The “Gheorghe Asachi” Technical University of Iasi

Faculty of Electronics, Telecommunications and Information Technology

Specialization: Telecommunication Technologies and Systems

Diploma Thesis

Scientific Coordinator:

Associate Professor Adriana Sîrbu, PhD.Eng.

Graduate:

Vacariu Dragos

IASI 2018

The “Gheorghe Asachi” Technical University of Iasi

Faculty of Electronics, Telecommunications and Information Technology

Specialization: Telecommunication Technologies and Systems

Diploma Thesis

Traffic simulators for studying ad-hoc vehicular networks

Scientific Coordinator:

Associate Professor Adriana Sîrbu, PhD.Eng.

Graduate:

Vacariu Dragos

IASI 2018

Table of Content

Introduction

With the rapid advancements in the automotive industry, vehicles are now coming with equipped sensors, on board units and other processing as well as communication capabilities. Because of this matter, the VANET has come into existence, holding the opportunity to make people's life and death decisions by predicting and helping the drivers and the pedestrians on the road feel safer.

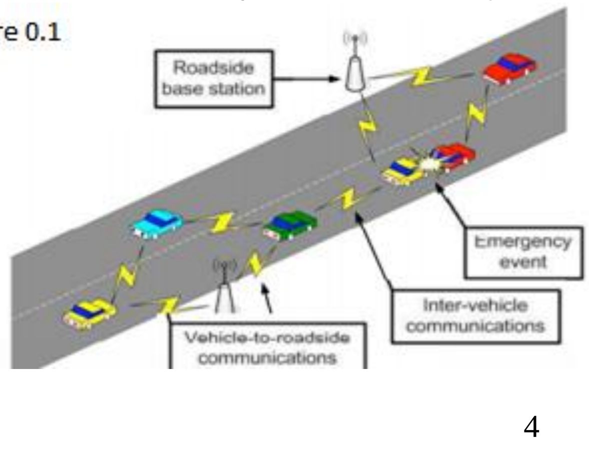
The road traffic conditions in Europe affect the safety of the population: around 40,000 people die and more than 1.5 million are injured every year on the roads, as one of the latest studies shows. In addition, the traffic jams generate huge waste of time and fuel which affects not only the economy but also increases the environmental impacts of surface transportation travel.

From the last decade, mobile communication techniques have transformed the automotive industry by providing anytime anywhere communication between different devices. This ease of communication allows exchange of valuable information between devices just on the go. The advances in the information technology and communication have easily supported the idea of communication between mobile devices and among these advancements the concept of Vehicular Ad-hoc Networks came into light opening new possibilities to avail the use of safety applications.

A Vehicular Ad-hoc Network or VANET refers to a network created in an ad-hoc manner where different moving vehicles and other connecting devices come in contact over a wireless medium and exchange useful information to one another. A small network is created at the same moment with the vehicles and other devices behaving as individual nodes. Whatever the information a node possess is transferred to all the other nodes. Similarly, each node after transferring their set of data receive a new set of data from the other nodes. After accumulating all the data, the nodes work to generate useful information out of the data and then again transmit the information to other devices.

The vehicles which are being launched in the market are now coming with equipped on board sensors which make it easy for the vehicle to join in the network and to benefit of VANET.

Figure 0.1


Roadside base station
Emergency event
Inter-vehicle communications
Vehicle-to-roadside communications

Vehicular Networks & Traffic Flow

What is a vehicular network?

It is a network which serves as an important technology required for the implementation of the most applications related to vehicles, vehicle traffic, drivers, pedestrians or passengers. An example of such applications is the emergency driver assistant, a system which monitor the behavior of the driver, and in case of a medical emergency in which the system concludes that the driver is no longer fit for driving the vehicle, the car takes the control of the brakes and steering until it completely stops safely.

Some other application could and might be made in the near distant future: just imagine a GPS system which is able to inform the driver about any incoming vehicles in its way, or a GPS system which is able to compute the time required to reach a certain point as destination, according to the real-time traffic and meteorological conditions.

The excitement about this particular topic is not only due to applications presented, but also because of the potential benefit to provide large scale solutions for safety, time-saving, blockades and busy city sectors prevention.

How to achieve that?

In order to achieve the presented objectives, some project opportunities were presented between 2000 and 2009, some of them will be enumerated and detailed below:

Intersection collision warning: consist in a project which targets to decrease the risk of lateral collisions for vehicles that are approaching road intersections, by road side unit detectors which would inform the signaled driver about the possible income of such action.

Lane change assistance: consist in a project which targets to decrease the risk of lateral collisions for vehicles that are accomplishing a lane change within a blind spot.

Overtaking vehicle warning: consist in a project which aims to prevent collisions between vehicles which involve in overtaking situations. The warning will be triggered when a vehicle involves in overtaking another vehicle which is already involved in overtaking a third vehicle.

Co-operative merging assistance: consist in a project which aims to provide a system which cooperates with drivers in junction merging maneuver situations. When vehicles involve in merging maneuvers the system will cooperate with road side units which will ensure that the maneuver can be performed safely.

Wrong way driving warning: consist in a project which aims to detect whether a driver is driving on the wrong side of the road. In such case the forbidden heading would signal this situation to the other vehicles and road side units.

Rear end collision warning: consist in a project which aims to prevent the risk of rear-end collisions due to slow downs or road curvatures. The driver of a vehicle in front will be informed on a possible risk of rear-end collision, whereas the driver in his back will receive a warning as well, about head on collision possibility.

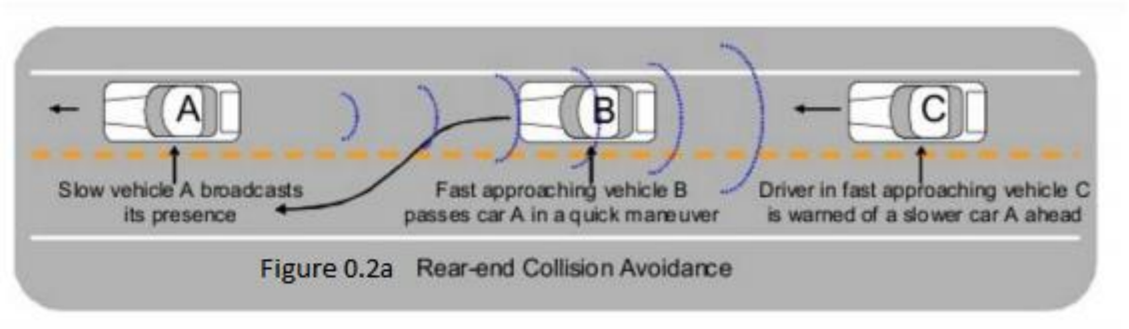

A
B
C
Slow vehicle A broadcasts its presence
Fast approaching vehicle B passes car A in a quick maneuver
Driver in fast approaching vehicle C is warned of a slower car A ahead

Figure 0.2a Rear-end Collision Avoidance

Head on collision warning: is a project which aims to prevent the risk of head-on collisions by sending early warnings to vehicles which are travelling in opposite direction, or to warn the driver when is approaching on vehicles which are slowing down.

Traffic condition warning: is a project which aims to warn any vehicle when detecting some rapid traffic evolutions. In such case the blockades and busy city sectors could be prevented.

Speed management: is an application which aims to assist the driver in maintaining safe speed according to road sectors, weather conditions or legislative regulations.

Stationary vehicle warning: is an application which aims to detect vehicles which were disabled due to accidents, breakdowns or any other reasons. When such a situation is detected, the other traffic participants will be informed about that.

Emergency electronic break lights: is an application which aims to provide support in case of hard braking a vehicle. In such case the vehicle which has to hard brake informs the other vehicles by using the cooperation of other vehicles of the road side units, about the situation.

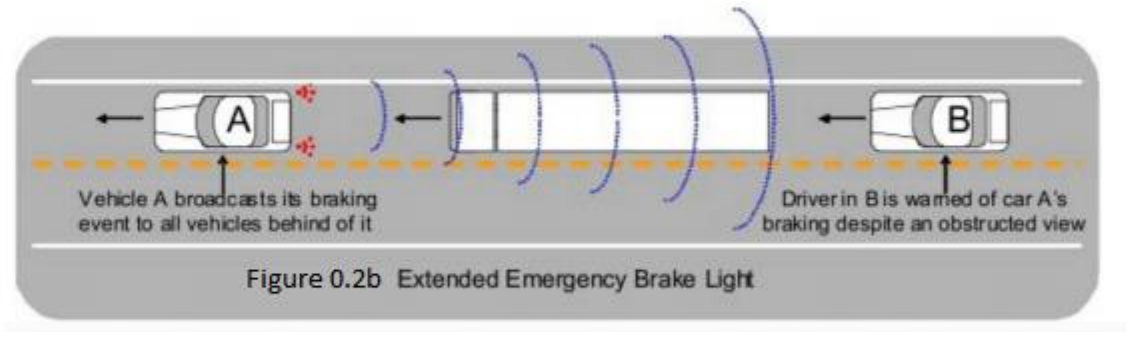

A
B
Vehicle A broadcasts its braking event to all vehicles behind of it
Driver in B is warned of car A's braking despite an obstructed view

Figure 0.2b Extended Emergency Brake Light

Hazardous location notification: is an application which ensures that any vehicle or road side unit which encounters an obstacle, construction work, or slippery road condition can broadcast a signal to the other vehicles in the nearby area.

Signal violation warning: is an application used to broadcast a warning to all the neighborhood vehicles each time when a signal violation is detected by the road side units. The violation information gets also broadcasted by the road side unit.

What is an ad-hoc network?

An ad-hoc network or VANET is a considered central part of intelligent transportation systems which enables all the traffic participants to exchange information and coordinate their behavior. As no underlying infrastructure is required and the message exchange is carried out with low latency, VANET is considered an excellent tool to reduce congestion. Applications aiming to ease the congestion require information about local vehicle density to identify the congested roads and to offer an alternative. There is always a risk of causing congestion on the alternative routes, as well, so there are some innovative approaches which search to determine the driving behavior which leads to breakdowns. An approach was to use the vehicle-to-vehicle communication, in which messages would be periodically broadcasted containing statuses about the vehicle's position, velocity, acceleration, and time stamp.

VANETs allow a tight connection between the physical driving and the communication systems, which provides the need of an application oriented approach. These networks are created by applying the principles of mobile ad hoc networks (MANETs), which provides spontaneous creation of a wireless network for data exchange to the domain of vehicles. In general, a Mobile ad hoc network is a collection of wireless nodes communicating with each other in the absence of any infrastructure.

Computer simulation have become a valuable instrument in the field of research which is due to the unfeasibility of the large-scale world experiments. Simulators are mainly used to assess the impact of a proposed strategy with both very realistic propagation and traffic models.

A critical aspect in a simulation, is the need for mobility models which reflects as close as possible the real behavior of vehicular traffic. When dealing with vehicular traffic simulations, two main mobility models can be distinguished: macro-mobility models, and micro mobility models.

Macro-mobility models: take into consideration all the macroscopic aspects which influence the vehicular traffic: the road topology, constraining cars movement, the per-road characterization, the speed limits which are defined, the number of lanes the street has, overtaking and safety rules over each street in the topology.

Micro-mobility models: take into consideration the aspects which refers to the driver's behavior such as: the behavior that a driver has when is interacting with other drivers, or the traveling speed a driver has in different traffic conditions, the acceleration, deceleration and overtaking criteria, the behavior in presence of road intersections and traffic signs. The general driver's attitude is related to factors such as: age, sex, or mood.

It would be desirable for VANETs simulations that both macro-mobility and micro-mobility model descriptions to be considered in modeling vehicular movements, but practically, many non-specific mobility models employed in VANETs simulations ignore these guidelines, and fail to reproduce the particular aspects of vehicular motion, such as: car acceleration and deceleration in presence of nearby vehicles, queuing at road intersections, clustering caused by semaphores, vehicular congestion and traffic jams.

Traffic flow dynamics is a field of traffic science which can be distinguished by time scale studies. Traffic flow dynamics in the real world includes time scales ranging from about one second to a few hours. Human reaction times and the time gap between two vehicles following each other can take up to 1s, while braking and acceleration maneuvers typically take several seconds. In the city traffic, the period of one red-to-green cycle of traffic lights take about 1 min while, on freeways, the period of traffic oscillations and stop-and-go waves take between 5 and 20 min. Finally, the traffic demand serving as variable (model input) for traffic flow models varies on time scales from one hour, to which is illustrated by the term “rush hour”.

The dynamics on time scales which are smaller than one second are relevant for car manufacturers and typically to applications which include the control of vehicle components such as engine, brakes, and transmission, the dynamics of skidding, and the operation of various assistance systems such as electronic stability programs, airbags, and adaptive cruise control.

There are numerous applications for traffic flow dynamics and simulation including the following: generation of surrounding traffic in driving simulators, determination of optimal routes in connected (traffic-dependent) navigation systems, the optimization of logic behind operation of traffic lights, exploitation of the location for finding the nearest fuel station, motels, cafes etc.., maintenance of minimum security distance, emergency vehicle warning, and even parking assistant.

The ideal way of observing traffic evolution is by mounting cameras on the top of tall building, or on airplanes. In this way tracking softwares can extract trajectories, and positions from each vehicle over time, from video footage or from a series of photographs. That’s also one of the reasons why 2D traffic representation can be more useful than 3D.

The trajectory data is the most comprehensive data available, being the only data which allows direct and unbiased measurements on the traffic density and the frequency of lane change. The idea of trajectory extracting was developed within some manufacturer’s navigation systems, which can record and send it the information back to them.

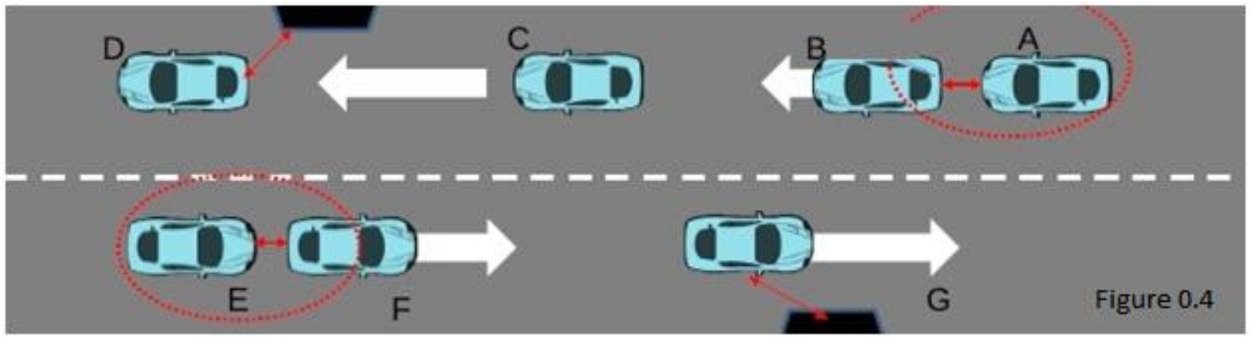


Figure 0.4

In the project I am going to present the Traffic simulator make use of 2D graphics and 20ms frame rendering, which provide a maximum of 50 FPS. Using a camera as mentioned above, will provide a 2D overview on the traffic.

