With support of NSF Award no. EEC-0754741

GrooveNet3: A Scalable Traffic Simulator for Congestion Probing and Prediction

NSF Summer Undergraduate Fellowship in Sensor Technologies
Uchenna Kevin Anyanwu (Computer Engineering) – California State University at San Jose
Advisor: Rahul Mangharam

Abstract

Traffic congestion is a huge problem for developed countries such as the United States. Severe traffic congestion that can slow traffic up to several miles is common for everyday drivers in United States. Before the problem can be solved, government agencies and private agencies need reliable data to help understand traffic congestion. The goal of this research is to enable GrooveNet, a vehicle-to-vehicle simulation program, to record and analyze large amounts of historical traffic data, quickly and efficiently, and provide a playback capability on historical data. Methods used to develop this robust capability were proposed by Kanul, et al, at Carnegie Mellon University. Using historical traffic data, GrooveNet displays congestion over time and proposes a less congested route for a simulated vehicle. By gathering historical data, the upcoming version of GrooveNet will have a large knowledge repository that will ultimately give way to sophisticated machine learning and prediction algorithms that allow GrooveNet to navigate vehicles from origin to destination, safely and quickly, thus decreasing traffic congestion.

1. Introduction

This research proposes a vehicle-to-vehicle simulator called GrooveNet which will not only retrieve historic information generated by vehicles, but also, in real-time, produce information on the current state of traffic, using efficient and fast real-time congestion probing techniques.

GrooveNet is a vehicle-to-vehicle hybrid (allows communication between real and virtual vehicles) simulation program. The graphic below shows how two vehicles propagate information between one another using assistance from virtual vehicles. No matter how far two real vehicles are apart from each other, GrooveNet will find the fastest path that will allow two real vehicles to communicate (*Figure 1).* GrooveNet allows vehicles to communicate time-critical information, such as air-bag deployment, vehicle malfunction, and more, as means to promote safety and awareness for drivers within a bounded, topological region. GrooveNet is an open-source research project spearheaded by Dr. Rahul Mangharam. GrooveNet currently simulates hundreds of vehicles on any street map in the US. Each vehicle can be equipped with a model. For example, the Random Walk Model allows a particular vehicle to approach an intersection and randomly choose which way to turn at the intersection.

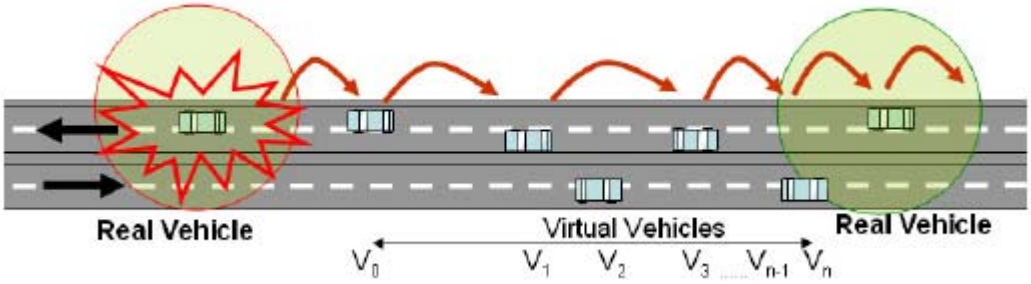

Figure 1

In 2007, Kanual Shah, Raj Rajkumar, Ph.D, and Rahul Mangharam, Ph.D, presented several models that will enable vehicles in GrooveNet to probe traffic conditions and make reasonable decisions about which path to take in order to minimize distance traveled and time to reach a destination. Kanual, et al, proposed two types of models, which characterizes the behavior of each vehicle. These modes are Active congestion probing and Passive congestion probing. Active congestion probing includes passive congestion control, which enables vehicles to send and receive traffic information from a central server, and active congestion control, which enables vehicles to send and receive traffic information from each other. Passive congestion probing enables each vehicle to maintain data about paths taken and combine that data with transportation data to create trend chart or historical information.

This project has several goals. The first is to understand current traffic generation, recording and playback capabilities of GrooveNet 2.0. Second, develop a mechanism for recording historic speed and traffic density data. Third, develop a mechanism for recording traffic accidents. Fourth, define an efficient binary format to record and playback historic traffic data. Fifth, integrate the above within GrooveNet 3.0. Lastly, use historic traffic data to predict traffic congestion and determine alternate less-congested routes in GrooveNet 3.0. The overall goal is to demonstrate traffic congestion development under normal conditions and under traffic incidence in the simulator, GrooveNet.

