr &&
    dest_lane_ != nullptr)
{
    if (!turning_)
    {
        float distanceSourceTarget =
            Settings::CalculateDistance(source_lane_->GetEndPosition(),
                                         dest_lane_->GetStartPosition());
        float angle = Settings::CalculateAngle(source_lane_->GetDirection(),
                                                dest_lane_->GetDirection());

        // if going in a straight line
        if (angle < 1.f && angle > -1.f)
        {
            angular_vel_ = 0;
        } else
        {
            float turningRadius =
                (distanceSourceTarget / 2.f) / (sin(angle * M_PI / 360.f));
            float parameter = 2.f * M_PI * turningRadius;
            float turningParameter = (angle / 360.f) * parameter;

            angular_vel_ = angle / turningParameter;
        }

        turning_ = true;
    }

    if (source_lane_ != nullptr)
    {
        prev_intersection_ =
            curr_map_-
                ->GetIntersection(source_lane_->GetIntersectionNumber());
        source_lane_->PopVehicleFromLane();
        source_lane_ = nullptr;
    }

    state_ = TURN;
    //set rotation
    acc_ = (Settings::AccWhileTurning) ? acceleration / 2.f : 0;
    return TURN;
}

// check distance from stop (if lane is blocked)
if (source_lane_ != nullptr && source_lane_ != dest_lane_-
    && source_lane_->GetIsBlocked() &&
    !this->getGlobalBounds().contains(source_lane_->GetEndPosition()))
{
    float

```

```

distanceFromStop = Settings::CalculateDistance(this->getPosition(),
                                               source_lane_
                                               ->GetEndPosition())
- this->getSize().y / 2;
float brakingDistance = -(speed_ * speed_) / (2 * deceleration);

if (distanceFromStop < brakingDistance + Settings::MinDistanceFromStop)
{
    // if the lane is blocked, send the stopline-distance
    // to the lane and try to set the queue length
    if (speed_ == 0 && active_)
    {
        // if this is the last car with STOP state in lane,
        // the queue length is the distance from this vehicle
        // to the end of the lane;
        source_lane_->SetQueueLength(distanceFromStop);
    }

    // ignore the vehicle in front
    vehicle_in_front_ = nullptr;
    turning_ = false;
    state_ = STOP;
    acc_ = deceleration;
    return STOP;
}

// check if car has left intersection and is now in targetLane
if (dest_lane_ != nullptr
    && dest_lane_->getGlobalBounds().contains(this->getPosition()))
{
    // set previous intersection to nullptr
    prev_intersection_ = nullptr;

    // we need to transfer vehicle to target lane
    transfer_vehicle(dest_lane_);

    turning_ = false;
    acc_ = acceleration;
    state_ = DRIVE;
    return DRIVE;
}

// check if car is no longer in intersection
if (dest_lane_ == nullptr
    && !source_lane_->getGlobalBounds().contains(this->getPosition()))
{
    source_lane_->PopVehicleFromLane();

    turning_ = false;
}

```

```

        ++to_be_deleted_;
        state_ = DELETE;
        return DELETE;
    }

    // default = just drive
    active_ = true;
    turning_ = false;
    acc_ = acceleration;
    state_ = DRIVE;
    return DRIVE;
}

/// update a vehicle's location
void Vehicle::Update(float elapsedTime) {

    if (Settings::DrawVehicleDataBoxes)
    {
        data_box_->Update(this->getPosition());
        data_box_
            ->SetData("Speed", Settings::ConvertVelocity(PXS, KMH, speed_));
    }

    if (state_ != DELETE)
    {
        drive();

        // activate car
        if (!active_ && state_ == DRIVE)
            active_ = true;

        apply_changes(elapsedTime);
    }
}

/// apply the calculated next position
void Vehicle::apply_changes(float elapsed_time) {
    // apply acceleration
    speed_ += acc_ * elapsed_time * Settings::Speed;
    //float running_speed_ = speed_ * Settings::Speed;

    // apply max Speed limit
    if (speed_ > max_speed_)
        speed_ = max_speed_;

    // apply min Speed limit
    if (speed_ < 0)
        speed_ = 0;

    // set rotation relative to currentSpeed, to create a constant turning radius
}

```

```

Transform t;
this->rotate(angular_vel_ * speed_ * elapsed_time * Settings::Speed);

t.rotate(this->getRotation());

// rotate the movement vector in relation to the forward vector (0,1)
movement_vec_ = t.transformPoint(Settings::BaseVec);

// apply movement vector on position, relative to elapsed time to ensure
// a constant Speed at any FPS

this->move(movement_vec_ * speed_ * elapsed_time * Settings::Speed);
}

/// render the vehicle
void Vehicle::Draw(RenderWindow *window) {
    window->draw(*this);

    if (Settings::DrawVehicleDataBoxes)
        data_box_->Draw(window);
}

```

ui / widgets

QSFMLCanvas.cpp

```
//  
// Created by Samuel Arbibe on 01/01/2020.  
  