A Simulator's Architecture

A VANET simulator has two main components: a **network component** – which is capable of simulating the behavior of a wireless network, and a **vehicular component** – which can provide accurate mobility model for the nodes of VANET.

In order to describe these components, we consider a microscopic discrete-event simulator which would include components for event management such as: an **even queue** which will register the events in the same order as they occur, the **modules for event processing** and a **logical clock** for the **simulation time management**.

The **simulation time** is quantified in fixed pieces, so that each event can be associated to a specific timestamp with an acceptable resolution. At every moment in the simulation time, the **Event Engine** pulls the current events from the queue and handles them in a random order by calling the event processing modules. The vehicular traffic component is being represented by the mobility module.

The **Event Queue** can hold three types of events: **send**, **receive** and **GPS**. A **send** event for a specified node would trigger the call to the node's procedure responsible for preparing a message. It also schedules the corresponding **receive** event for the message receiver that are determined by the simulator according to the wireless network. The **receive** event is associated with a node or a group of nodes to which the message is **transmitted**. Its action is to call the appropriate **handler** in each of the receiving nodes. A **handler** is represented by an individual function.

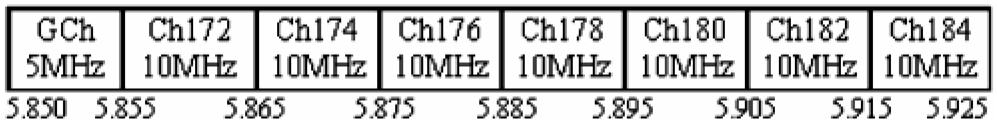
The **GPS event** is scheduled at a regular time interval for each node, in order to simulate the way an actual VANET application collect data periodically from the GPS.

The **mobility module** repeatedly updates the position of each note which is being represented by an individual vehicle. This model is called the **vehicular mobility model** and it takes into account vehicle interactions like: passing by, car following patters, changing lanes, traffic rules, the behavior of different drivers etc.

The main advantage of such an architecture is the possibility to execute or emulate the actual code of a vehicular application without significant changes, by using appropriate simulator interfaces.

The role of short range communication

In VANET all type of communication are wireless, a special wireless frequency band assigned to VANET is called DSRC (dedicated short range communication). A DSRC is a new frequency band that is allocated by Federal Communication Commission (FCC); which purpose is to provide application for public safety. The DSRC provides multiple channels with transmission range between 5.850 and 5.925 GHz. This frequency band is divided into seven channels and each channel range is about 10 MHz. The data transfer rate that DSRC provides is up to 27 Mbps, having an important role because it makes possible for vehicles to communicate with each other and also with the infrastructure.

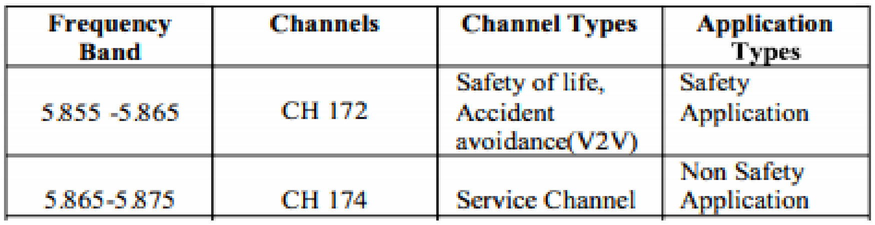


DSRC Channel Structure

DSRC provides three basic channels:

1. Reserved Channel: First 5 MHz is frequency band reserved; it is called the guard channel. It is the lowest spectrum in whole frequency band of DSRC, consisting of two types of channels, safety application and non-safety application channels.
2. Services Channel: which are divided into two categories:
    a. Category A: The channels: CH 172 and CH 184 are used in this category, being reserved for safety related applications.
    b. Category B: The channels CH174, CH 176, CH180, and CH182 are used in category B. These channels are used for non-safety applications. The number of non-safety application is very large, that's why four channel are assigned for it.
3. Control Channel: The channel: CHl78 is a Control channel, generally used for safety related application, broadcast messages and also to provide advertise services.

DSRC CHANNEL, FREQUENCY RANGE AND APPLICATION TYPES



Frequency Band	Channels	Channel Types	Application Types
5.855 -5.865	CH 172	Safety of life, Accident avoidance(V2V)	Safety Application
5.865-5.875	CH 174	Service Channel	Non Safety Application

5.875-5.885	CH 176	Service Channel	Non Safety Application
5.895-5.905	CH 180	Service Channel	Non Safety Application
5.905-5.915	CH 182	Service Channel	Non Safety Application
5.885-5.895	CH 178	Control Channel	Safety Application
5.915-5.925	CH 184	Public safety, High power and long range	Safety Application

Safety of users' life is the basic user requirement on the road. The main problem in the current system is if one accident happen due to some reasons no one pass this message to the other vehicles and many other vehicles collide there. Here we need an application that avoid such kind of situations. VANET provides two types of potential applications: safety and non-safety applications which facilitate the users. Safety applications guide the users by sending some warning messages and avoid accidents on the road. Non-safety applications deal with comfort of the users and the journey providing some entertainment services.

The proposed model of user requirements is divided into a three-layer structure. First layer describe the basic user's requirements which are Security, privacy and trust. Second and third layers are consisting of the safety and non-safety applications of the VANET and these two layers meet the first layer requirement. Objective of this model is to serve the users and to reduce the death rate using VANET applications.

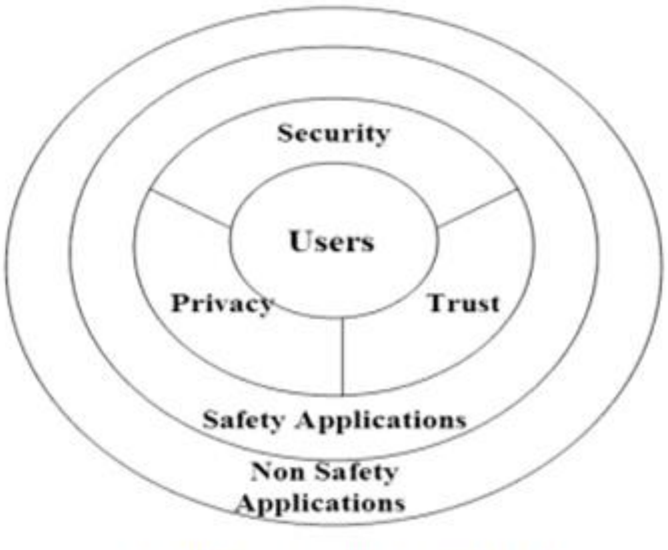


Figure 0.5 User Requirements Model

1. **Security:**

Security is an important user's requirement in VANET as if VANET is not secure from attackers then these applications may not achieve the primary purpose which is to make the human lives safer. For example: one wrong message may create a trouble on the road because the medium is open and speed of the vehicle is so high and users do not have time to verify the message before its execution.

Safety related applications may not work properly without achieving minimum security level. For instance: Extended Brakes Light (EBL) application needs security otherwise an attacker may generate warning messages and create problems on road. Some security requirements are mentioned here. If the system meet these entire requirements then it should be safe from attackers.

2. **Privacy:**

The content of the message should be secure and it should not be accessible to unauthentic users. Unauthorized person could create problems for legitimate users if they access the life safety messages. Privacy in VANET is to secure the user's personal data and their location. When user sends any message to other vehicles, it should not affect their privacy. The users need privacy and may not allow seeing their personal data and their locations. Only authorized parties (such as police, law enforcement agencies) may use the personal information of a particular user.

The personal information relevant to authorized parties could be:

- The name of the driver;
- The license plate of the vehicle;
- The current speed of the vehicle
- The location and position of the vehicle
- The route for travelling;
- Some internal sensor data within the car: Example: a sensor which checks the alcohol in the expired air.

3. **Trust:**

Trust is the key element for the security of the system. When the user has ensured that VANET safety and non-safety applications are secure and by using these applications their privacy could not affected, then the level of trust should be increased. When users receive any messages from other nodes or from the infrastructure itself, it should be trusted because the user reacts according to the message. To establish the trust, it is required to provide trust between the users in the communication of vehicle to vehicle (V2V) and vehicle to infrastructure (V2I).

4. **Safety applications:**

The VANET is a very important part of intelligent transport system (ITS). Safety applications are the most important application of VANET because it is directly related to the users and its priority is high due to human life saving factor. The main goal of safety application is to provide safety of vehicles and its passengers from road accidents.

5. **Non-Safety applications:**

Non-Safety applications would contain: convenience applications - road status services, parking availability services, comfort applications - personal and diagnostics services.

Road Status Services are part of convenience applications which main objective is to provide information about the road, if any kind of problem appears on the road. Basic task is to detect and notify about the congestion on the road and sends this information to other users. Using this kind of information, the users could adopt alternative routes for traveling, with such an application people could save their time and fuel.

Personal and Diagnostics Services also called PDS, is part of comfort applications having the main objective of helping users to download or upload personalized vehicle settings or diagnostics to the infrastructure, or from the infrastructure.

Other services:

Map Download Services – it is a type of portal that provides valuable information about a certain area where the user is driving. Maps are available and can be downloaded from mobile hotspots or home station about the specific location.

Entertainment Services – is a type of service who aims to entertain the user during a journey by allowing him to watch any movie or favorite program.

The following figure will provide an overview for the other services available:

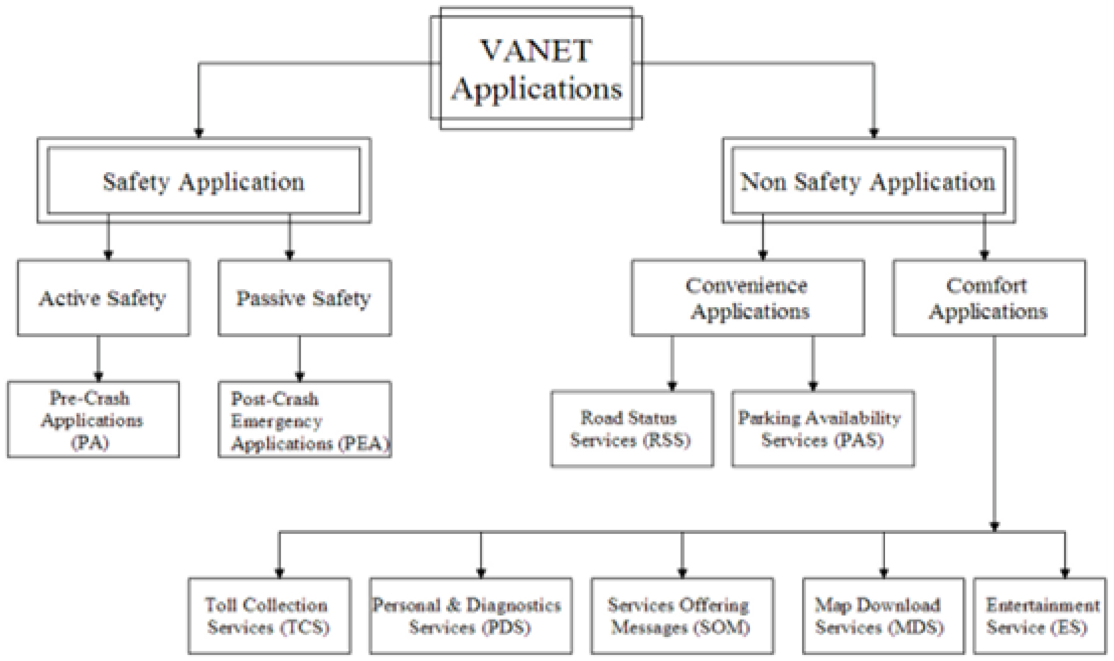

VANET Applications
Safety Application
Non Safety Application
Active Safety
Passive Safety
Convenience Applications
Comfort Applications
Pre-Crash Applications (PA)
Post-Crash Emergency Applications (PEA)
Road Status Services (RSS)
Parking Availability Services (PAS)
Toll Collection Services (TCS)
Personal & Diagnostics Services (PDS)
Services Offering Messages (SOM)
Map Download Services (MDS)
Entertainment Service (ES)

Figure 0.6 Summary of all Applications

Regarding the applications which are available you will find: commercial applications – to provide web access and audio/video streaming, productive applications – to provide support for time utilization, fuel saving, and environmental benefits.

Required knowledge for building a simulator

The knowledge which was required for building a virtual traffic simulator such as: programming languages, development libraries, algorithms, graphic APIs, concepts, paradigms and so on… will be listed and explained within this work section.

C vs C++ Programming languages

What is C++?

C++ is a programming language developed by Bjarne Stroustrup in 1980. The main purpose of it was to add Object Oriented Programming concepts to the C language. The C language was developed earlier in 70's by Denis Ritchie and Brian Kernighan, the same people who originated the **K&R Notation**, the original dialect of C programming language which was first introduced into the book "**The C Programming Language**".

The main differences between the two programming languages can be seen as it follows:

C Programming Language	**C++ Programming Language**
It supports only procedural programming for code development.	It supports both procedural programming and object oriented programming for code development
Since C is not an object oriented programming language it has no support for polymorphism.	Being an object oriented programming language C++ supports all the fundamental OOP concepts: polymorphism, encapsulation, inheritance, and abstraction.
Data and functions are free separated entities.	When creating Objects data and functions are combined together to form a single whole new entity.
It does not support function overloading.	It supports function overloading.
It does not have the namespace feature.	It supports the namespace feature.
It does not provide support for error handling.	It provides support such as exceptions handling for supporting the errors.
It uses functions such as: scanf() and printf() for inputting and outputting.	It uses objects such as: cout and cin for inputting and outputting
C is only a part of C++	C++ contains all the C functionality, combined with a lot of different alternatives.

When to use C language:

- When needing a portable code across different platforms. The C compilers exists for enormous number of platforms, whereas C++ compilers are limited to a small number of platforms (example: PC's);

- When the platform does not support C++ (example: in case of Embedded Systems);
- When there are performed interactions with other languages (example: lower level languages like: assembler);
- When developing applications with implied amount of resources, C allows manual memory management.
- A feature of C programs is that they tend to run faster that C++ ones.

When to use C++ language:

- When developing application which make use of complex data types;
- When the programmer efforts need to be minimalized;
- When needing richer programming libraries;
- When needing error support or exception handling.
- When developing large scale projects;

Why did I used C++?

- It was easier to structure my code.
- It was easier to work with and API
- The length and complexity of the code was improved.
- For the OOP features;
- Because of the rich pre-defined libraries that comes with it.
- Lower efforts to achieve my goal.

What is OOP?

OOP (meaning Object Oriented Programming) is paradigm based on the concept of "**objects**". An object is an instance of a class, which is a complex data type containing fields and properties.

A **field** is a member of a class consisting in basic data types such as: **int**, **float**, **double**, **char**, **string**, etc.