2. Background

Traffic congestion is a major problem in fast, developing countries. In the United States, there is a need to minimize traffic congestion and increase safety for all drivers. According to U.S Mobile Report congestion is getting worse in America's 437 urban areas [5]. Unfortunately, there is no one treatment for traffic congestion.

2.1 Vehicle to Vehicle Simulator

Unlike traffic simulators that handle traffic flow using a queuing perspective, GrooveNet was developed to enable real-time communication between real vehicles, simulated vehicles and between real and simulated vehicles. Incorporating GPS and Wireless networks allows GrooveNet vehicles to obtain a holistic view of traffic flow and make intelligent path predictions based on that current state of traffic. The following figure shows two different vehicles communicating traffic information. Vehicle status from a real vehicle can be broadcast to other vehicles (real and simulated) within a wide range. (*Figure 2*).

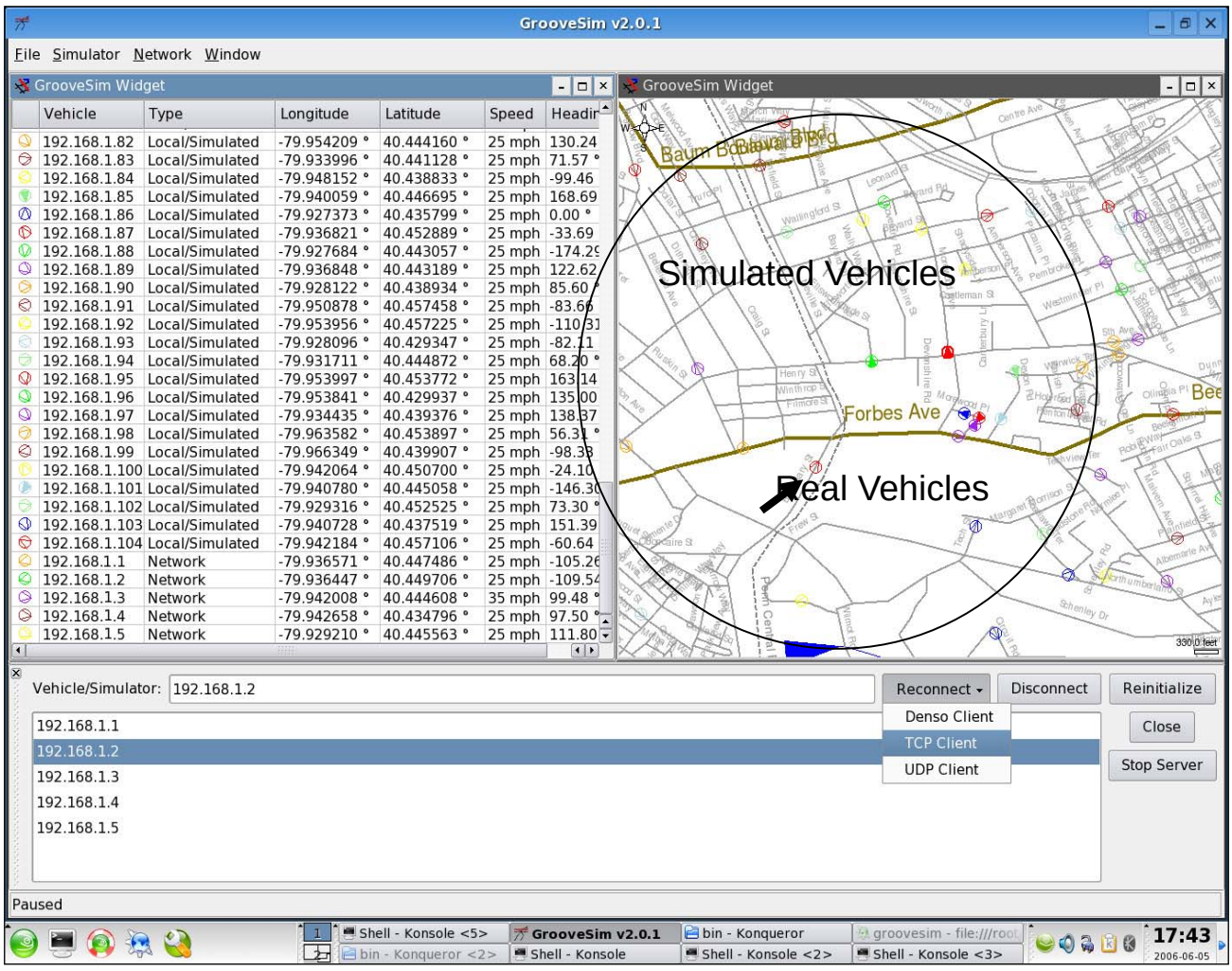

Figure 2

2.2 **Active congestion probing**
Active congestion probing is important to vehicles that need to determine the optimal time and distance to a location in real-time. This type of congestion probing provides every vehicle the ability to communicate and work together in order to minimize traffic congestion. The gain in this technique is that accurate data about traffic congestion can be achieved.