  
#include "....../sim/simulator/Settings.hpp"  
#include "QsfmlCanvas.hpp"  
  
#ifdef Q_WS_X11  
#include <Qt/qx11info_x11.h>  
#include <X11/Xlib.h>  
#endif  
  
QSFMLCanvas::QSFMLCanvas(QWidget *Parent,  
                           int intervalTime,  
                           int drawIntervalTime) :  
    QWidget(Parent),  
    RenderWindow(VideoMode(0, 0),  
                "Sim",  
                Style::Default,  
                ContextSettings(0, 0, Settings::AntiAliasing)) {  
    // Setup some states to allow direct rendering into the widget  
    setAttribute(Qt::WA_PaintOnScreen);  
    setAttribute(Qt::WA_OpaquePaintEvent);  
    setAttribute(Qt::WA_NoSystemBackground);  
  
    // Set strong focus to enable keyboard events to be received  
    setFocusPolicy(Qt::StrongFocus);  
  
    // Setup the widget geometry  
    // setup for high DPI devices, adjust sfml widget size to actual size  
  
    parent_ = Parent;  
  
    Settings::SFMLRatio =  
        parent_->devicePixelRatio(); // save ratio for later use  
    resize(parent_->size());  
  
    // Setup the logic timer  
    logic_timer_.setTimerType(Qt::PreciseTimer);  
    logic_timer_.setInterval(intervalTime);  
  
    // Setup the draw timer  
    draw_timer_.setInterval(drawIntervalTime);
```

```

        is_init_ = false;
    }

void QSFMLCanvas::showEvent(QShowEvent *) {
    if (!is_init_)
    {
        // Under X11, we need to flush the commands sent to the server to ensure
        // that SFML will get an updated view of the windows
#ifdef Q_WS_X11
        XFlush(QX11Info::display());
#endif

        // Create the SFML window with the widget handle
        RenderWindow::create(sf::::WindowHandle) winId());

        // Let the derived class do its specific stuff
        on_init();

        // Setup the timer to trigger a refresh at specified framerate
        connect(&logic_timer_, SIGNAL(timeout()), this, SLOT(repaint()));
        logic_timer_.start();

        // Setup the draw timer to trigger a redraw at specified framerate
        //connect(&draw_timer_, SIGNAL(timeout()), this, SLOT(redraw()));
        connect(&draw_timer_, SIGNAL(timeout()), this, SLOT(redraw()));
        draw_timer_.start();

        is_init_ = true;
    }
}

QPaintEngine *QSFMLCanvas::paintEngine() const {
    return nullptr;
}

void QSFMLCanvas::paintEvent(QPaintEvent *) {
    // Let the derived class do its specific stuff
    logic_cycle();
}

void QSFMLCanvas::redraw() {
    draw_cycle();
}

```

SimModel.cpp

```
//  
// Created by Samuel Arbibe on 07/03/2020.  
  
  
#include <src/sim/simulator/Set.hpp>  
#include "SimModel.hpp"  
  
SimModel::SimModel(QObject *parent)  
    : QAbstractTableModel(parent) {  
    row_count_ = 0;  
}  
  
void SimModel::populateData(const vector<Set *> *data) {  
    table_.clear();  
    row_count_ = 0;  
  
    for(unsigned i = 0; i < data->size(); i++)  
    {  
        Set * set = (*data)[i];  
        if(!Settings::DrawCurrentSetOnly || (Settings::DrawCurrentSetOnly && set->GetSetNumber() == Set::CurrentSet))  
        {  
            vector<Simulation * > * sims;  
            sims = set->GetSimulations();  
            for (unsigned j = 0; j < sims->size(); j++)  
            {  
                Simulation * s = (*sims)[j];  
                table_.append({QString::number(s->GetSetNumber()),  
                               QString::number(s->GetSimulationNumber()),  
                               QString::fromUtf8(ctime(s->GetStartTime())),  
                               QString::fromUtf8(ctime(s->GetEndTime())),  
                               QString::number(s->GetElapsedTime()),  
                               QString::number(s->GetVehicleCount()),  
                               QString::number(s->GetResult()),  
                               });  
                this->row_count_++;  
            }  
        }  
    }  
}  
  
int SimModel::GetIdByRow(int rowNumber) {  
  
    if (table_.size() >= (rowNumber + 1))  
    {  
        return table_[rowNumber][1].toInt();  
    }
```

```

    }
    return 0;
}

int SimModel::rowCount(const QModelIndex & /*parent*/) const {
    return row_count_;
}

int SimModel::columnCount(const QModelIndex & /*parent*/) const {
    return 7;
}

QVariant SimModel::data(const QModelIndex &index, int role) const {
    if (role == Qt::DisplayRole)
    {
        return table_.at(index.row()).at(index.column());
    }
    return QVariant();
}