A **property** is a method or a function which by the concept is used to describe the functionality of the **object**.

Both **fields** and **properties** can use access-modifiers such as: **public**, **private** or **protected** in order to provide encapsulation. The **encapsulation** is one of the fundamental OOP concepts which consist in binding the fields and the properties which manipulate those fields, in order maintain the proper control of the values which gets passed in or out from that class. In other words: encapsulation is used to provide read-only, write-only or read-write policy to the fields inside the class, making use of functions called getters or setters whom purpose is to return or modify the value of a private/protected member.

The main concepts of OOP are: **inheritance**, **encapsulation**, **polymorphism**, and **abstraction**.

Inheritance is the concept which provides the extension of the functionality of a class into another. In other words by inheritance a class can provide its functionality to the derived classes, just like how parents can provide their features to the children they give birth to.

Polymorphism is the concept which applies in case of different inherited complex data types which share the same functionality but overloaded. In other words, if a parent class contains functionality which can be overloaded by the child class, using polymorphism the child class will be able to choose its own definition regardless of the parent's.

Abstraction is the concept which provides a way in which a class cannot be instantiated (object cannot be created) marking it in this way as being a parent-only class, which means that the class can be inherited and its functionality can be extended somewhere else but not used by its own objects. In order to provide an abstract class in C++ at least one of that class properties needs to be set as **virtual**.

Some build-in C++ classes are: **Vector**, **Ifstream**, **String**, **Exception**, **Ofstream**, **Stringstream.**

OpenGL

What is OpenGL?

Is an application programming interface (API) which is used for 2D and 3D graphics rendering. The API interacts with the GPU for achieving hardware accelerated rendering, the definition of it consists in a set of independent functions which may get called by the client. Being a language-independent makes OpenGL a cross-platform.

The first version of OpenGL was released in 1997 and it was called OpenGL 1.1 since then 17 more version were released the last one was OpenGL 4.6 released on 31 July 2017.

The history of all the OpenGL versions and their improvements across the time can be seen below:

Version:	**Improvements:**
OpenGL 1.1	Introduces the texture objects
OpenGL 1.2	Introduces the 3D textures
OpenGL 1.3	Adds multi-texturing, multi-sampling, texture compression
OpenGL 1.4	Adds the depth textures feature.
OpenGL 1.5	Introduces the Vertex Buffer Object.
OpenGL 2.0	Introduces the GLSL 1.1
OpenGL 2.1	Introduces the GLSL 1.2 which adds the Pixel Buffer Object and sRGB textures
OpenGL 3.0	Introduces the GLSL 1.3 which adds the Texture Arrays, Conditional rendering and the Frame Buffer Object.
OpenGL 3.1	Introduces the GLSL 1.4 which adds the Texture Buffer Object and the Uniform Buffer Object
OpenGL 3.2	Introduces the GLSL 1.5 which adds the Geometry Shader, and the multi-sampled textures
OpenGL 3.3	Introduces the GLSL 3.30 which adds new functionality such as: 64-bit precision for rendering.

OpenGL 4.0	Introduces GLSL 4.00 with Tessellation on GPU
OpenGL 4.1	Introduces GLSL 4.10 with developer friendly debugger
OpenGL 4.2	Introduces GLSL 4.20 with Shaders using atomic counters and shader packing for performance improvements.
OpenGL 4.3	Introduces GLSL 4.30 which provides the feature of compute shaders leveraging GPU parallelism.
OpenGL 4.4	Introduces GLSL 4.40 with Buffer Placement Control.
OpenGL 4.5	Introduces GLSL 4.50 with Direct State Access and DX11 emulation features.
OpenGL 4.6	Introduces GLSL 4.60 with more efficient geometry processing and shader execution.

What is a shader?

A **shader** is a user-defined program which is implemented with the purpose of executing one of the programmable stages for the rendering pipeline. There are several types of shaders each of them serving their own purpose:

Vertex Shader – is the programmable shader stage in the pipeline which handles the processing of each individual vertex. In this stage each vertex gets draw and connected with the others from the same set, inside the window.

Evaluation Shader – is the shader stage which takes the result provided by Tessellation operation and computes the interpolated positions and other vertex data.

Geometry Shader – is the programmable shader stage which handles the processing of primitives. A primitive is a vertex stream or an ordered list of vertices. The geometry shader is optional.

Fragment Shader – is the programmable shader stage which handles the processing of the fragments generated by the Rasterization process, into a set of colors. The Rasterization is the process in which each primitive (discussed above) is broken down into discrete elements called fragments. You can think at the Rasterization process, as the method of selecting vertex by vertex from the vertex stream.

Compute Shader – is the programmable shader stage which gets used for computing arbitrary information while rendering. Usually this shader is used for task which doesn’t imply the vertices or fragments which are being rendered.

The rendering pipeline of OpenGL can be observed in the following figure:

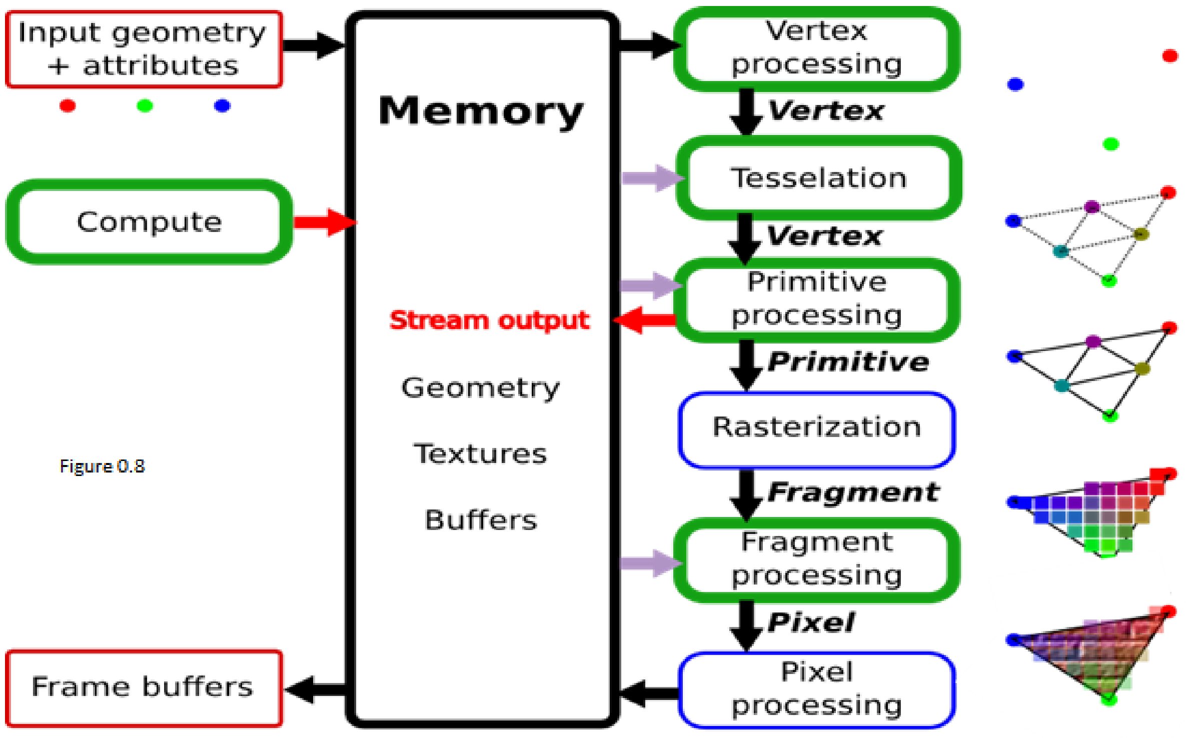


Figure 0.8

Each shader will be programmed using **GLSL** (OpenGL Shading Language).

Some examples of shaders can be seen below:

```
//This is the vertex shader:
#version 400                //the version of the GLSL used.

in vec3 position;           //the input vector
in vec3 VertexColor;        //the input vector

out vec3 Color;             //the output vector carried out to fragment shader

void main()
{
    Color = VertexColor; //The color to be set in fragment shader is the
//color which came through the vertex shader;

    gl_Position = vec4(position, 1.0); //this will draw the received
//vertices;
}

//This is the fragment shader:
#version 400                //the version of the GLSL used.

in vec3 Color;     //the input vector

void main()
{
      FragColor = vec4(Color, 1.0); //The color to be set, is the color
      //received with 1.0 opacity.
}
```
As you can see, the GLSL language is very similar to C programming language. The main purpose of GLSL was to provide more direct control over the graphics pipeline. Before GLSL the assembly language was used in this context.

How to use these shaders?

In order to use these shaders, the main C program needs to read the code and send it through the pipeline for its compilation as it follows:

```
// These are the strings which will store the code for each shader
std::string vertexSource = "void main {/*My vertex shader code*/}";
std::string fragmentSource = "void main {/*My fragment shader code*/}";

// Creating a vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);

// Send the vertex shader source code to the graphics card
const GLchar *source = (const GLchar *)vertexSource.c_str();
      /* Note that std::string.c_str is NULL character terminated.*/
glShaderSource(vertexShader, 1, &source, 0);

// Compile the vertex shader
glCompileShader(vertexShader);

//Check the compilation results:
GLint CompilationResult = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &CompilationResult );
if( CompilationResult == GL_FALSE )
{
      GLint msgLength = 0;
      glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &msgLength);
      std::vector<GLchar> infoLog(msgLength); /*creating a vector for the
message log.*/
      glGetShaderInfoLog(vertexShader, msgLength, &msgLength, &infoLog[0]);
/*getting the message log*/

      // Delete the shader, I don't need it anymore.
      glDeleteShader(vertexShader);

      //Print the error message to the screen.
      cout << infolog[0] << endl;

      // In this simple program, just leave
      return;
}
```
Similarly I'll do for the fragment shader:

```
// Creating a fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);

// Send the vertex shader source code to the graphics card
source = (const GLchar *)fragmentSource.c_str();
glShaderSource(fragmentShader, 1, &source, 0);
```

```
// Compile the fragment shader
glCompileShader(fragmentShader);

glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &CompilationResult);
if (CompilationResult == GL_FALSE)
{
      GLint msgLength = 0;
      glGetShaderiv(fragmentShader, GL_INFO_LOG_LENGTH, &msgLength);

      // The msgLength includes the NULL character
      std::vector<GLchar> infoLog(msgLength);
      glGetShaderInfoLog(fragmentShader, msgLength, &msgLength, &infoLog[0]);

      //Delete the shader the both shaders, I can't use either of them now
      glDeleteShader(fragmentShader);
      glDeleteShader(vertexShader);

      //Print the error message to the screen.
      cout << infolog[0] << endl;

      //Leave
      return;
}
```

Having the shaders compiled successfully, I’ll need to have them linked to form a single program:

```
// At this point vertex and fragment shaders are successfully compiled.
// Now it's time to link them together into a program.
// Get a program object.
GLuint program = glCreateProgram();

// Attach our shaders to our program
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);

// Link our program
glLinkProgram(program);

GLint isLinkedOkay = 0;
glGetProgramiv(program, GL_LINK_STATUS, (int *)&isLinkedOkay);
if (isLinkedOkay == GL_FALSE)
{
      GLint msgLength = 0;
      glGetProgramiv(program, GL_INFO_LOG_LENGTH, &msgLength);

      // The maxLength includes the NULL character
      std::vector<GLchar> infoLog(msgLength);
      glGetProgramInfoLog(program, maxLength, &msgLength, &infoLog[0]);

      // I don't need the program anymore.
      glDeleteProgram(program);
      // I don't need the shaders anymore.
      glDeleteShader(vertexShader);
      glDeleteShader(fragmentShader);
      cout << infolog[0] << endl;
```

```
    return;
}

// Always detach shaders after a successful link.
glDetachShader(program, vertexShader);
glDetachShader(program, fragmentShader);
```

At this moment, if everything worked as expected. I have shaders running at the GPU level.

C vs GLSL

In the following table it will be presented some different aspects regarding the 2 languages:

C Language	GLSL Language
- General-purpose programming language	- Domain specific programming language
- It doesn't know what a "texture" is.	- It understands the meaning of "texture".
- The code written gets executed at the CPU level.	- The code written gets executed at the GPU level.
- It provides cross-platform programming.	- It provides cross-platform programming.
- Low-level programming language.	- High-level programming language
- Recursion is allowed.	- Recursion is forbidden.
- Supports loops, branching, if-else statements, switch, user-defined functions, user-defined data types, memory management, arrays, strings, pointers, macros, modularity.	- Supports loops, branching, if-else statements, switch, user-defined functions.

The GLSL shaders needs to be passed to the hardware driver for the compilation. The compilation is made within an application which uses OpenGL API.

Simple DirectMedia Layer

Also called SDL is a cross-platform development library designed to provide low-level access to audio, keyboard, mouse, joystick, and graphics hardware using OpenGL or Direct3D (also called DirectX). This library can handle video, audio, networking, and multi-threading as well.

The library was internally written in C language.

Why do I use SDL?

One of the main topics I am interested in, while using SDL is the OpenGL context handler.

A thing which cannot be provided by the OpenGL library itself is the context. In order to draw something with OpenGL I need a context, which you can think of as a piece of paper. The OpenGL provides all the colored pencils that you might need, but no paper, and for that there is the need of using additional libraries.

The known libraries which can provide such context are: SDL, GLFW, FREE-GLUT, GLUT, each of those provides different features and functionality.

SOIL (Simple-OpenGL-Image-Library)

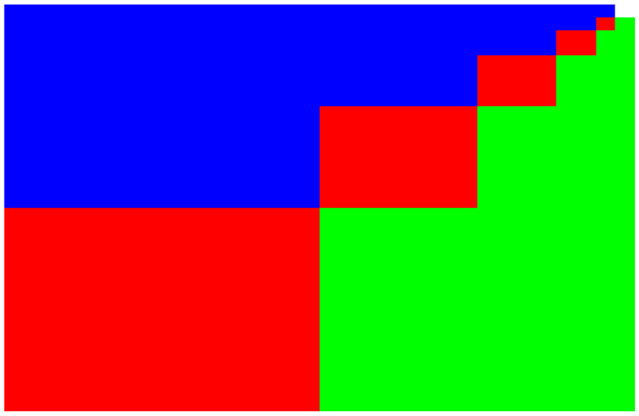
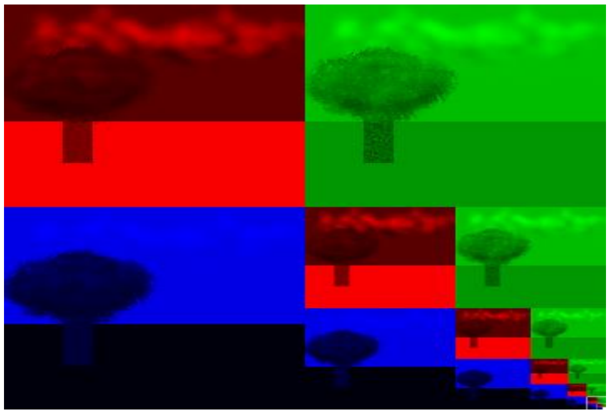
Soil is a tiny library which can be used for uploading images and textures to OpenGL. In order to texture a form, you need a way to upload that texture, SOIL provides that way. The most interesting feature that SOIL has despite the other similar libraries is the “Alpha color channel handling”. The alpha channel as it is called in graphics, represents the transparent pixels which can be found in some image formats such as: PNG or GIF.