2.3 Passive congestion probing
Passive congestion probing combines route information and historical transportation data, located in a database, from each vehicle to produce a trend chart that can be used by vehicles to make intelligent decisions.

2.4 Hybrid congestion probing

The Hybrid congestion approach combines active and passive congestion approaches. First, the idea is to obtain active congestion information, and then, if there isn't sufficient active congestion information, GrooveNet will use the historic/trend data.

2.5 Traffic Models

Various models have been created to understand traffic flow using historic data. These models are classified as Macroscopic, Microscopic, nd Mesoscopic models. The Macroscopic model is similar to water flowing through a pipe. Microscopic model characterizes individual vehicles with detailed behavior. Mesoscopic model defines individual vehicles with aggregate behavior. One macroscopic model of interest is “The Lighthill, Whithman and Richards model” (Figure 2). The model can be written as two forms follows:

$$\frac{n(x)\partial C(x,t)}{\partial t} + \frac{\partial q(x,t)}{\partial x} = 0 \qquad \frac{\partial K}{\partial t} + \frac{\partial}{\partial x} Q_e(K,x) = 0$$

Figure 2

This is expressed as the sum of partial derivatives, where C(x,t) means traffic density, n(x) is the number of lanes at position x, and q(x,t) is the traffic flow of vehicles per hour at location x at time t. “The Lighthill, Whithman and Richards model” is called the Law of Conservation of vehicles in traffic.

3 Current GrooveNet HistoricalData and Algorithms

GrooveNet has several data structures for extracting, saving, and printing historical data, but is not integrated with the overall system. In order to enable GrooveNet to record and play back historical data, the primary goal is to reconstruct and complete the data structures and functions implemented by Kanul.

3.4 HistoricalData Struct

```
Typedef struct HistoricalData {
        int RecordId , SegmentSpeed;
```

```
        int speedL[48], speedR[48];
}HistoricalData;
```

The HistoricalData data structure was implemented to organize variables in memory that pertains to historical data. The importance of the above implementation is to serve as storage for historical information, which will be written to a file. The variables are as follows: first, two integer arrays of forty eight segment (lane) speeds; second, an integer for RecordId, which is an identification for a road on the map.

HistoricalInfo Class

```
Class HistoricalInfo{
public:
        //Constructor and Destructor functions
        int Load ( …);
        inline bool IsLoaded (… ) const
        void Save( … );
        void Setspeed( … );
        int Extractinfo( … );
        int GetRecIndex(  … );
        void LoadHistoricalInfo (…);
private:
        bool m_Loaded;
        std::vector<HistoricalData> recSpeed;
}
```

The above HistoricalInfo class is the current baseline for analyzing and displaying traffic data in GrooveNet. The goal of this class is to give the software an efficient mechanism for handling historical information. The Load function, working together with ExtractInfo and isLoaded function, is used to load historical information—a text file in binary format—from the hard drive to memory. The Save function, which uses Setspeed, GetRecIndex, and GetIndex, writes traffic speed data to a file in binary efficient format. The private members are m_Loaded, which allows GrooveNet to know if the system is safe for processing. The private dynamic array of HistoricalData structures, which is an STL (standard template library) implementation of a dynamic array, is used to store structures of historical datum in memory, which will be written to a file on the hard drive.

4. Revised GrooveNet HistoricalData and Algorithms

Several revisions need to be made in Kanual's implementation of the HistoricalData class, structures and accompanying algorithms.