QVariant SimModel::headerData(int section,
                             Qt::Orientation orientation,
                             int role) const {
    if (role == Qt::DisplayRole && orientation == Qt::Horizontal)
    {
        switch (section)
        {
        case 0: return QString("Set");
        case 1: return QString("ID");
        case 2: return QString("Start Time");
        case 3: return QString("End Time");
        case 4: return QString("Simulated Time");
        case 5: return QString("Vehicle Count");
        case 6: return QString("Score");
        default: return QString("#");
        }
    }
    return QVariant();
}

```

ui

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow) {
    ui->setupUi(this);

    sf::ContextSettings settings;

    SimulatorEngine = new Engine(ui->SimulatorFrame);

    DistanceUnits currentDistanceUnit =
        static_cast<DistanceUnits>(ui->DistanceUnitComboBox->currentIndex());
    VelocityUnits currentUnit =
        static_cast<VelocityUnits>(ui->VelocityUnitComboBox->currentIndex());

    // load presets
    ui->LaneWidthSlider->setMinimum(int(Settings::MinLaneWidth));
    ui->LaneWidthSlider->setMaximum(int(Settings::MaxLaneWidth));
    ui->LaneWidthSlider->setSliderPosition(int(Settings::LaneWidth));
    ui->VehicleSpawnRateSlider->setMinimum(0);
    ui->VehicleSpawnRateSlider->setMaximum(2000);
    ui->VehicleSpawnRateTextBox
        ->setText(QString::number(Settings::VehicleSpawnRate * 1000.f));
    ui->VehicleSpawnRateSlider
        ->setSliderPosition(int(Settings::VehicleSpawnRate * 1000.f));
    ui->ZoomSlider->setSliderPosition(int(Settings::Zoom * 99));
    ui->LaneWidthValueEdit
        ->setText(QString::number(Settings::GetLaneWidthAs(currentDistanceUnit)));
    ui->CarMaxSpeed->setText(QString::number(Settings::GetMaxSpeedAs(
        VehicleTypeOptions::SMALL_CAR,
        currentUnit)));
    ui->MotorcycleMaxSpeed->setText(
        QString::number(Settings::GetMaxSpeedAs(VehicleTypeOptions::LONG_CAR,
                                                currentUnit)));
    ui->TruckMaxSpeed->setText(QString::number(Settings::GetMaxSpeedAs(
        VehicleTypeOptions::MEDIUM_CAR,
        currentUnit)));

    ui->PhaseDelayLineEdit->setText(QString::number(Settings::PhaseDelay));
    ui->PhaseDelaySlider->setSliderPosition(Settings::PhaseDelay);
    ui->PhaseTimeSlider->setMaximum(int(Settings::MaxCycleTime));
    ui->PhaseTimeSlider->setMinimum(int(Settings::MinCycleTime));
```

```

ui->TrainingProgressBar->setHidden(true);

ui->Graph->setContentsMargins(0, 0, 0, 0);
ui->Graph->addGraph();
ui->Graph->xAxis->setLabel("simulation");
ui->Graph->yAxis->setLabel("score");

model_ = new SimModel(this);
selected_row_ = 0;

// connect the simulation finished event to its slot here
QObject::connect(SimulatorEngine,
                  SIGNAL(SimulationFinished()),
                  this,
                  SLOT(on_SimulationFinished()));
QObject::connect(SimulatorEngine,
                  SIGNAL(SetFinished()),
                  this,
                  SLOT(on_SetFinished()));
}

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

void MainWindow::on_SimulationFinished() {
    reload_sim_table();
    resize_sim_table();

    reload_sim_graph();

    Set * currentSet = SimulatorEngine->GetSet(Set::CurrentSet);
    if(currentSet != nullptr)
    {
        ui->TrainingProgressBar->setValue(int(currentSet->GetProgress() * 100.f));
    }
}

void MainWindow::on_SetFinished() {
    ui->AbortButton->setEnabled(false);
    ui->TrainingProgressBar->setHidden(true);
}

void MainWindow::reload_sim_table() {
    delete model_;
    model_ = new SimModel(this);
    model_->populateData(SimulatorEngine->GetSets());
    ui->SimTable->setModel(model_);
    ui->SimTable->scrollToBottom();
}

```

```

void MainWindow::reload_sim_graph() {
    Set * currentSet = SimulatorEngine->GetSet(Set::CurrentSet);
    if(currentSet != nullptr)
    {
        OVector<double> x, y;
        vector<Simulation *> * data;
        data = currentSet->GetSimulations();
        int count = 0;
        float max = 0;

        for(unsigned i = 0; i < data->size(); i++)
        {
            Simulation * sim = (*data)[i];
            x.append(sim->GetSimulationNumber());
            y.append(sim->GetResult());

            if(sim->GetResult() > max){
                max = sim->GetResult();
            }
            count++;
        }
        ui->Graph->graph(0)->setData(x, y);
        ui->Graph->rescaleAxes();
        ui->Graph->replot();
    }
}

void MainWindow::showEvent(QShowEvent *ev) {
    QMainWindow::showEvent(ev);

    // things that can be done only after complete init
    SimulatorEngine
        ->ResizeFrame(ui->SimulatorFrame->size() * Settings::SFMLRatio);

    reloadOptionData();
}

void MainWindow::reloadOptionData() {
    // set intersection number range for future use

    ui->FromIntersectionComboBox->clear();
    ui->ToIntersectionComboBox->clear();
    ui->IntersectionComboBox->clear();
    ui->IntersectionNumberComboBox->clear();
    ui->NearRoadComboBox->clear();
    ui->ToPhaseComboBox->clear();
    ui->ShowLanesForPhaseComboBox->clear();
    ui->ToCycleComboBox->clear();
    ui->PhaseTimeComboBox->clear();
}

```

```

ui->FromLaneComboBox->clear();
ui->ToLaneComboBox->clear();
ui->AssignLaneToPhaseComboBox->clear();
ui->ToRoadComboBox->clear();

for (const QString s : SimulatorEngine->map->GetIntersectionIdList())
{
    ui->FromIntersectionComboBox->addItem(s);
    ui->ToIntersectionComboBox->addItem(s);
    ui->IntersectionComboBox->addItem(s);
    ui->IntersectionNumberComboBox->addItem(s);
}

for (const QString sd : SimulatorEngine->map->GetRoadIdList())
{
    ui->ToRoadComboBox->addItem(sd);
    ui->NearRoadComboBox->addItem(sd);
}

for (const QString sd : SimulatorEngine->map->GetLaneIdList())
{
    ui->FromLaneComboBox->addItem(sd);
    ui->ToLaneComboBox->addItem(sd);
}

for (const QString p : SimulatorEngine->map->GetPhaseIdList())
{
    ui->ShowLanesForPhaseComboBox->addItem(p);
    ui->PhaseTimeComboBox->addItem(p);
    ui->ToPhaseComboBox->addItem(p);
    ui->AssignLaneToPhaseComboBox->addItem(p);
}

for (const QString p : SimulatorEngine->map->GetCycleIdList())
{
    ui->ToCycleComboBox->addItem(p);
}

reload_lane_options();
}

void MainWindow::reload_lane_options() {

    ui->AssignedLanesListView->clear();

    int phaseNumber = ui->ShowLanesForPhaseComboBox->currentText().toInt();
    if (phaseNumber != 0)
    {
        for (auto &p : SimulatorEngine->map->GetLaneIdList(phaseNumber))
        {