The other features that Soil has is the ability to modify and save the modifications of the images in the following formats: TGA, BMP, DDS with RGB or RGBA channels (with transparency or not). Amongst the modification that SOIL can make are:

- Can flip images vertically.
- Can multiply alpha on load (this is performed in order to correct the blending).
- Can scale and rescale in a safe RGB value range.
- Can convert RGB channels into YCoCg color space channels.
- Can create mipmaps.

What is a mipmap?

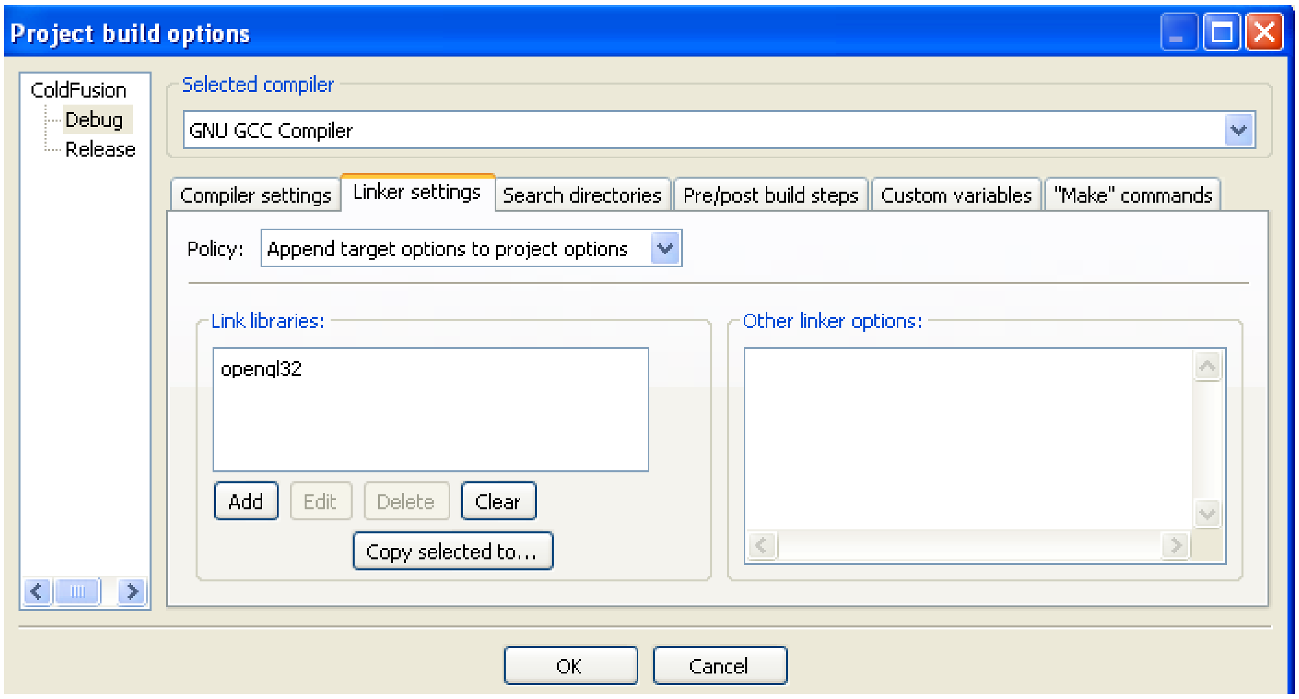
A mipmap (or MIP map) is an optimized sequence of images, where each sequence is part of the same image rendered progressively with a lower resolution than it is the original image.

Creating the basics

Before heading into further details about the project, I need to set up the environment, as well as the basics of the development.

The IDE used for the project is CodeBlocks. In order to be able to use OpenGL as well as the other libraries mentioned in the introduction part, I need to set the linker. Since OpenGL is embedded into each IDE, the only setting which needs to be done is adding the name of the **Dynamic Linked Library** file at the linker section:



Selected compiler: GNU GCC Compiler

Policy: Append target options to project options

Link libraries: opengl32

Other linker options:

The SDL, SOIL and GLEW libraries will be installed similarly, after copying the relevant files at the following locations:

The header files -> */CodeBlocks IDE/MinGW/include/*

The lib files -> */CodeBlocks IDE/MinGW/lib/*

The binary files -> */CodeBlocks IDE/MinGW/bin/*

Any other source files, if it's the case will be added to the working project.

Creating the window

In order to create a window and glContext to draw to, I will need to include the following libraries:

```
#include <SDL.h> //this is the SDL header file
#include <glew.h> //this is the OpenGL Extension Wrangler
#include <iostream> //this is standard C++ library for input and output
```

Then I can make use of the following SDL Library functions:

```
SDL_Init(SDL_INIT_EVERYTHING); //This will initialize each component of the
//library

SDL_Window* window = SDL_CreateWindow("License work", SDL_WINDOWPOS_CENTERED,

      SDL_WINDOWPOS_CENTERED, 800, 500, SDL_WINDOW_OPENGL);
```

Knowing that **SDL_CreateWindow()** function returns a pointer of type **SDL_Window** and takes in 6 parameters:

1. A const char* title – consisting into the title of the window;
2. An int x – consisting the position of the window on X axis regarding the Desktop.
3. An int y – consisting the position of the window on Y axis regarding the Desktop;
4. An int w – consisting the width of the window (measured in pixels).
5. An int h – consisting the height of the window (measured in pixels).
6. An uint32 flags – consisting into one of the following flags:

Flag:	**Description:**
SDL_WINDOW_FULLSCREEN	Fullscreen window.
SDL_WINDOW_FULLSCREEN_DESKTOP	Fullscreen window at the current desktop resolution.
SDL_WINDOW_OPENGL	Window optimized for OpenGL context.
SDL_WINDOW_HIDDEN	Window not visible.
SDL_WINDOW_BORDERLESS	Window without borders.
SDL_WINDOW_RESIZABLE	Window which can be resized.
SDL_WINDOW_MINIMIZED	Window which is minimized.
SDL_WINDOW_MAXIMIZED	Window which is maximized.
SDL_WINDOW_INPUT_GRABBED	Window with grabbed input focus. Which can be used for drag&drop.
SDL_WINDOW_ALLOW_HIGHDPI	Window which will be created in high-DPI mode if supported.

If the window is not successfully created, the returned value will be a null pointer. So in order to check for successfulness, I will use the following syntax:

```
if ( window == nullptr )
    {
        cout << "Failed to create window : " << SDL_GetError();
            int a;
            cin >> a; //this will block the cmd from closing.
    }
```

Where the function: **SDL_GetError();** will return a message with the information about the specific error which has occurred.

Further, considering the window creation being successfully, an OpenGL context shall be created in order to be able to draw and render.

```
SDL_GLContext glContext = SDL_GL_CreateContext(window);
GLenum glewResult = glewInit();
```

The **glewInit()** function will initialize the **OpenGL Extension Wrangler** entry points, and if it's successful will return **GLEW_OK**.
The **OpenGL Extension Wrangler** is a library (also named "**glew**") which is used for obtaining information about the supported extensions from the graphics driver. Since there are multiple versions of OpenGL, this will check which fits the used graphics card, and what OpenGL functionality can be used on the client's computer.

If the **glewInit()** returns 0, it means that the extension is not fully compatible with the client's machine, case in which the full functionality of **glew** cannot be used. But to circumvent this situation, a global switch can be turned on by setting **glewExperimental = GL_TRUE;** case which ensures that the valid entry points within the library can be used. In other words, not all the functionality in glew can be used, but only the supported ones.

In order to check whether it's need for **glewExperimental** or not, I will use the following syntax:

```
if (glewResult!= GLEW_OK)
    {
            cout << "Glew Error: " << glewGetErrorString(error) << endl;
                  int a;
                cin >> a; //this will block the cmd from closing.
    }
```

The function: **glewGetErrorString()** – returns a string containing the error value.

Now, imagine that I can create a class for Window and its functionality. In C++ a class consists in 2 files: the **.cpp** file, and the **.h** file. The **.cpp** file contains the constructor and destructor of the class, whereas the **.h** file contains the class declaration with its fields and properties. If I name the class in a suggestive way, I will have: **Window.cpp** and **Window.h** files.

The content of **Window.h** shall be:

```
#include <glew.h>
#include <SDL.h>
#include <iostream>
class Window
{
    public:
        Window(); //the declaration of the constructor;
        virtual ~Window(); //the declaration of the destructor;
        //Getters - the encapsulation of the Window.
        SDL_Window* getWindow()
        {
            return window;
        }
    private:
        SDL_Window* window; // the main window pointer;
        const int screen_Height = 594; //the height of the window;
        const int screen_Width = 1001; //the width of the window;
        SDL_GLContext glContext; //the context of the window.
        GLuint worldTexture; //this hold the identifier for the environment
//texture.
}
```

The content of **Window.cpp** shall be:

```
#include "Window.h"
//The constuctor of the window.
Window::Window() //the Window:: -> specifies that this belongs to Window
class exclusively.
{
    //Initialize the SDL library
    SDL_Init(SDL_INIT_EVERYTHING); //this is going to initialize everything.
    //Creating the window
    window=SDL_CreateWindow("License work", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, screen_Width, screen_Height , SDL_WINDOW_OPENGL);
      if ( window == nullptr )
    {
        cout << "Failed to create window : " << SDL_GetError();
            int a;
            cin >> a; //this will block the cmd from closing.
    }
    //Creating the context for drawing inside the window.
      glContext = SDL_GL_CreateContext(window);
      GLenum error = glewInit();
      if (error != GLEW_OK)
    {
            cout << "Glew Error: " << glewGetErrorString(error) << endl;
                  int a;
                cin >> a; //this will block the cmd from closing.
    }
}

Window::~Window() //the Window:: -> specifies that this belongs to Window
class exclusively.
{
    //dtor
}
```

Considering that I have a main function inside a file named **main.cpp**. I can put all this together and create our window as it follows:

```
#include "Window.h" //inside this header our class was defined.
int main(int args, char** argv) //these 2 arguments are required by the SDL
library
{
    Window windObject; //Creating a window object.
    SDL_Window* window = windObject.getWindow(); //the window created by
constructor described above will be held by this variable.
    while(window != nullptr) {} //just don't let the program terminate while
the user didn't closed the window.
}
```

Creating a Car class

Since a Traffic is composed by car objects, let us define our Car class and describe the behavior of each traffic participant.

Following the same naming convention described already for the Window class, I will have out Car class composed from 2 files: **Car.cpp** and **Car.h**

The content of **Car.h** will be:

```
class Car{
public: Car(); //the main constructor
      Car(const int speed); //the overloaded constructor
      virtual ~Car(); //the destructor
      GLfloat CarCoordinates[8];
      /*
      Since I'm programming using OpenGL API, I need to use common OpenGL
      data types, this is what GLfloat is. Since each 2D car is composed of 4
      vertices, and each vertex has 2 dimensions on X axis and Y axis I will
      have 8 values, 2 for each vertex.
      */
      const int CarHeight; /*this will be the distance from front of the car
until the rear of it, measured in pixels.*/
      const int CarWidth; /*this will be the distance from the left side to
the right side of the car, measured in pixels*/
      const GLuint CarTexture; /*this variable will hold the texture for the
car.*/
      int Speed; /*this variable will hold the speed value through each a car
is travelling, also measured in pixels.*/
      int CarRotatedPosition; /*This will be variable determining the
position in which a car is moving:
      1 – Car is moving UP
      2 – Car is moving RIGHT
      3 – Car is moving DOWN
      4 – Car is moving LEFT*/
      bool isCarMoving; /*this variable will tell whether the car is moving
or not.*/
}
```

Having these fields ready, I need to create a constructor for their initialization. The constructor will be defined by the Car.cpp:

```
#include "Car.h"
Car::Car(const int speed) //This constructor belongs to Car class
{
    CarHeight = 40;
    CarWidth = 20;
    Speed = speed;
    isCarMoving = true;
}
//Overriding the constructor
Car::Car(){//This constructor belongs to Car class as well. Overloading a
constructor is a way of providing sets of constructor which differs in the
type of parameters or the number of it.

}

Car::~Car() //This is the destructor
{
}
```

More functionality will be added to this class as I ride along.

Creating a Route class

Since each Car object has to travel somewhere, I will create a derived class containing all the Car's functionality, and extending its barriers.

What will this class provide to us?

- A route for each car object to follow.
- A function which will handle the car traveling;
- Read-only properties for the Car **private/protected** fields.

The read-only properties of a class are often called "**Getters**" whereas the write-only properties are called "**Setters**", this two properties make the core of **Encapsulation**.

Later on, additional functionality might be added: collision detection, driving priority checking.

The **Route.h** file shall contain the class declaration as it follows:

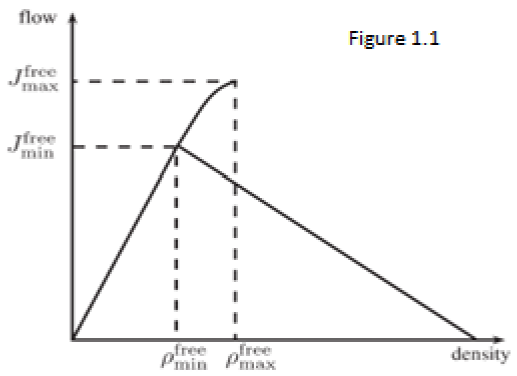
```
#include "Car.h" //included in order for the Car class to be found.

class Route: public Car
{
public:
        Route(const int Traveling_speed); //Constructor
        virtual ~Route(); //Destructor
}
A constructor is a special type of subroutine called to create an object. It
prepares the new object for use, often accepting arguments that the
constructor uses to set required member variables. The constructor is called
when the object is created, whereas the destructor does the opposite thing
and gets called when the object is deleted.
```

The **Route.cpp** file which contain the constructor and destructor will be:

```
#include "Route.h"
Route::Route(const int Traveling_speed): Car(Traveling_speed) //This will
call the base class constructor first
{
    //ctor
}
Route::~Route()
{
    //dtor
}
```

Moving the Car

Traffic flow which each vehicle constitutes present a variety of phenomena: from dynamic phase transitions over self-organization to the formation of shock waves. So there is a large interest in understanding the relation between the traffic flow and the vehicle density. The diagram below is showing the flow-density relation:



Where:
J = the traffic flow (measured in vehicles per hour and lane);
P = vehicle density (measured in vehicle per kilometer)

For low vehicle density: $\rho > \rho^{\text{free}}_{\text{max}}$ vehicle can travel at their desired speed.
The functional relation between the two can be expressed as it follow: $J = v\rho.$

Heading back to programming this whole thing, I will have to add a Route to follow for a Car object. In order to do that, I will define a function name **RouteNr1()** inside **Route.h** as **private**. The function will be called by a **public** travelling function named **FollowingRoute()** which will ensure that the car is traveling on each frame.