First, the HistoricalData structure declaration needed to be changed in order to reflect a simple database that contains essential traffic information. SpeedL and speedR variables were clearly not feasible because writing at most 80 speed information to a file in disk would hoard a lot of disk space. Lieu of using SpeedL and SpeedR, the best approach was to obtain average speed of each segment. RecordId variable was retained. Additional variables were added to the HistoricalData struct. The revised structure is as follows:

```
typedef struct HistoricalData {
int RecordId;
int SpeedDirection;
int SegmentSpeed;
int NumVehicles;
}HistoricalData;
```

Second, the HisticalInfo class structure needed to be revamped. This class is important to efficiently getting historical data from GrooveNet to a file and reading from it. When redesigning the class, it was important to keep the overall functionality as is. The first change was to delete all unwanted functions such as Setspeed, Getspeed, and GetRecIndex. The second change was to add important functions such as HistoricData_Read, Historic_Count, h_count.

```
class HistoricalInfo
{
public :
        HistoricalInfo();
        ~HistoricalInfo();
        int Load(…);
        inline bool IsLoaded() const
        {
                return m_bLoaded;
        }
        void Save(…);
        int  Extractinfo(…);
        int  HistoricData_Count(… );
        void HistoricData_Read(…);
        void LoadHistoricalinfo();
        std::list<unsigned int> returnListRec();
private:
        bool m_bLoaded;
        std::vector<HistoricalData> recSpeed;
        std::list<unsigned int> records;
        int h_count;
        QString h_strFilename;
};
```

The most important function of the HistoricalInfo class structure is the Save and LoadHisticInfo function. The Save function allows GrooveNet to save segment identification numbers that have vehicles moving less than 80 percent of the road speed every 5 seconds of the simulation time. In order to develop this algorithm, a function to capture data every five seconds needed to be implemented. Within that function, a call to the Save must be made. The Save function iterates through all segments, finding all vehicles in the same segment, taking the average speed of all vehicles in the segment, and saving all average speed segment ids that are less than 80 percent of the road speed. All data is saved on to multiple master files and an index file. The master file has all historical data, separated by a delimiter, which divides each historical data entry.

The user programmer must notice that each row is 5 seconds of recording time. When played by the

simulator, each row will be displayed as 1 second.

5. Playback Feature

A playback feature requires several buttons to be implemented into the current user interface. Qt graphics library was utilized to develop the following buttons: PLAY, FASTFORWARD, REWIND, PAUSE, and STOP (figure 3). These buttons are used to control the Map Visual, which displays the data by the proposed HistoricalInfo Class. The following screen shot displays the new user interface of GrooveNet:

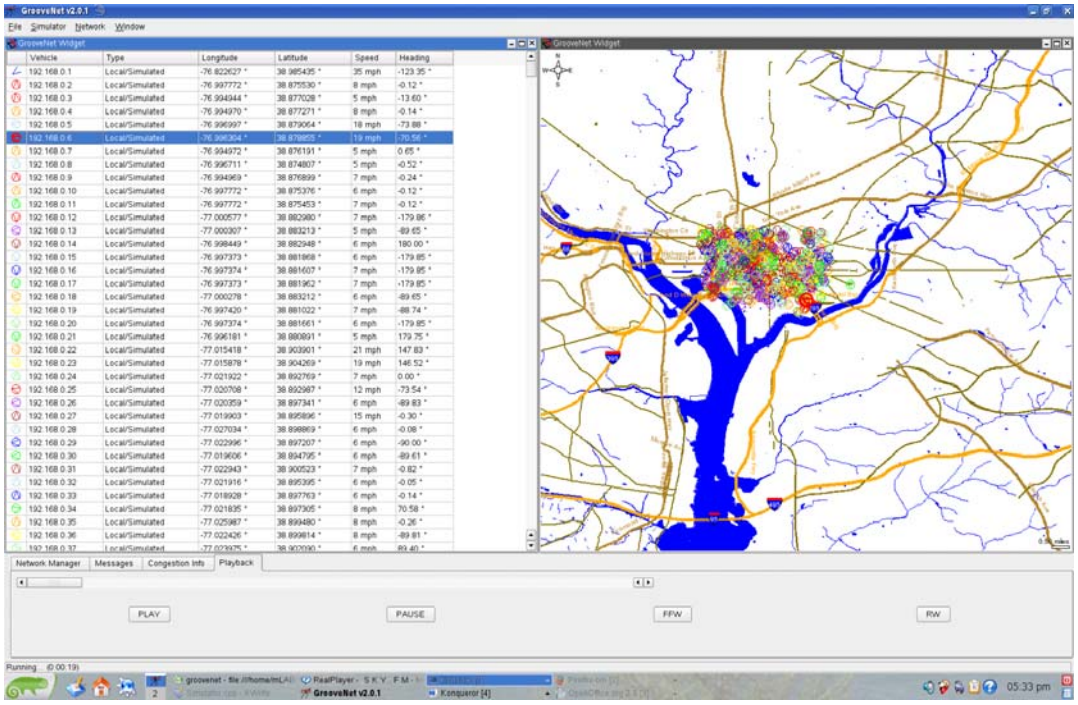

Figure 3

A QPlayback data structure is derived from QWidget class in order to create the above features. The constructor class, which creates the play, pause, fast-forward, and rewind buttons, requires that we define signals and slots. Signals—play, pause, rewind, stop, fast-forward (not displayed) —activate messages that are sent to the MapWidget, which has a slot implementation for receiving messages.