```

```

        QString s;
        s.append("Lane ");
        s.append(p);
        ui->AssignedLanesListView->addItem(s);
    }
}

// When mouse is clicked, use click coordinates in map setup
void MainWindow::mouseDoubleClickEvent(QMouseEvent *event) {
    if (event->buttons() == Qt::LeftButton)
    {

        QPoint clickPoint;
        Vector2f point;

        clickPoint = SimulatorEngine->mapFromGlobal(QCursor::pos());

        point.x = clickPoint.x();
        point.y = clickPoint.y();

        if (point.x > 0 && point.y > 0)
        {
            // draw a point on simulator canvas to indicate last clicked position

            point = SimulatorEngine->DrawPoint(point);

            ui->IntersectionXEdit->setText(QString::number(int(point.x)));
            ui->IntersectionYEdit->setText(QString::number(int(point.y)));

            ui->statusbar->showMessage(
                tr("You can now Click 'Add Intersection' to add an intersection at
the clicked position "));

            // check for lane selection
            Lane *selectedLane = SimulatorEngine->map->SelectedLane;
            bool isLaneSelected = (selectedLane != nullptr);
            ui->DeleteButton->setEnabled(isLaneSelected);
            if (isLaneSelected)
            {
                QString selectionText = "Selected: Lane {";
                selectionText
                    .append(QString::number(selectedLane->GetLaneNumber()));
                selectionText.append("}, Road {");
                selectionText
                    .append(QString::number(selectedLane->GetRoadNumber()));
                selectionText.append("}, Intersection {");
                selectionText.append(QString::number(selectedLane
->GetIntersectionNumber())));
            }
        }
    }
}

```

```

        selectionText.append("}", Direction {});
        selectionText
            .append(QString::number(selectedLane->GetDirection()));
        selectionText.append("}");

        ui->statusbar->showMessage(selectionText);

    }

    // check for car selection
    Vehicle *selectedVehicle = Vehicle::SelectedVehicle;
    bool isVehicleSelected = (selectedVehicle != nullptr);
    if (isVehicleSelected)
    {
        QString selectionText = "Selected: Vehicle ";
        selectionText.append(QString::number(selectedVehicle
            ->GetVehicleNumber()));

        selectionText.append("}");
        /*
        selectionText.append("Status : {");
        selectionText.append(QString::number(selectedVehicle->GetState()));
        selectionText.append("}");
        */
    }

    ui->statusbar->showMessage(selectionText);
}

}

void MainWindow::on_AddIntersectionButton_clicked()
{
    if (ui->IntersectionXEdit->text().length() > 0
        && ui->IntersectionYEdit->text().length() > 0)
    {
        int x = ui->IntersectionXEdit->text().toInt();
        int y = ui->IntersectionYEdit->text().toInt();

        // check for usable data
        if (x > 0 && x < SimulatorEngine->map->GetSize().x && y > 0
            && y < SimulatorEngine->map->GetSize().y)
        {
            Vector2f position(x, y);
            if (SimulatorEngine->map->AddIntersection(0, position) != nullptr)
            {
                ui->statusbar->clearMessage();
                ui->statusbar
                    ->showMessage(tr("Intersection Successfully added."), 5000);
                // set data for future use
            }
        }
    }
}

```

```

        reloadOptionData();
        return; // success
    }
}
ui->statusbar->showMessage(tr(
    "Could not add Intersection. please check that the entered values are
correct. "));
}

void MainWindow::on_AddConnectingRoadButton_clicked() {
    int intersection1 = ui->FromIntersectionComboBox->currentText().toInt();
    int intersection2 = ui->ToIntersectionComboBox->currentText().toInt();

    // check for usable data
    if (intersection1 != intersection2)
    {
        if (SimulatorEngine->map
            ->AddConnectingRoad(0, intersection1, intersection2) != nullptr)
        {
            ui->statusbar->clearMessage();
            ui->statusbar
                ->showMessage(tr("Connecting Road Successfully added."), 5000);
            // refresh spinboxes data
            reloadOptionData();
            return; // success
        }
    }

    ui->statusbar->showMessage(tr(
        "Could not add Connecting Road. please check that the entered values are
correct. "));
}

void MainWindow::on_AddRoadButton_clicked() {
    int intersectionNumber = ui->IntersectionComboBox->currentText().toInt();
    int connectionSide = ui->ConSideComboBox->currentIndex() + 1;

    if (SimulatorEngine->map->AddRoad(0,
                                         intersectionNumber,
                                         connectionSide,
                                         Settings::DefaultLaneLength))
    {
        // refresh spin-boxes data
        reloadOptionData();

        ui->statusbar->clearMessage();
        ui->statusbar->showMessage(tr("Road Successfully added."), 5000);
        reloadOptionData();
        return; // success
    }
}