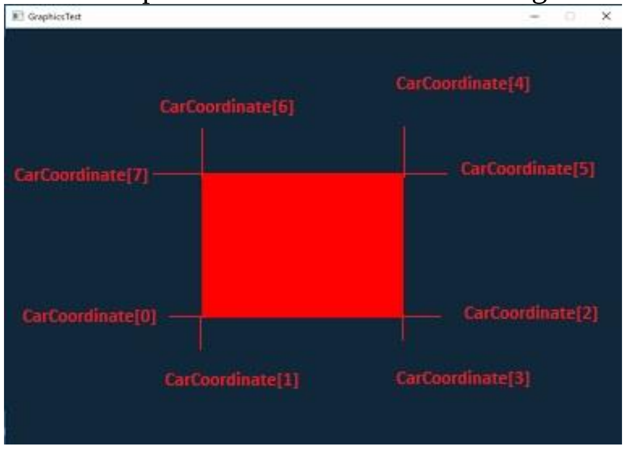
So for the definition and declaration, I will have:

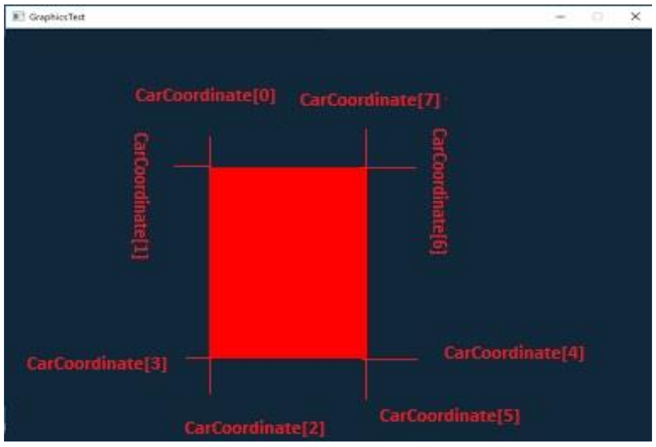
```
private:
    const int routeNumber; //this variable will hold the value of the route
to be followed.
    void RouteNr1() //this will ensure that the car is always on the Route
    {
    //The magic of the track1 happens here:
    if( CarCoordinates[1] < 120 ) //if the car Vertex1 Y coordinate is less
//than 120 pixels which is the value of Curve1 Y coordinate:
    {
        MovingCarUp(); //Let the car travel UP direction
    }
    else //it means I need to turn left, because I reached the curve.
    {
      if(CarCoordinates[2] < 570) // if car Vertex2 X coordinate is //less
      than 570 pixels which is the value of Curve2 X coordinate:
      {
         RotateCarHorizontally(); //Turn the car from vertical //position to
         Horizontal position
         CarRotatedPosition = 2; //Set this to 2, because the car //is now
         traveling LEFT direction
         MovingCarRight(); //Move the car on LEFT
      } else { // Enter here if it’s already on Curve2
          if(CarCoordinates[1] < 640) //while is not out from the //screen
          {
             RotateCarVertically();//Turn the car from vertical //position to
Vertical position
             CarRotatedPosition = 1; //Set this to 1, because the //car is
now traveling UP direction
             MovingCarUp(); //Move UP
          } } } }
```

How will the traffic flow work?

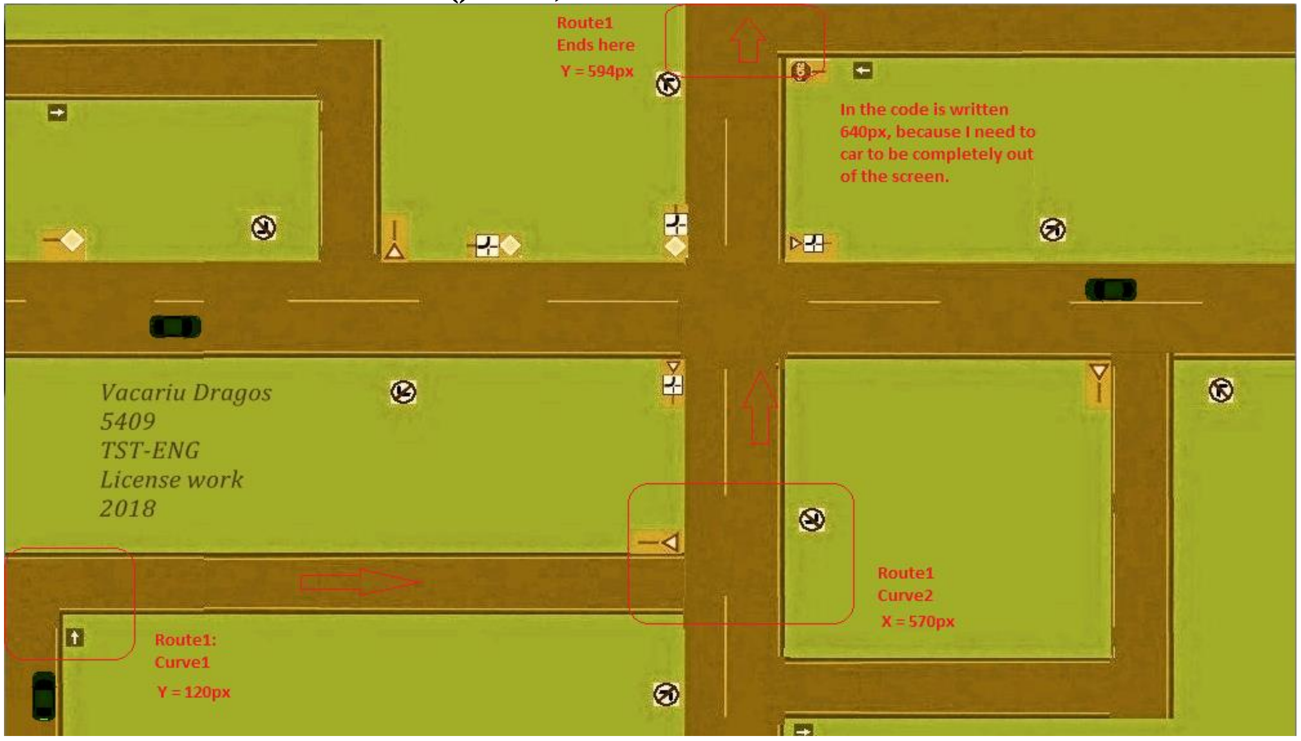
Since each car object consists in a rectangle with 8 coordinates, the car will move LEFT when all the 4 X coordinates will be increased, it will move RIGHT when all the X's are decreased. Similarly, it will move UP when all the 4 Y coordinates will be increased, and it will move DOWN when all Y's are decreased. In order to increment or decrement those coordinates, I use the speed variable which hold a number of pixels.

In order to turn the car around I will have to redraw the rectangle from vertical position to horizontal position as it is in the following schematic:





Since we calculated the **RouteNr1()** above, here is the relevant schematic for it:



In a similar manner, it was designed any Route on the track:

In order for this Route to be valid, I need to set up the following:

- The Car Coordinates need to be set on the Right Bottom corner of the screen;
- The Car needs to be Rotated Vertically;
- The Car needs to be Textured facing the Top of the screen;
- The function which will move the car on each frame needs to be designed.

The first free points mentioned above, shall be done into the Route class constructor, as it follows:

```
Route::Route(const int Traveling_speed):Car(Traveling_speed) //This will call
the base class constructor first
{
routeNumber = 1;
switch(routeNumber)
{
   case 1:{
      //Setting the car on the buttom corner of the screen. Since
//I call the RotateCarVertically() function below, the other coordinates will
be set there using the CarHeight and CarWidth constants.
       CarCoordinates[0] = 20.1; //the x coordinate of first vertex
       CarCoordinates[1] = 14.3; //the y coordinate of first vertex
       CarCoordinates[2] = 20.1; //the x coordinate of second vertex
       CarCoordinates[7] = 14.3; //the y coordinate of fourth vertex
       CarRotatedPosition = 1; // 1. means heading to the top of the screen.
//CarRotatedPosition will help us settle the Texture of the car: facing top
or bottom of the //screen
       RotateCarVertically(); //Making sure the car is placed vertically.
            break;
        }
      default:
      {
            /*it will never get in here.*/
      }
}
```

The value of **CarWidth** which sets the width of the Car object is set to 20px, since the road on the map has a maximum width of 80px, and only one lane per sense. In real world a lane has a width of about 3 meters, with enough room for a large size car to pass, usually a truck has 1.5 the width of a regular car, which is why I chose 20px for each car object.

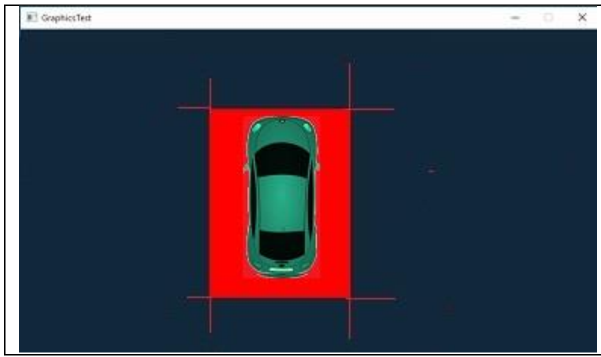
The value of **CarHeight** which is the variable to store the length of the car object is about 40px, because in real world a car shall fit on a lane even when it's placed perpendicularly.
The **RotateCarVertically()** function as well as **RotateCarHorizontally()** function since it has Car functionality within it, it will be defined into the Car class, as a **protected** property.

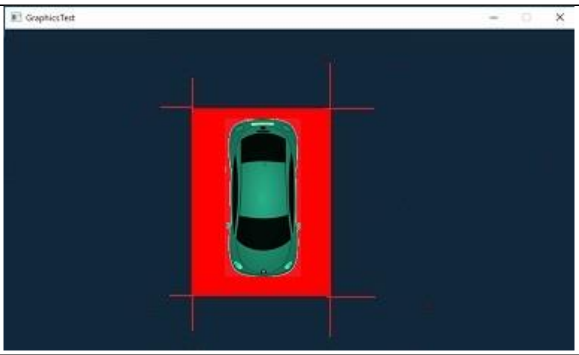
A protected property means it will be inherited by the Route class:

Inside the file **Car.h**, I will have added the following:

```
class Car
{   /*The old functionality is here*/
```

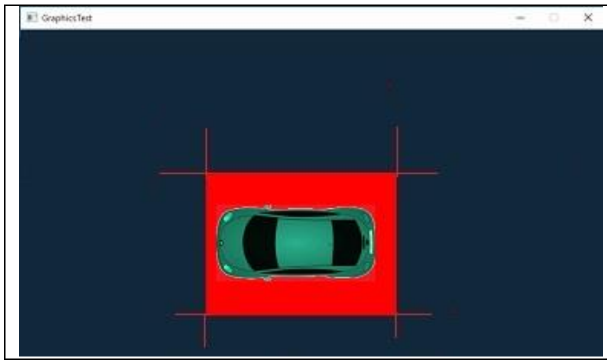
```
   protected:
   void RotateCarVertically()
   {
/*A car will be made of a rectangle, each rectangle has 4 points, each pair
of 2 points has same X coordinate. Since I set the first vertex and the X for
the second vertex the Y of the second vertex will be obtained from the Y of
the first vertex + CarHeight.*/
      CarCoordinates[3] = CarCoordinates[1] + CarHeight;
      CarCoordinates[4] = CarCoordinates[2] + CarWidth;
      CarCoordinates[5] = CarCoordinates[1] + CarHeight;
      CarCoordinates[6] = CarCoordinates[2] + CarWidth;
   }
```

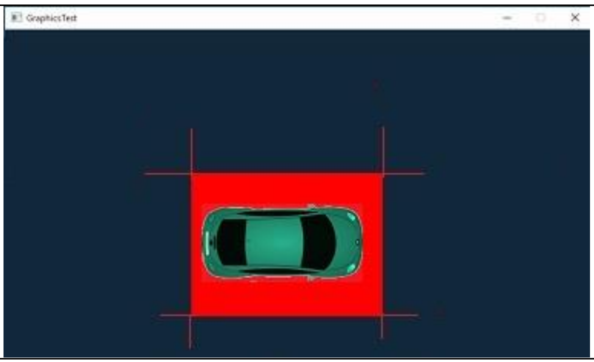




Regardless of the side which the car is facing: either top of the screen or bottom of the screen, the same rectangle has to be drawn. So the texture is going to fit in accordingly, this is going to happen in Draw() method of the car object.

```
   void RotateCarHorizontally()
   {
      CarCoordinates[3] = CarCoordinates[1] + CarWidth;
      CarCoordinates[4] = CarCoordinates[2] + CarHeight;
      CarCoordinates[5] = CarCoordinates[1] + CarWidth;
      CarCoordinates[6] = CarCoordinates[2] + CarHeight;
   }
```





Similarly like in the function above, the same rectangle will be drawn when the car is facing left side of the screen, or right side of the screen.

```
/*The car moving functions:*/
   void MovingCarUp()
   {
       if(isCarMoving)
       {   //Increasing only the Y coordinate of each vertex.
        CarCoordinates[1]+=Speed;
```

```
         CarCoordinates[3]+=Speed;
         CarCoordinates[5]+=Speed;
         CarCoordinates[7]+=Speed;
        }
   }
   void MovingCarDown()
   {
      if(isCarMoving)
      {  //Decreasing only the Y coordinate of each vertex.
         CarCoordinates[1]-=Speed;
         CarCoordinates[3]-=Speed;
         CarCoordinates[5]-=Speed;
         CarCoordinates[7]-=Speed;
      }
    }
    void MovingCarLeft()
    {
      if(isCarMoving)
     {  //Decreasing only the X coordinate of each vertex.
         CarCoordinates[0]-=Speed;
         CarCoordinates[2]-=Speed;
         CarCoordinates[4]-=Speed;
         CarCoordinates[6]-=Speed;
      }
    }
   void MovingCarRight()
   {
       if (isCarMoving)
       {    //Increasing only the Y coordinate of each vertex.
             CarCoordinates[0]+=Speed;
             CarCoordinates[2]+=Speed;
             CarCoordinates[4]+=Speed;
             CarCoordinates[6]+=Speed;
       }
   }
}
```

Now to actual make the car move I will have the following function defined in the Route class, inside file **Route.h**:

```
#include "Car.h"
class Route: public Car
{     public:
      void FollowingRoute()
        {
            switch(routeNumber){
                case 1:{ RouteNr1();  break;}
                default: {/*It will never reach here.*/}
      }
}
```

Drawing to the screen

In order to draw to the glContext inside the Window, I will have to add some functionality to Window class and Car class.