```

```

    }

    ui->statusbar->showMessage(tr(
        "Could not add Road. please check that the entered values are correct. "));
}

void MainWindow::on_AddLanePushButton_clicked() {
    int roadNumber = ui->ToRoadComboBox->currentText().toInt();
    bool isInRoadDirection = ui->InRoadDirectionCheckBox->isChecked();

    if (SimulatorEngine->map->AddLane(0, roadNumber, isInRoadDirection))
    {
        ui->statusbar->clearMessage();
        ui->statusbar->showMessage(tr("Lane Successfully added."), 5000);
        reloadOptionData();
        return; // success
    }
    ui->statusbar->showMessage(tr(
        "Could not add Lane. please check that the entered values are correct. "));
}

void MainWindow::on_SnapToGridCheckBox_stateChanged(int arg1) {
    bool isChecked = ui->SnapToGridCheckBox->isChecked();

    ui->ShowGridCheckBox->setChecked(isChecked);
    ui->ShowGridCheckBox->setEnabled(isChecked);
    SimulatorEngine->SetSnapToGrid(isChecked);
}

void MainWindow::on_ShowCurrentSetOnlyCheckBox_stateChanged(int arg1) {
    Settings::DrawCurrentSetOnly = arg1;
    reload_sim_table();
}

void MainWindow::on_ShowGridCheckBox_stateChanged(int arg1) {
    Settings::DrawGrid = arg1;
}

void MainWindow::on_LaneWidthSlider_sliderMoved(int position) {
    DistanceUnits unit =
        static_cast<DistanceUnits>(ui->DistanceUnitComboBox->currentIndex());
    // setting the value will cut the value to the current range
    ui->LaneWidthSlider->setValue(position);
    Settings::LaneWidth = ui->LaneWidthSlider->value();
    ui->LaneWidthValueEdit
        ->setText(QString::number(Settings::GetLaneWidthAs(unit)));
    // check if init was called, if was then reload map
    if (SimulatorEngine->map != nullptr)
        SimulatorEngine->map->ReloadMap();
}

```

```

void MainWindow::on_DistanceUnitComboBox_currentIndexChanged(int index) {
    // get slider position -> lane width as px
    ui->LaneWidthValueEdit

    ->setText(QString::number(Settings::GetLaneWidthAs(static_cast<DistanceUnits>(index
))));}
}

void MainWindow::on_LaneWidthValueEdit_editingFinished() {
    // get the entered value in the current unit
    float enteredValue = ui->LaneWidthValueEdit->text().toFloat();

    // get the current unit
    DistanceUnits currentUnit =
        static_cast<DistanceUnits>(int(ui->DistanceUnitComboBox
            ->currentIndex()));

    // convert the entered unit to PX
    enteredValue =
        Settings::ConvertSize(currentUnit, DistanceUnits::PX, enteredValue);
    // send it to this function that simulate a slider movement
    on_LaneWidthSlider_sliderMoved(enteredValue);
}

void MainWindow::on_ZoomSlider_valueChanged(int value) {
    float zoomValue = 1.f - value / 100.f;
    SimulatorEngine->UpdateView(Vector2f(0, 0), zoomValue);
}

void MainWindow::on_CarMaxSpeed_editingFinished() {
    // get the entered value for the max speed
    float enteredValue = ui->CarMaxSpeed->text().toFloat();

    // get the current unit
    VelocityUnits currentUnit =
        static_cast<VelocityUnits>(int(ui->VelocityUnitComboBox
            ->currentIndex()));

    // convert the entered value to px
    enteredValue = Settings::ConvertVelocity(currentUnit,
                                                VelocityUnits::PXS,
                                                enteredValue);

    // save the changes
    Settings::MaxSpeeds[VehicleTypeOptions::SMALL_CAR] = enteredValue;
}

void MainWindow::on_MotorcycleMaxSpeed_editingFinished() {
    // get the entered value for the max speed

```

```

float enteredValue = ui->CarMaxSpeed->text().toFloat();

// get the current unit
VelocityUnits currentUnit =
    static_cast<VelocityUnits>(int(ui->VelocityUnitComboBox
        ->currentIndex()));

// convert the entered value to px
enteredValue = Settings::ConvertVelocity(currentUnit,
                                            VelocityUnits::PXS,
                                            enteredValue);

// save the changes
Settings::MaxSpeeds[VehicleTypeOptions::LONG_CAR] = enteredValue;
}

void MainWindow::on_TruckMaxSpeed_editingFinished() {
    // get the entered value for the max speed
    float enteredValue = ui->CarMaxSpeed->text().toFloat();

    // get the current unit
    VelocityUnits currentUnit =
        static_cast<VelocityUnits>(int(ui->VelocityUnitComboBox
            ->currentIndex()));

    // convert the entered value to px
    enteredValue = Settings::ConvertVelocity(currentUnit,
                                                VelocityUnits::PXS,
                                                enteredValue);

    // save the changes
    Settings::MaxSpeeds[VehicleTypeOptions::MEDIUM_CAR] = enteredValue;
}

void MainWindow::on_VelocityUnitComboBox_currentIndexChanged(int index) {
    VelocityUnits currentUnit = static_cast<VelocityUnits>(index);
    // re-display all the velocities
    ui->CarMaxSpeed->setText(QString::number(Settings::GetMaxSpeedAs(
        VehicleTypeOptions::SMALL_CAR,
        currentUnit)));
    ui->MotorcycleMaxSpeed->setText(
        QString::number(Settings::GetMaxSpeedAs(VehicleTypeOptions::LONG_CAR,
                                                currentUnit)));
    ui->TruckMaxSpeed->setText(QString::number(Settings::GetMaxSpeedAs(
        VehicleTypeOptions::MEDIUM_CAR,
        currentUnit)));
}

void MainWindow::on_MultiColorCheckBox_stateChanged(int arg1) {
    Settings::MultiColor = ui->MultiColorCheckBox->isChecked();
}