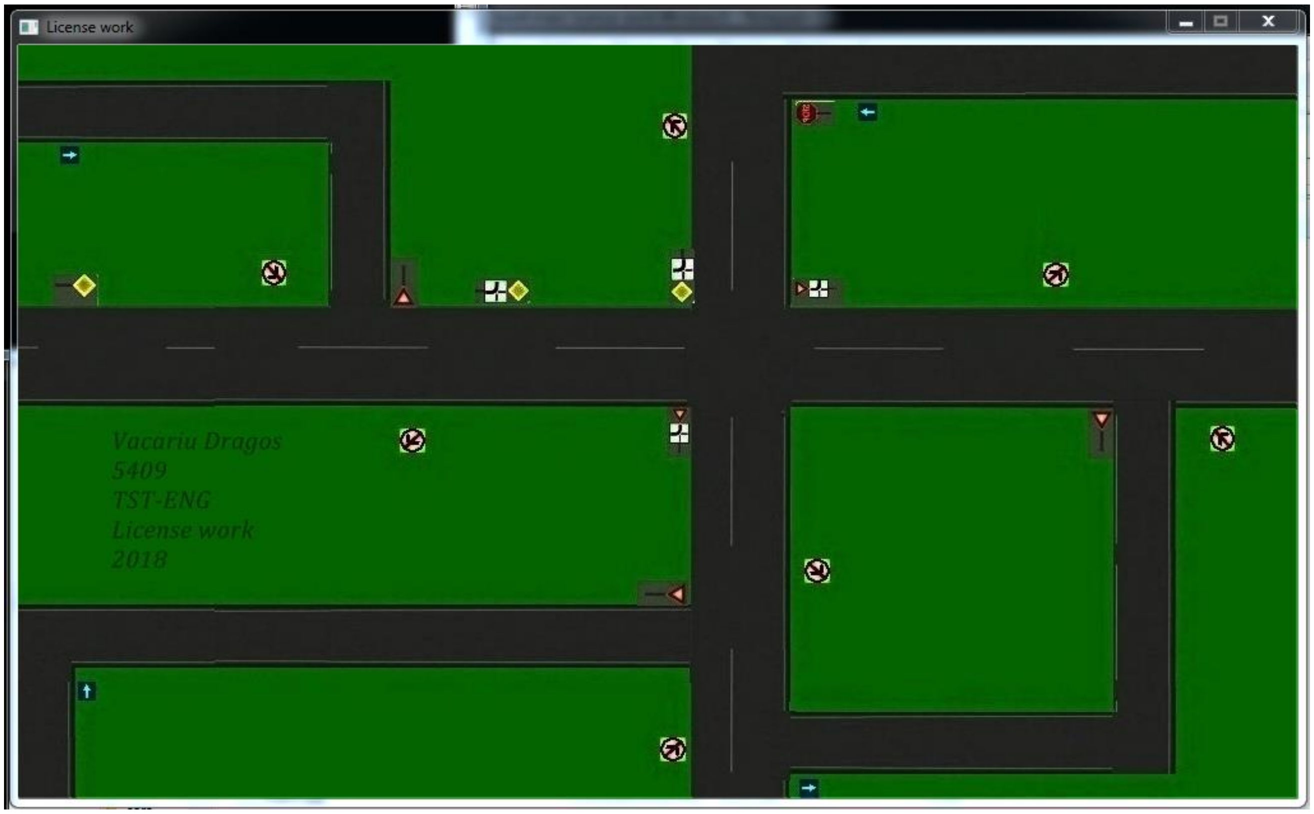
The function which will setup the **Window** and the **glContext** for the other drawings will be the following:

```
#include "Car.h"
using namespace std;
class Window
{
    public:
      void DoTheRendering()
        {
             //translating the normalized coordinates into pixels
            glViewport(0, 0, screen_Width, screen_Height);
            glMatrixMode(GL_PROJECTION);
            glLoadIdentity();
            glOrtho(0, screen_Width, 0, screen_Height,

                  /*These 2 coordinates are for 3D: Z axis, which need to be
            at least 1 unit even for 2D drawing*/

                    -100, 100);
            glMatrixMode(GL_MODELVIEW);
            glLoadIdentity();
            //translation done.
/*The OpenGL System of Coordinates has values between range -1 and +1 for
both X and Y axis, the translation consists in mapping those coordinates in
pixels between 0 and whatever the height and width of the Window is.*/

            //Setting the environment for texturing:
            glEnable(GL_TEXTURE_2D); //enable 2D texturing
            glEnable(GL_BLEND); //enable Blending;
            glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); //using
transperancy.

            //Let's draw the content:
            glClear(GL_COLOR_BUFFER_BIT); //clear the screen each time,
before drawing;
            glBindTexture(GL_TEXTURE_2D, worldTexture); //use this texture
            glBegin(GL_QUADS); //draw a rectangle.
            glTexCoord2f(0.0, 0.0);//the bottom right corner of the
//rectangle is the bottom right corner of the texture;
            glVertex2f(0.0, 0.0); //set the bottom right corner of the
//rectangle on the bottom right corner of the Window;
            glTexCoord2f(0.0, 1.0); //the bottom left corner of the
//rectangle is the bottom left corner of the texture;
            glVertex2f(0.0, screen_Height); //set the bottom left corner of
//the rectangle on the bottom left corner of the Window;
            glTexCoord2f(1.0, 1.0); //the top left corner of the rectangle is
//the top left corner of the texture;
```

```
            glVertex2f(screen_Width, screen_Height); //set the top left
//corner of the rectangle on the top left corner of the Window;
            glTexCoord2f(1.0, 0.0); //the top right corner of the rectangle
//is the top right corner of the texture;
            glVertex2f(screen_Width, 0.0); //set the top right corner of the
//rectangle on the top right corner of the Window;

            glEnd(); //at this moment, the vertices are set, and a rectangle
//with window size is created and textured inside the window.
        }
```

In geometry as well as in OpenGL a vertex is a point in which two or more lines can meet. In OpenGL while drawing Quads which means Rectangles, after placing the 4 vertices the lines get automatically drawn and the figure gets solid.

Each 3D model gets composed of vertices, and more likely from unified triangles, because a base rule notes that: any other geometric figure can be obtained from triangles.
NOTE: The more vertices a figure has the more rounded it looks.
After calling **DoTheRendering()** function into the main loop of **int main()** the result should be the following:



In order to draw car object on the screen, I also need to set a Drawing function for that object, the Drawing function shall be created inside the **Car** class, and it should be **public** since it will be called from outside the class.

```
class Car
{ public:
      /*The old functionality is here*/
```

```
void Draw(int DelayDecreaser) //this DelayDecreaser parameter will be
used for the rendering speed.
 {
    DelayDecreaser-=1;
    glBindTexture(GL_TEXTURE_2D, this->CarTexture);
   switch(CarRotatedPosition)
  {
      case 1:{/*Texture the car facing the UP direction.*/
           glBegin(GL_QUADS);
           glTexCoord2f(0.0, 0.0);
           glVertex2f(CarCoordinates[0], CarCoordinates[1]);
           glTexCoord2f(0.0, 1.0);
           glVertex2f(CarCoordinates[2], CarCoordinates[3]);
           glTexCoord2f(1.0, 1.0);
           glVertex2f(CarCoordinates[4], CarCoordinates[5]);
           glTexCoord2f(1.0, 0.0);
           glVertex2f(CarCoordinates[6], CarCoordinates[7]);
           glEnd();
           break;
      }case 2: {
        /*Texture the car facing the RIGHT direction.*/
           glBegin(GL_QUADS);
           glTexCoord2f(1.0, 0.0);
           glVertex2f(CarCoordinates[0], CarCoordinates[1]);
           glTexCoord2f(0.0, 0.0);
           glVertex2f(CarCoordinates[2], CarCoordinates[3]);
           glTexCoord2f(0.0, 1.0);
           glVertex2f(CarCoordinates[4], CarCoordinates[5]);
           glTexCoord2f(1.0, 1.0);
           glVertex2f(CarCoordinates[6], CarCoordinates[7]);
           glEnd();
           break;
      }case 3: {
        /*Texture the car facing the DOWN direction.*/
           glBegin(GL_QUADS);
           glTexCoord2f(1.0, 1.0);
           glVertex2f(CarCoordinates[0], CarCoordinates[1]);
           glTexCoord2f(1.0, 0.0);
           glVertex2f(CarCoordinates[2], CarCoordinates[3]);
           glTexCoord2f(0.0, 0.0);
           glVertex2f(CarCoordinates[4], CarCoordinates[5]);
           glTexCoord2f(0.0, 1.0);
           glVertex2f(CarCoordinates[6], CarCoordinates[7]);
           glEnd();
           break;
      }case 4:{
        /*Texture the car facing the LEFT direction.*/
           glBegin(GL_QUADS);
           glTexCoord2f(0.0, 1.0);
           glVertex2f(CarCoordinates[0], CarCoordinates[1]);
           glTexCoord2f(1.0, 1.0);
           glVertex2f(CarCoordinates[2], CarCoordinates[3]);
           glTexCoord2f(1.0, 0.0);
           glVertex2f(CarCoordinates[4], CarCoordinates[5]);
           glTexCoord2f(0.0, 0.0);
           glVertex2f(CarCoordinates[6], CarCoordinates[7]);
           glEnd();
```

```
                break;
            }
        }
        SDL_Delay( 20 - (DelayDecreaser*2) );
}
```

Note that we set an **SDL_Delay()** which is a function which will delay the rendering by 20ms – doubled the **DelayDecreaser** value. The purpose of this function is to maintain the same rendering speed regardless of how many Car object will have to draw. That is because mainly the more cars to draw means more work to do by the GPU, which provides a time gap between single car and multiple car drawings.
The **glBindTexture()** function will send the texture through the rendering pipeline.
The switch statement within the function will ensure that the texture is properly arranged according to the direction the car is facing.

For example: when the car is facing the top of the screen we have the vertical rectangle as when it is facing the bottom of the screen, so the texture coordinate shall be shifted from the top vertices to the bottom vertices.

The drawing functions and the function which move the car on the tack will be called inside the main loop in **int main():**

```
#include "Window.h" //inside this header our class was defined.
#include "Route.h"
int main(int args, char** argv)
{
    Window windObject; //Creating a window object.
    SDL_Window* window = windObject.getWindow(); //the window created by
constructor described above will be held by this variable.
    Route car(2);
    while(window != nullptr) //while the window is opened.
      {
      /*This is the main loop. This will keep the things on going.*/
            windObject.DoTheRendering(); //Render the window
            car.Draw(); //Render the car
            car.FollowingRoute(); //Move the car
            //Check if the car shall die and be recreated
            if( car.getCarCoordinates(2) > 1000 || car.getCarCoordinates(1) >
      600 || car.getCarCoordinates(2) < 0 || car.getCarCoordinates(1) < 0 )
            {
                  /*This will happen after the car leaves the screen.*/
                car = Car(2);
            }
            SDL_GL_SwapWindow(window);
      } //just don't let the program terminate.
}
```

Now, what happens at this point is that I have a car moving the **RouteNr1(),** and once it exists the window the car get respawned at the starting position, and keeps riding the same.

Texturing objects

Since I didn't mentioned everything about texturing an object, now it's the time. In order to texture a rectangle, which is what the entire project consist in, I use the following function:

```
GLuint loadTexture(const char* filename)
{
     int width, height; //this will store the dimensions of the image.
     unsigned char* image = SOIL_load_image(filename, &width, &height,
                                  0, SOIL_LOAD_RGBA);
//this function will load the image, using RGBA channels.
     if(image==nullptr) //if the image was not found.
     {
          cerr << "Failed to load the texture" << " " << filename << endl;
          cout << "Please check the integrity of the project." << endl;
           exit(1); //exit the program at this moment.
     }
     GLuint textureid; //this will store the texture identifier.
     glGenTextures(1, &textureid); //generate a texture.
     glBindTexture(GL_TEXTURE_2D, textureid); //the texture will be 2D
     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
       GL_RGBA, GL_UNSIGNED_BYTE, image); //the texture will contain the
image.
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
//GL_TEXTURE_MIN_FILTER is used whenever the pixel being textured maps to an
area greater than one texture element.
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
//GL_TEXTURE_MAG_FILTER  is used when the pixel being textured maps to an area
less than or equal to one texture element
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
      GL_MIRRORED_REPEAT); //let the texture be mirrored when it's the case.
     SOIL_free_image_data(image); //delete the image.
     return textureid; //return the texture identifier.
}
```

This function will be defined in a common place: let's say: **Common.h.** Alternatively it can be defined inside a class with static modifier which will allow it to be called without an instance.

In order to load the texture for the environment and for the Car objects, I introduced the following function calls into **Window** constructor and **Car** constructor.

```
#include <SOIL.h> //since I use SOIL_load_image() to load images, I need to
add this header in each class as well as adding the source code included
within the development library to my project.
#include "Common.h" //This is where I defined loadTexture() function
Car::Car(const int speed)
{
      /*The old functionality is here*/
      string filepath = "textures/cars/"; // the relative filepath to the
resource.
      CarTexture = loadTexture( filepath.c_str() +  "car.pngs");
}
Window::Window()
{
      /*The old functionality is here*/
      worldTexture = loadTexture("textures/environment/environment00.gfx");
}
```

Adding these resources to the project creates a dependency between the executable file and the resource. The executable file has to be placed in the project root directory in order to find the resources used for texturing.

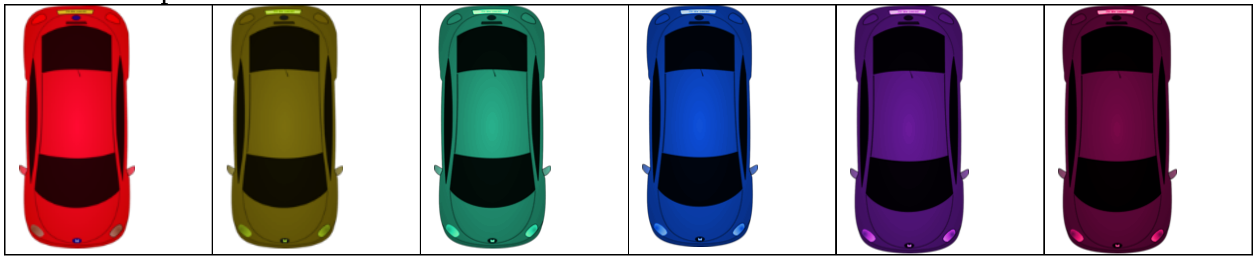
Random Car Texture and World Time Texture Change

Since traffic flow is created by different kind of vehicles, here I will present how to randomly change the Car Textures, in order to provide a more traffic realistic view.

Random changes on car texturing shall be provided. To do that I will use the following function in define inside **Car.h**:

```
class Car
{
/*The old functionality is here*/
protected:
      void CarTextureSelector()
      {
         const string textures[14] = {"car.pngs", "car2.pngs", "car3.pngs",
      "car4.pngs", "car5.pngs", "car6.pngs", "car7.pngs", car8.pngs",
      "car9.pngs", "car10.pngs", "car11.pngs", "car12.pngs", "car13.pngs",
      "car14.pngs"};
         int selector = (rand()% 14); //provide a random choice from 0 to 13.
         string filepath = "textures/cars/"; //the relative filepath to the
      resource.
         filepath += textures[selector]; //append the filepath to the chosen
      texture.
         CarTexture = loadTexture( filepath.c_str() ); // load the texture;
      }
}
```

In the string textures were mentioned the filenames of the textures used by cars. In the figure below it is presented how these textures look like:



In order for the **CarTextureSelector()** to do its job, it has to be called right in the moment when creating a car object. That would happen inside the constructor of Car class:

```
Car::Car(int speed)
{     /*The old functionality is here:*/
      CarTextureSelector();
}
```

At this moment when creating a car object, a random texture will be chosen from the list.

Time is relative from real world to virtual world but still is never a constant. So how to provide Time Change in the Virtual Environment.

What is the virtual environment?

A virtual environment is a computer-based simulation which may be populated an explored. In many virtual worlds the time is passing faster than in the real world, in order to provide a faster progress.

In order to provide time passage to traffic virtual environment, I had to find a way to measure real world time, which I found below:

In **Window.h** I created the following functions:

```
#include <time.h>
#include <SOIL.h>
class Window
{
    public:
      /*The old content describe above is here*/
      bool getTimePassed()
      {
          int amountOfSecondsPassed = difftime( time(0), start_time);
/*The function above will measure the difference of time between last time
when the function was called and now.*/
          if(amountOfSecondsPassed > 1) //if only a second passed.
          {
              start_time = time(0); //store the current time;
              the_time[1]+=1; //increase minutes in my virtual world
              if(the_time[1]>59) //check if I have 60 minutes
              {
                 the_time[1]=0; //reset the minutes to 0;
                 the_time[0]++; //increase the hours
                 if(the_time[0]>23) //if a day has passed;
                 {   the_time[0]=0; //reset the hours;    }
                setBackgroundTexture();
               }
               return true;
           }
           return false;
      }
     int getTime(char c)
     {
         switch (c) {
             case 'm':
             case 'M': { return (int)the_time[1]; //return minutes; }
             case 'h':
             case 'H': {return (int)the_time[0]; //return hours;  }
         }
     }
  private:
/*The old content describe above is here*/
      char the_time[2]; //this will store hours an minutes in virtual world;
      time_t start_time; //this will store the current time in real world;
}
```

The **setBackgroundTexture()** function will be defined later, and it will provide changes to the environment texture in relation with time.

After finding a way to measure time, a virtual clock can be created using the **rand()** method **the_time** field and **getTime()** property.
Having the 2 functions and the 2 fields set, I will alter the constructor within **Window.cpp** file as it follows:

```
Window::Window()
{
/*The old content describe above is here*/
    start_time = time(0);
    srand(time(0));
    the_time[0] = rand()%24; //the time will be randomized
    the_time[1] = rand()%60;
    setBackgroundTexture();
}
```

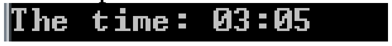
At this moment when the window Object is created a random Day Time get created which will be modified each second by the **getTimePassed()** method. The difference between the real-world time and the virtual-world time is 60:1 which means 60 seconds in real world is an hour in the virtual world.

In order to change the environment texture, the definition of **setBackgroundTexture()** shall be provided inside **Window.h** and that will be the following:

```
void setBackgroundTexture()
{
  switch(the_time[0])
  {
    case 0:
    case 1: {
     worldTexture = loadTexture("textures/environment/environment00.gfx");
      break;
    }case 2:
    case 3: {
      worldTexture = loadTexture("textures/environment/environment01.gfx");
       break;
    }case 4:
    case 5: {
      worldTexture = loadTexture("textures/environment/environment02.gfx");
      break;
    }     /*And so on for each of the 2 hours passed in the environment*/
  }
}
```

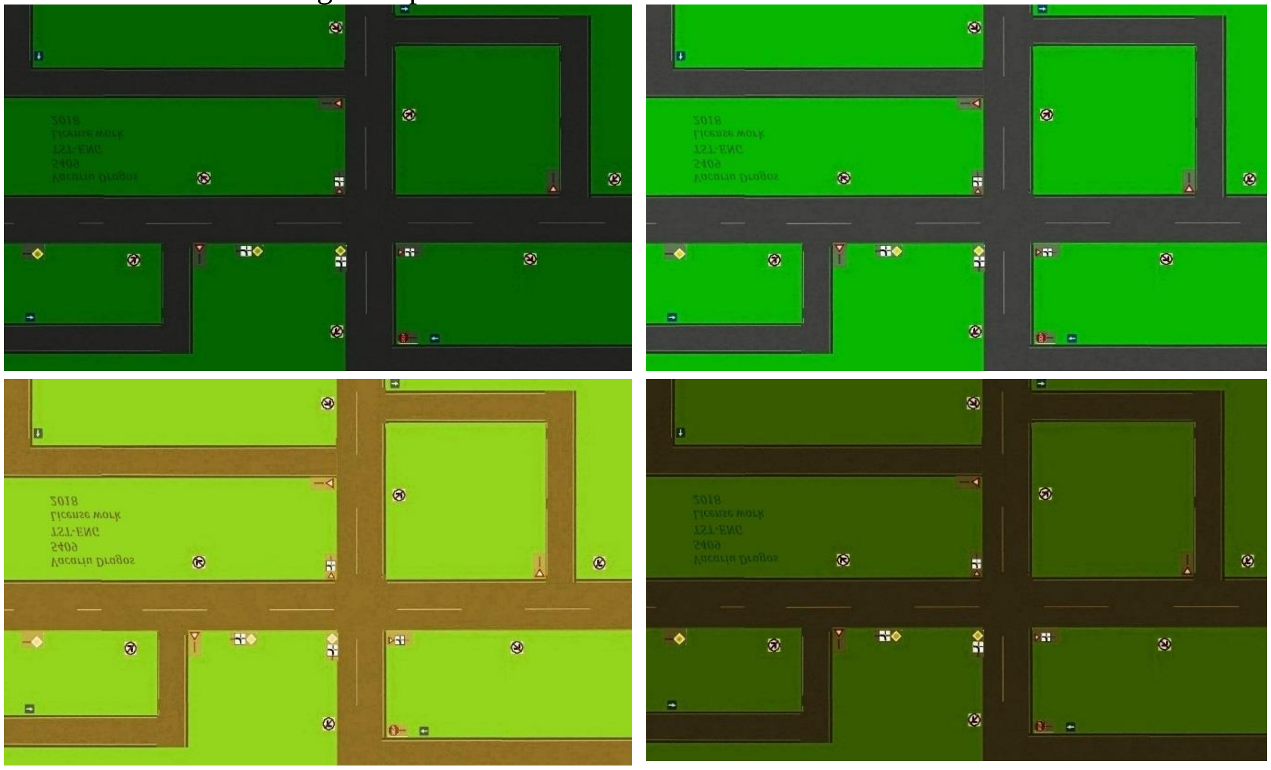
In the main loop of the main function the **getTimePassed()** and the **getTime()** methods will be called:

```
using namespace std;
int main(int args, char** argv)
{ /*The old functionality is here*/
  while(window != nullptr){
    /**The old functionality is also here
    SDL_GL_SwapWindow(window); //where window is SDL_Window*
    /*Just after swapping the buffers*/
    if(windObject.getTimePassed()){
       cout << "The time: ";
       if( windObject.getTime('h') < 10) {//Where a is object of time Window;
           cout << "0" << windObject.getTime('h');
        }else{
           cout << windObject.getTime('h');
        }if(windObject.getTime('m') < 10) {
           cout << ":0" << windObject.getTime('m');
        }else{
           cout << ":" << windObject.getTime('m') << endl;
        }
    }
}
```
At this point the virtual world clock printed in the console, as it follows:



```
The time: 03:05
```

The immersion which is the perception of being physically present in a non-physically world is now created by the time passage rate: 1min = 1h, as well as the 2D graphic design.
The environment will change accordingly with the time passage creating Day to Night changes which can be seen amongst the pictures below:



Since Day Time gets constructed within the Window constructor, it can now be used for various actions. For example: In order to get traffic accordingly to the virtual time, a function can be created to distribute the car objects properly.

Inside **main.cpp** file the following function shall be designed:
```
int getRealisticTraffic(const int hours){
    srand (time(NULL)); //Creating a random generator seeded by time.
    int nrOfCars;
    switch(hours)
    {
        case 0: case 1: case 2: case 3: case 4: {
            nrOfCars = (rand()%3 )+ 1;
            break;
        }case 5: case 6: {
            nrOfCars = (rand()%5 )+ 1;
            break;
        }case 7: case 8: case 9: case 10: {
           nrOfCars = (rand()%6 )+ 1;
           break;
        }case 11: case 12: case 13: case 14: {
            nrOfCars = (rand()%5 )+ 1;
            break;
        }case 15: case 16: case 17: {
            nrOfCars = (rand()%6 )+ 1;
            break;
        }case 18: case 19: case 20:{
            nrOfCars = (rand()%4 )+ 1;
            break;
        }case 21: case 22: case 23: {
            nrOfCars = (rand()%3 )+ 1;
            break;
        }
    }
    return nrOfCars;
}
```

Populating the map

Considering that more track have been created by the same model presented at section "**Moving the car**" the next phase would be how to populate the environment. In order to do that a vector of type Route will be created and the main function will be updated as it follows:

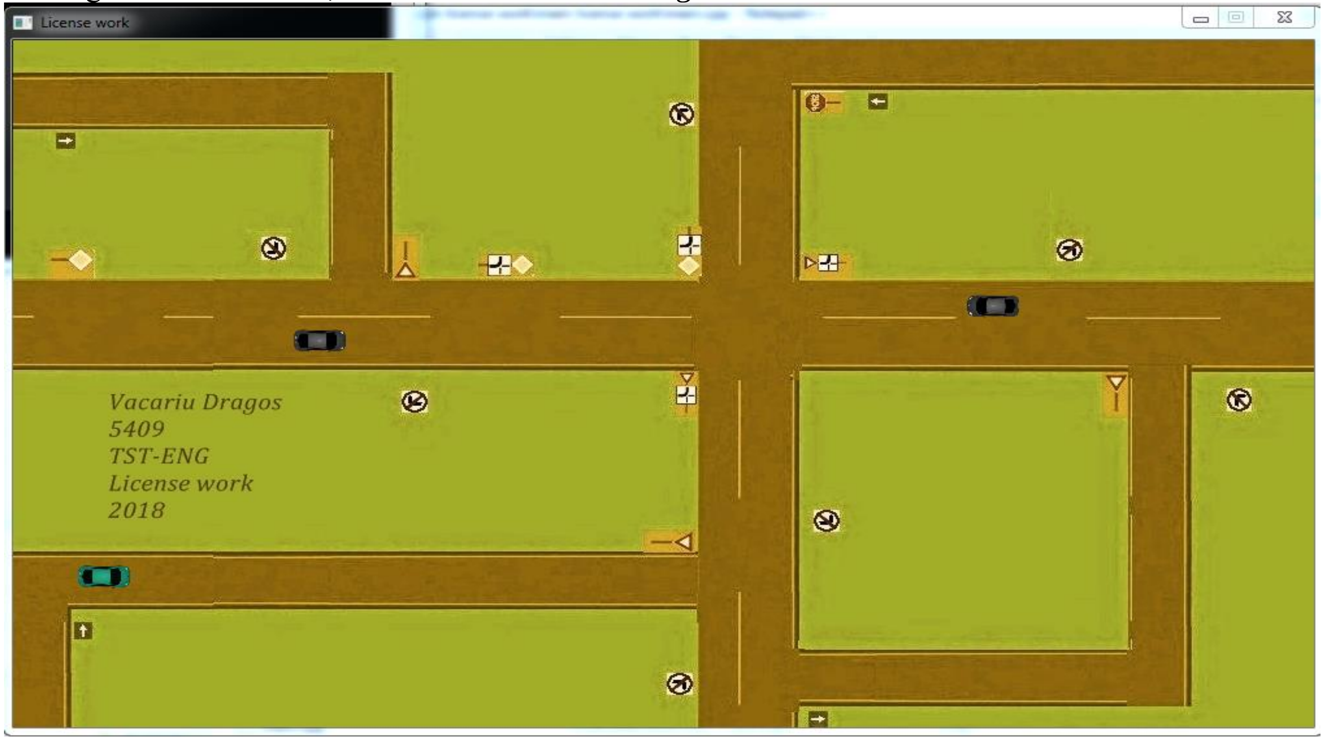
```
#include "Route.h"
#include "Window.h"
#include <time.h>
#include <vector>
//Prototyping the functions:
int getRealisticTraffic(const int hours);
int main(int args, char** argv)
{
    Window windObject; //object of type Window;
    SDL_Window* window = a.getWindow(); //Calling the constructor which 
creates the window
    vector<Route> cars;
    int population = getRealisticTraffic( windObject.getTime('h') );
   /*As you note getRealisticTraffic () gets called to provide traffic 
according with the time which was set in the Window constructor*/
    for(int i=0; i < population; i++){
        cars.push_back( Route(2, i+1) );
    }
while(window != nullptr)//while the window is opened.
    {
         windObject.DoTheRendering();
        for(int i=0; i < cars.size(); i++)
        {
            cars[i].Draw( cars.size() ); //draw each car
            cars[i].FollowingRoute(); //move each car
            if( cars[i].getCarCoordinates(2) > 1000 || 
cars[i].getCarCoordinates(1) > 600 || cars[i].getCarCoordinates(2) < 0 || 
cars[i].getCarCoordinates(1) < 0 ){
            /*Entered here, it means the cars got out from the screen.*/
                cars.erase( cars.begin() + i); // remove it from the vector
            }
        }
        //Check whether it's time to add a new car to the vector:
```

```
        if( (rand()%100 ) == 5 && (cars.size() < 7) ) //this will be done
randomly {
            cars.push_back(Route(2)); //create a new car
        }
        SDL_GL_SwapWindow(window); //Put everything on the screen
    if(windObject.getTimePassed()){
       cout << "The time: ";
       if( windObject.getTime('h') < 10) {
           cout << "0" << windObject.getTime('h');
        }else{
           cout << windObject.getTime('h');
        }if(windObject.getTime('m') < 10) {
           cout << ":0" << windObject.getTime('m');
        }else{
           cout << ":" << windObject.getTime('m') << endl;
        }
    }
  }
}
```

What happens to a car after it gets pushed back from the vector?

The garbage collector is the computer automatic form of memory management. When an object is no longer used in a program the garbage collector attempts to reclaim the used memory from the object, amongst this process the destructor gets called. In a language with automatic garbage collection mechanism it is difficult to ensure that the destructor is even invoked, in such languages unlinking an object from existing resources is done explicitly from calling the appropriate function.