```

```

}

void MainWindow::on_DeleteButton_clicked() {
    Lane *selectedLane = SimulatorEngine->map->SelectedLane;
    if (selectedLane != nullptr)
    {
        int laneNumber = selectedLane->GetLaneNumber();
        // if deletion was successful
        if (SimulatorEngine->map->DeleteLane(selectedLane->GetLaneNumber()))
        {

            QString text = "Lane ";
            text.append(QString::number(laneNumber));
            text.append(" has been deleted. ");
            ui->statusbar->showMessage(text);

            // set spinbox ranges
            reloadOptionData();
        }
    }
}

void MainWindow::on_ResetButton_clicked() {
    QMessageBox msgBox;
    msgBox.setText("Are you sure you want to reset map?");
    msgBox.setInformativeText(
        "The map and all the active vehicles will be deleted.");
    msgBox.setStandardButtons(QMessageBox::Ok | QMessageBox::Cancel);
    msgBox.setDefaultButton(QMessageBox::Ok);
    int ret = msgBox.exec();

    switch (ret)
    {
    case QMessageBox::Ok: SimulatorEngine->ResetMap();
        ui->statusbar->showMessage(tr("map has been reset."));
        reloadOptionData();
        break;
    case QMessageBox::Cancel:break;
    }
}

void MainWindow::on_LoadMapButton_clicked() {
    QFileDialog dialog(this);
    dialog.setFileMode(QFileDialog::ExistingFile);
    dialog.setNameFilter(tr("JSON Files (*.json)"));
    dialog.setViewMode(QFileDialog::Detail);

    QStringList fileNames;
    if (dialog.exec())
    {

```

```

        fileNames = dialog.selectedFiles();
        SimulatorEngine->LoadMap(fileNames.front().toString());
        reloadOptionData();
    }
}

void MainWindow::on_LoadNNButton_clicked() {
    QFileDialog dialog(this);
    dialog.setFileMode(QFileDialog::ExistingFile);
    dialog.setNameFilter(tr("JSON Files (*.json)"));
    dialog.setViewMode(QFileDialog::Detail);

    QStringList fileNames;
    if (dialog.exec())
    {
        fileNames = dialog.selectedFiles();
        SimulatorEngine->LoadNet(fileNames.front().toString());
        reloadOptionData();
    }
}

void MainWindow::on_SaveMapButton_clicked() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"),
                                                    "map.json",
                                                    tr("JSON Files (*.json)"));
    SimulatorEngine->SaveMap(fileName.toString());
}

void MainWindow::on_SaveNNButton_clicked() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"),
                                                    "map.json",
                                                    tr("JSON Files (*.json)"));
    SimulatorEngine->SaveNet(fileName.toString());
}

void MainWindow::on_ShowDataBoxesCheckBox_stateChanged(int arg1) {
    Settings::DrawRoadDataBoxes = arg1;
    Settings::DrawLightDataBoxes = arg1;
    Settings::DrawVehicleDataBoxes = arg1;
}

void MainWindow::on_FasterButton_clicked() {
    Settings::Speed *= 2;
    QString text = "Running speed: x";
    text.append(QString::number(Settings::Speed));
    ui->RunningSpeedLabel->setText(text);
}

void MainWindow::on_SlowerButton_clicked() {
    Settings::Speed /= 2.f;
}

```

```

    QString text = "Running speed: x";
    text.append(QString::number(Settings::Speed));
    ui->RunningSpeedLabel->setText(text);
}

void MainWindow::on_PauseButton_clicked() {
    static float prev_speed = 1.f;
    if (Settings::Speed != 0.f)
    {
        prev_speed = Settings::Speed;
        Settings::Speed = 0.f;
        ui->PauseButton->setText(tr(">"));
        ui->RunningSpeedLabel->setText(tr("Paused."));
    } else
    {
        Settings::Speed = prev_speed;
        ui->PauseButton->setText(tr("||"));
        QString text = "Running speed: x";
        text.append(QString::number(Settings::Speed));
        ui->RunningSpeedLabel->setText(text);
    }
}

void MainWindow::on_RunSetButton_clicked() {

    int vehicleCount = ui->CarCountSpinBox->value();
    int generations = ui->SimulationCountSpinBox->value();

    if (!Simulation::SimRunning)
    {
        SimulatorEngine->RunSet(vehicleCount, generations);
        ui->AbortButton->setEnabled(true);
        ui->TrainingProgressBar->setValue(0);
        ui->TrainingProgressBar->setHidden(false);
    } else
    {
        ui->statusbar->showMessage(tr(
            "Another simulation is currently running, please wait for it to
finish"),
            5000);
    }
}

void MainWindow::on_AddRouteButton_clicked() {
    int lane1 = ui->FromLaneComboBox->currentText().toInt();
    int lane2 = ui->ToLaneComboBox->currentText().toInt();

    if (lane1 != 0 && lane2 != 0)
    {
        if (SimulatorEngine->map->AddRoute(lane1, lane2))