Using the **getRealisticTraffic()** method the environment has been populated, and each car is moving on it's own route, as it's skatched in the image below:



The getter used to access the protected field **CarCoordinates** will be **define** as public property in **Route** class as it follows:

In file **Route.h**:
```
#include "Car.h"
class Route: public Car
{
    public: /*The old functionality is here*/
      GLfloat getCarCoordinates(const int index) //This will make the
//CarCoordinate field be read-only outside this class.
      {
        if(index < 0 || index > 7) //checking for the boundaries of the array
        {
          cout << "Wrong value had been entered for reading the Car
                                Coordinate." << endl;
                exit(1);
        }
            return CarCoordinates[index];
     }
};
```

Collision Detection

To avoid collision for the cars which are following routes which get across, functions which will control the **isCarMoving** field of Car class shall be created inside the **Car.h**:
```
class Car
{ /*The old functionality is here*/
   protected:
   void StopTheCar(){
       isCarMoving = false; //this will stop the car from moving, since
        //moving is done only when this flag is true
   }
   void StartTheCar(){
       isCarMoving = true; //this will set the car back on moving
   }
};
```

In a VANET network vehicles to vehicles and vehicles to roadside unit communications will save several lives and forestall injuries. According to several applications, if a vehicle reduces its speed considerably once identifying an accident then vehicle broadcast its location to its neighbor vehicles. And different receivers can try to transfer the message to the vehicles further behind them and therefore the vehicle in question can emit some alarm to its vehicles and different vehicles behind. During this process, a lot of vehicles way behind can get an alarm signal before they see the accident and may take any better decision.

In code, in order to detect several collisions, the following function was designed:

```
void CollisionDetection(vector <Route> cars)
{
   for(unsigned int i=0; i<cars.size(); i++)
   {
     if( uniqueID != cars[i].uniqueID)
     {
     //Perpendicular collision
        if(  (getCarCoordinates(7) < cars[i].getCarCoordinates(7) + 50 &&
     getCarCoordinates(7) > cars[i].getCarCoordinates(7) - 50)
    && ( getCarCoordinates(6) > cars[i].getCarCoordinates(0) - 50
    && getCarCoordinates(6) < cars[i].getCarCoordinates(0) + 50)
    && ( getCarCoordinates(6) > cars[i].getCarCoordinates(5) - 50
    && getCarCoordinates(6) < cars[i].getCarCoordinates(5) + 50)
    && CarRotatedPosition%2==1  && cars[i].CarRotatedPosition%2==0)
    {
       if(isCarMoving && cars[i].isCarMoving)
       {
          StopTheCar();
       }
    }
    else if(  (getCarCoordinates(6) < cars[i].getCarCoordinates(6) + 50
     && getCarCoordinates(6) > cars[i].getCarCoordinates(6) - 50)
     && ( getCarCoordinates(7) > cars[i].getCarCoordinates(7) - 50
     && getCarCoordinates(7) < cars[i].getCarCoordinates(7) + 50)
     && ( getCarCoordinates(7) > cars[i].getCarCoordinates(0) - 50
     && getCarCoordinates(7) < cars[i].getCarCoordinates(0) + 50)
     && CarRotatedPosition%2==0 && cars[i].CarRotatedPosition%2==1)
     {
       if(isCarMoving && cars[i].isCarMoving)
       {
            StopTheCar();
       }
     }
                    //Running into another car:
    else if(  (getCarCoordinates(6) > cars[i].getCarCoordinates(0) - 50
    && getCarCoordinates(6) < cars[i].getCarCoordinates(6) + 50)
    && ( getCarCoordinates(7) < cars[i].getCarCoordinates(1) + 50
    && getCarCoordinates(5) > cars[i].getCarCoordinates(5) - 50)
    && CarRotatedPosition%2==0 && CarRotatedPosition ==
        cars[i].CarRotatedPosition)
    {
        if(isCarMoving && cars[i].isCarMoving)
        {
           StopTheCar();
        }
    }
    else
    {
        StartTheCar();
    }
} } }
```
Note that a new field **uniqueID** has been added to the Car Object, in order to provide support for this feature. **UniqueID** will take values starting from 0, and count as much as the number of cars displayed until the current moment.

Road Topology

The selection of the road topology is a key factor to obtain realistic results when simulating car movements. Important aspects to take into considerations would be: the length of the street, the frequency of road intersections, the density of building, the structure of the street, the material which covers the street, the bumps from the road. These aspects affect mobility metrics such as: average speed or the density over the stimulated map.

Some topologies could be defined in the following ways:

User-define graphs: the road topology is specified by listing the vertices of the graph and their interconnecting edges. The graph in this case refers mostly to 3D graphic designs, which allows setting of heights, hills and bumps to the roads by using the Y axis.

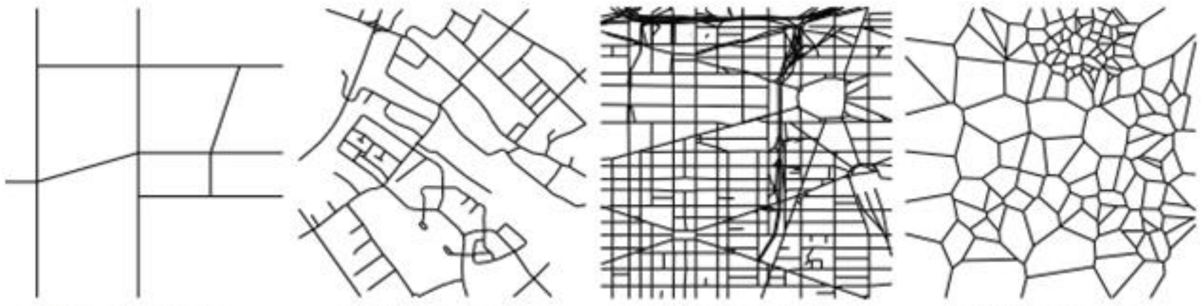
GDF map: is the road topology which imports the Geographical Data file (GDF), using such maps is not possible without a proper library. Most of GDF file libraries are not free for use.

TIGER map: is the road topology which extracts a TIGER database from a map. A TIGER database is actually a Topologically Integrated Geographic Encoding and Referencing System that automates the mapping and related geographic activities required to support some programs. The level of detail in this topology is not as high as the one provided by GDF standard, but this database is open and free to use.

Clustered Voronoi graph: is the road topology which is randomly generated by creating Voronoi tessellation on a set of non-uniformly distributed points.

In all these cases the road topology is implemented as a graph over whose edges the movement of vehicles is constrained.

These topologies are illustrated by the image below:



(a) User-defined topology (b) GDF map topology (c) TIGER map topology (d) Clustered Voronoi

Figure 1.2 Road topology

Vehicular traffic movement in urban scenarios are far from being random. Cars tend to move between points of interest, which are often common to many drivers and can change in time (example: offices and schools can be strong attraction points, but mainly in the morning when people get to work, or school). In order to make cars move in a simulator: 2 choices are given for the trip generation. The first is a random trip, as the start and stop points of movement are randomly selected among the vertices of the graph which represent the road topology. The second is an activity sequence generation, in which a set of start and stop points are provided in an explicit way, by the road topology description. In such case the cars are forced to move among them.

In the created project, the trip is randomly chosen within a list of possible routes in which the start and the stop points were explicitly selected.

Regardless of the trip generation method, the path computation and the selection of the best sequence of edges to reach a selected destination, can be performed in three ways. The first way is to select the shortest path to reach the destination. The second way does not only consider the length of the path but also the traffic congestion level, by weighting the cost of traversing an edge also on number of cars already traveling on it. The third way extends the first two ways by also accounting for the road speed limit when calculating the cost of an edge, in a way that fastest routes are preferred.

The combination of trip generation and path computation methods offers a wide range of possibilities, when the definition of vehicular movement paths is a factor of interest for simulations.

A stressful situation is that vehicles behavior can dynamically vary in presence of traffic lights, according to red-to-green and green-to-red switches. In the red-to-green case, a car currently decelerating to stop at a red light will accelerate again if the semaphore turns green before it has completely halted. In the last case, a vehicle keeping its pace towards a green light will try to stop if the light becomes red before it has passed through the intersection.

In this last case, a minimum breaking distance is evaluated by means of simple kinematic formula:

$$\bar{s} = v\,t - \frac{\kappa b}{2}t^2 = v\left(\frac{v}{\kappa b}\right) - \frac{\kappa b}{2}\left(\frac{v}{\kappa b}\right)^2 = \frac{v^2}{2\kappa b}$$

This formula describes the space needed to come to completed stop in relation with the current speed of the vehicle, **v**, the time **t** and the deceleration value, **κb**. The last parameter represents the maximum safe deceleration, for example: the comfortable braking value **b** scaled by a factor **κ ≥ 1.** The final expression above is obtained by substitution of **t** with (**v/κb**), which is the time at which a zero velocity is reached with a constant deceleration **κb** on the current speed **v**.

The environmental ambience

In order to provide a more realistic note, environmental ambiance for the traffic will be added, by calling the following functions after grabbing the Window from its constructor inside the **int main()** function:

```
#include <SDL_mixer.h>
int main(int args, char** argv)
{
    Window windObject;
    SDL_Window* window = windObject.getWindow();

    Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 4096);
    Mix_Music* music;
    music=Mix_LoadMUS("ambience/busy street.mp3");
    Mix_PlayMusic(music, -1); //-1 will repeat the audio.
/*The rest of the code goes from here*/
}
```

The function **Mix_OpenAudio()** is used to initialize the mixer API, it takes in 4 parametes: int frequency, Uint16 format, int channels, int chunksize:

Frequency	It is the output sampling frequency in samples per second. (Hz)
Format	It is the format of the sample
Channels	It is the number of channels for the mixer: 2 – stereo, 1 - mono
Chunksize	Bytes used for output sample.

Mix_Music is a struct data type used for Music data which is define in **SDL_mixer** library.

Mix_LoadMUS() will return a pointer to Music data, whereas **Mix_PlayMusic()** method will play that music data.

The sound to be played as environmental ambiance was recorded from the flat’s balcony, and then converted into a proper **.mp3** file.

Synthesis

The simulation of VANET involves two different aspects. First, there are issues related to the communication among vehicles and the second very important aspect is related to the mobility of the nodes.

The problems regarding the communication among vehicles:

Network simulators usually cope with communication issues and focus on network protocol characteristics. Communication models highlight the information flows between vehicles which get interesting when vehicles move with relative high speeds. VANET applications are affected by wireless networking aspects such as transmission delay, packet loss or network access scheme. However, accurate network simulation implies additional complexity and makes several large scale VANET applications unsuitable for simulation.

Another matter regarding the communication of VANET is the security. Security should satisfy four main objectives: it should ensure that the information received is correct (information authenticity), the source is who he claims to be (message integrity and source authentication), the node sending the message cannot be identified and tracked (privacy), and the system is robust (robustness).

The information in VANET must be delivered to the nodes with in time limit so that a decision can be made by the node and perform action accordingly.

The problems regarding the mobility of nodes:

Traffic simulators take into account the traffic model, not necessarily in conjunction with VANET. In the majority of VANET applications, vehicles react to messages. For instance, if a driver receives a message saying that the road ahead is congested, that driver will probably change her/his route. In order to study such reactions, an integrated simulator is needed, one in which the relationship between the vehicle mobility module and the network communication module is bidirectional.

The mobility model implemented in some simulators is not a sufficiently accurate representation of actual vehicle mobility. For example, in some models, each vehicle moves completely independent of other vehicles, with a constant speed randomly chosen. Multi-lane roads or traffic control systems are not taken into consideration.

Generally, when choosing a mobility model for a VANET simulator, the complexity and the accuracy are the most important issues need to be considered. The benefit of using a very complex and accurate mobility model, together with the performance penalty, should be carefully evaluated, keeping in mind the purpose of the VANET simulator and the type of applications for which it is designed.

Conclusions

This paper work aimed to realize a brief introduction regarding the algorithmic way in which a traffic simulator can be programmed, what resources could be used, topologies to follow and applications in which such algorithm can be used to solve the real-world traffic problems.

While I was performing studies for finishing this paper work, some conclusions have been reached:

The simulation of vehicular networks on the road is generally made using Network simulations. A network simulation is a technique whereby a software program models the behavior of the network by calculating the interaction between the different network entities. Since VANETs are related to vehicular entities, the simulator would have to use a discrete event simulation where the modeling of the systems in which the variables state change within discrete moments of time.

VANETs can use any wireless networking technology as their basis. The most prominent are short range radio technologies like WLAN (either standard Wi-Fi or ZigBee).

The advantage of Wi-Fi networking technology for VANETs is that it can effectively locate a vehicle which is inside big campuses like universities, airports, and tunnels, based on wireless navigation system functions.

Providing vehicle–vehicle and vehicle–roadside communication within VANETs can considerably improve traffic safety and the comfort of driving or traveling. In addition, effective measures such as media communication between vehicles can be enabled, providing methods to track automotive vehicles. The vehicular communication is expected to contribute to more efficient roads by providing timely information to drivers.

VANETs can be used as part of automotive electronics, to identify an optimally minimal path for navigation with minimal traffic intensity. Such a system could be used as a city guide to locate and identify landmarks while travelling in a foreign city.

Security is an important issue for routing in VANETs, because many applications will perform life-or-death decisions which can have devastating consequences. The characteristics of VANETs make the secure routing problem more challenging than it is in other communication networks.

The performance of a routing protocol in VANETs depend heavily on the mobility model, the driving environment, the vehicular density, and many other facts.

In order to show the impact of VANETs over the traffic safety and its efficiency via simulations, accident and human behavior models are required, which means understand how drivers will react based on some additional information which must be provided.

Having a universal routing solution for all VANETs application scenarios or a standard evaluation criterion for routing protocols is necessary.

References

1. **Reducing Traffic Jams via VANETs**: Florian Knorr, Daniel Baselt, Michael Schreckenberg and Martin Mauve. IEEE Transactions on Vehicular Technology, Vol. 61, No. 8, October 2012.
2. **Traffic Flow Dynamics**: Martin Treiber, Arne Kesting, Institut für Wirtschaft und Verkehr: TomTom Development Germany GmbH
3. **A survey of mobility models in Wireless Adhoc Networks**: Fan Bai and Ahmed Helmy, University of Southern California U.S.A
4. **Vehicular Networking, A survey and Tutorial on Requirements, Architectures, Challenges, Standards, and Solutions**: Georgios Karagiannis, Onur Altintas, Eylem Ekici, Geert Heijenk, Boangoat Jarupan, Kenneth Lin, and Timothy Weil, IEEE Communications Surveys & Tutorials Accepted for Publication.
5. **Vehicular Mobility Simulation for VANETS,** Marco Fiore, Politecnico di Torino.
6. **Simulation of VANET Applications**, Valentin Cristea, Victor Gradinescu, Cristian Gorgorin, Raluca Diaconescu, Liviu Iftode
7. **Challenges in Vehicle Ad Hoc Network,** Ms. Divyalakshmi Dinesh, Prof. Manjusha Deshmukh, IJETMAS volume 2, December 2014
8. **User Requirements Model for Vehicular Ad hoc Network Applications,** Irshad Ahmed Soomro, Halabi, Hasbullah, Department of Computer and Information Sciences, Universiti Teknology PETRONAS