```

```

    {
        ui->statusbar->showMessage(tr("Route Added Successfully."));
        reloadOptionData();
        return;
    }
}
ui->statusbar->showMessage(tr("Could not add Route"));
}

void MainWindow::on_ReloadButton_clicked() {
    reloadOptionData();
}

void MainWindow::on_ShowRoutesCheckBox_stateChanged(int arg1) {
    Settings::DrawRoutes = arg1;
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // resize simulation frame
    SimulatorEngine
        ->ResizeFrame(ui->SimulatorFrame->size() * Settings::SFMLRatio);

    resize_sim_table();
}

void MainWindow::resize_sim_table() {
    // resize simulation table
    if (ui->SimTable->model() != nullptr)
    {
        int colCount = ui->SimTable->model()->columnCount();
        int colWidth = (ui->SimTable->size().width() - 20) / colCount;

        for (int i = 0; i < colCount; i++)
        {
            ui->SimTable->setColumnWidth(i, colWidth);
        }
    }
}

void MainWindow::on_ShowLanesForPhaseComboBox_currentTextChanged(const QString
&arg1) {
    reload_lane_options();
    int phaseNumber = ui->ShowLanesForPhaseComboBox->currentText().toInt();

    if (phaseNumber != 0 && Settings::ShowSelectedPhaseLanes)
    {
        SimulatorEngine->map->SelectLanesByPhase(phaseNumber);
    }
}

```

```

void MainWindow::on_AssignedLanesListView_itemClicked(QListWidgetItem *item) {
    //int selectedLane = item->text().toInt();
}

void MainWindow::on_AddPhaseButton_clicked() {

    int cycleNumber = ui->ToCycleComboBox->currentText().toInt();

    SimulatorEngine->map->AddPhase(0, cycleNumber, Settings::DefaultCycleTime);
    reloadOptionData();
}

void MainWindow::on_AddLightButton_clicked() {

    int phaseNumber = ui->ToPhaseComboBox->currentText().toInt();
    int roadNumber = ui->NearRoadComboBox->currentText().toInt();

    if (phaseNumber != 0 && roadNumber != 0)
    {
        if (SimulatorEngine->map->AddLight(0, phaseNumber, roadNumber))
        {
            ui->statusbar->showMessage(tr("Route Added Successfully."));
            reloadOptionData();
            return;
        }
    }
    ui->statusbar
        ->showMessage(tr("Could not add Traffic Light. View console for details"));
}

void MainWindow::on_PhaseTimeSlider_sliderMoved(int position) {

    int phaseNumber = ui->PhaseTimeComboBox->currentText().toInt();

    if (phaseNumber != 0)
    {
        if (SimulatorEngine->map->SetPhaseTime(phaseNumber, position))
        {
            ui->PhaseTimeSlider->setValue(position);
            ui->statusbar->showMessage(tr("Phase time set successfully"));
            ui->PhaseTimeLineEdit->setText(QString::number(position));
            return;
        }
    }
    ui->statusbar->showMessage(tr("Could not set phase time."));
}

void MainWindow::on_PhaseDelaySlider_sliderMoved(int position) {
    ui->PhaseDelaySlider->setValue(position);
    Settings::PhaseDelay = position;
    ui->PhaseDelayLineEdit->setText(QString::number(position));
}

```

```

        ui->statusbar->showMessage(tr("Phase delay changed."));
    }

void MainWindow::on_AssignLaneButton_clicked() {
    int phaseNumber = ui->AssignLaneToPhaseComboBox->currentText().toInt();
    Lane *lane = SimulatorEngine->map->SelectedLane;

    if (phaseNumber != 0 && lane != nullptr)
    {
        if (SimulatorEngine->map
            ->AssignLaneToPhase(phaseNumber, lane->GetLaneNumber()))
        {
            ui->statusbar->showMessage(tr("Orange Light delay changed."));
            reloadOptionData();
            return;
        }
    }

    ui->statusbar->showMessage(tr(
        "To Assign a lane to a phase, please select click on a lane to select
it."));
}

void MainWindow::on_PhaseTimeComboBox_currentTextChanged(const QString &arg1) {
    int phaseNumber = ui->PhaseTimeComboBox->currentText().toInt();
    if (phaseNumber != 0)
    {
        Phase *p = SimulatorEngine->map->GetPhase(phaseNumber);
        if (p != nullptr)
        {
            ui->PhaseTimeLineEdit->setText(QString::number(p->GetCycleTime()));
            ui->PhaseTimeSlider->setSliderPosition(p->GetCycleTime());
        }
    }
}

void MainWindow::on_PhaseTimeLineEdit_editingFinished() {
    int phaseNumber = ui->PhaseTimeComboBox->currentText().toInt();
    float value = ui->PhaseTimeLineEdit->text().toFloat();

    if (phaseNumber != 0)
    {
        if (SimulatorEngine->map->SetPhaseTime(phaseNumber, value))
        {
            ui->PhaseTimeSlider->setValue(int(value));
            ui->statusbar->showMessage(tr("Phase time set successfully"));
            ui->PhaseTimeLineEdit->setText(QString::number(value));
            return;
        }
    }
}

```

```

        ui->statusbar->showMessage(tr("Could not set phase time."));
    }

void MainWindow::on_PhaseDelayLineEdit_editingFinished() {
    float value = ui->PhaseDelayLineEdit->text().toFloat();
    ui->PhaseDelaySlider->setValue(int(value));
    Settings::PhaseDelay = value;
    ui->PhaseDelayLineEdit->setText(QString::number(value));
    ui->statusbar->showMessage(tr("Phase delay changed."));
}

void MainWindow::on_ShowLaneBlockCheckBox_stateChanged(int arg1) {
    Settings::DrawLaneBlock = arg1;
}

void MainWindow::on_DrawTexturesCheckBox_stateChanged(int arg1) {
    Settings::DrawTextures = arg1;
    ui->MultiColorCheckBox->setEnabled(arg1);
}

void MainWindow::on_FollowSelectedCarButton_stateChanged(int arg1) {
    Settings::FollowSelectedVehicle = arg1;
}

void MainWindow::on_AbortButton_clicked() {

    if (Settings::Speed != 0)
    {
        on_PauseButton_clicked();
    }

    QMessageBox msgBox;
    msgBox.setText("The Set has been aborted.");
    msgBox.setInformativeText("Do you wish to save the set in a file?");
    msgBox.setStandardButtons(
        QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel);
    msgBox.setDefaultButton(QMessageBox::Save);
    int ret = msgBox.exec();

    switch (ret)
    {
    case QMessageBox::Save:

        on_SaveSimButton_clicked();
        ui->AbortButton->setEnabled(false);
        ui->TrainingProgressBar->setHidden(true);
        break;

    case QMessageBox::Discard:
        // clear the map of running simulations and vehicles.
    }
}

```

```

SimulatorEngine->ClearMap();
// delete the current set.
SimulatorEngine->DeleteCurrentSet();
reload_sim_table();
ui->AbortButton->setEnabled(false);
ui->TrainingProgressBar->setHidden(true);
break;

case QMessageBox::Cancel:

    break;
}

if (Settings::Speed == 0)
{
    on_PauseButton_clicked();
}
}

void MainWindow::on_SaveSimButton_clicked() {
QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"),
                                                "simulations.json",
                                                tr("JSON Files (*.json)"));
SimulatorEngine->SaveSets(fileName.toStdString());
}

void MainWindow::on_LoadSimButton_clicked() {
QFileDialog dialog(this);
dialog.setFileMode(QFileDialog::ExistingFile);
dialog.setNameFilter(tr("JSON Files (*.json)"));
dialog.setViewMode(QFileDialog::Detail);

QStringList fileNames;
if (dialog.exec())
{
    fileNames = dialog.selectedFiles();
    SimulatorEngine->LoadSets(fileNames.front().toStdString());
    reloadOptionData();
    reload_sim_table();
    resize_sim_table();
}
}

void MainWindow::on_SimTable_clicked(const QModelIndex &index) {
ui->RunDemoButton->setEnabled(true);
ui->DeleteSimButton->setEnabled(true);
selected_row_ = index.row();
}

void MainWindow::on_RunDemoButton_clicked() {

```

```

int simNumber = model_->GetIdByRow(selected_row_);
if (simNumber != 0)
{
    SimulatorEngine->RunDemo(simNumber);
} else
{
    ui->statusbar->showMessage(tr("Could not run demo."));
}

}

void MainWindow::on_DeleteSimButton_clicked() {
    int simNumber = model_->GetIdByRow(selected_row_);
    if (simNumber != 0)
    {
        if (SimulatorEngine->DeleteSimulation(simNumber))
        {
            QString s = "Simulation ";
            s.append(QString::number(simNumber));
            s.append(" has been deleted.");

            ui->statusbar->showMessage(s);
            reload_sim_table();
            reload_sim_graph();
        } else
        {
            QString s = "Simulation ";
            s.append(QString::number(simNumber));
            s.append(" failed to be deleted.");
        }
    } else
    {
        ui->statusbar->showMessage(tr("Could not delete simulation."));
    }
}

void MainWindow::on_ShowMinimapCheckBox_stateChanged(int arg1) {
    Settings::DrawMinimap = arg1;
}

void MainWindow::on_DensityColorCheckBox_stateChanged(int arg1) {
    Settings::LaneDensityColorRamping = arg1;
}

void MainWindow::on_VehicleSpawnRateSlider_sliderMoved(int position) {
    ui->VehicleSpawnRateTextBox->setText(QString::number(position));
    Settings::VehicleSpawnRate = position / 1000.f;
}

void MainWindow::on_VehicleSpawnRateTextBox_editingFinished() {

```

```

        float value = ui->VehicleSpawnRateTextBox->text().toFloat();
        ui->VehicleSpawnRateSlider->setValue(value);
        Settings::VehicleSpawnRate = value / 1000.f;
    }

void MainWindow::on_ShowSelectedPhaseLanesCheckBox_stateChanged(int arg1) {
    Settings::ShowSelectedPhaseLanes = arg1;
    if (arg1)
    {
        reloadOptionData();
    } else
    {
        SimulatorEngine->map->UnselectAll();
    }
}

void MainWindow::on_AddCycleButton_clicked() {
    int intersectionNumber = 0;
    if (ui->IntersectionNumberComboBox->isEnabled())
    {
        intersectionNumber =
            ui->IntersectionNumberComboBox->currentText().toInt();
    }

    if (SimulatorEngine->map->AddCycle(0, intersectionNumber) != nullptr)
    {
        reloadOptionData();
        ui->statusbar->showMessage(tr("Cycle successfully added."));
        return;
    }

    ui->statusbar->showMessage(tr("ERROR: Could not add cycle."));
}

void MainWindow::on_AssignToIntersectionCheckBox_stateChanged(int arg1) {
    ui->IntersectionNumberComboBox->setEnabled(arg1);
}

void MainWindow::on_RemoveLaneFromPhaseButton_clicked() {
    if (ui->AssignedLanesListView->selectedItems().count() == 1)
    {
        int laneNumber =
            ui->AssignedLanesListView->selectedItems().back()->text().mid(5)
            .toInt();

        SimulatorEngine->map->UnassignLaneFromPhase(laneNumber);
        reloadOptionData();
        ui->statusbar->showMessage(tr("Selected lane unassigned from phase."));
        return;
    }
}

```

```
}

ui->statusbar->showMessage(tr(
    "Please select a lane from the list to unassign it from its phase."));

}

void MainWindow::on_ShowNeuralNetCheckBox_stateChanged(int arg1) {
    Settings::DrawVisualNet = arg1;
}

void MainWindow::on_RunBestCheckBox_stateChanged(int arg1) {
    Settings::RunBestNet = arg1;
}
```