5.1 Play Function

The play function executes an algorithm that analyzes both an index and binary files. As described above, the index file contains the size in bytes that should be read by GrooveNet in order to obtain a data set. Each line represents one second of the recording. When this data has been read into memory, GrooveNet can open the specific binary data file, then reads one data set and displays the traffic information on the screen.

5.2 Fast-forward Function

The fast-forward function executes the same functionality as the play function instead it runs much faster. The programmer can adjust the speed by changing the variable global variable #defines in the header file of QPlayback.

5.3 Stop Function

The stop function resets the current position to the beginning of the historical file

5.4 Pause Function

The stop function holds the current position of the play button

6. Discussion and Conclusion

GrooveNet facilitates understanding dynamics of traffic congestion and alleviating traffic prevention. GrooveNet has the ability to simulate a wide array of traffic congestion scenarios on variety of maps. Enabling GrooveNet to record and play back historical data allows the user to experience the dynamics of traffic build-up over time and space. When using the Traffic recording functionality, GrooveNet compiles data as it is being produced and sends that data into a repository or data base, on the hard drive. At any time in the future, the user is able to either analyze the raw data dump or re-load it to GrooveNet for playback purposes. The user is able to playback the file as if it were a video and to view progression of simulated traffic overtime.

7. Recommendation

An index file method can be used to implement the rewind button in the playback functionality. The index is basically a catalog of how to find specific information in a huge data file. The index method can be applied to GrooveNet by writing a set of algorithms that writes to a separate file, printing the size in bytes of each data record in the historical file. This file is read in memory by GrooveNet, allowing it to traverse backward through the historical file.
The next step is to incorporate the playback feature in the parallel version of GrooveNet. This can be done easily using the proposed algorithms; however, there is always room for optimization for the new version. Also, since the parallel version of GrooveNet uses less processor power, GrooveNet can spend more time processing greater chunks of data for a smoother playback capability.

8. Acknowledgements

I would like to thank my advisors, Dr. Rahul Mangharam and Dr. Jan Van der Spiegel of the University of Pennsylvania, for their support and advice. I would also like to thank all Miroslav Pajic and Danny Lustig of the University of Pennsylvania for their patience and close collaboration with this project. I would also like to thank the National Science Foundation for providing deserving undergraduates, like myself, a chance to be a part of cutting edge research at the University of Pennsylvania.

9. References

[1] K. Shah, R. Rajkumar, and R. Mangharam, “Real-time traffic congestion probing Using Vehicle to Vehicle Networks,” M.S. thesis, Carnegie Mellon University, Pennsylvania, PA. December 2007.

[2] R. Mangharam, D. S. Weller, D. D. Stancil, R. Rajkumar, J. S. Parikh, GrooveSim: A Topography-Accurate Simulator for Geographic Routing in Vehicular Networks *Proceedings of Second ACM* International Workshop on Vehicular Ad hoc Networks (Mobicom/VANET 2005). Cologne, Germany. September 2005.

[3] R. Mangharam, D. S. Weller, R. Rajkumar, Priyantha Mudalige and Fan Bai, "GrooveNet: A Hybrid Simulator for Vehicle-to-Vehicle Networks," presented at *Second International Workshop on Vehicle-to-Vehicle Communications (V2VCOM),* San Jose, USA. July 2006.

[4] D. Schrank and T. Lomax, “The 2007 URBAN Mobility Report,” September 2007. [Online]. Avalaible:http://financecommission.dot.gov/Documents/Background%20Documents/mobility_ report_ 2007_wappx.pdf [Accessed: July 12, 2008]

[5] J.P.Lebacque, J.B. Lesort, F.Giorgi. “Introducing Buses into First-Order Macroscopic Traffic Flow Model,” *Transportation Research Record: Journal of the Transportation Research Board,*vol.1644/1998, pp. 70- 79, January 2007.

10. APPENDIX : SOURCE CODE

```
#ifndef _QPLAYBACK_H
#define _QPLAYBACK_H

#include <qwidget.h>
#include <qmutex.h>
#include <qlayout.h>
#include <qlistbox.h>
#include <qlabel.h>
#include <qlineedit.h>
#include <qtextedit.h>
#include <qpushbutton.h>
#include <qscrollbar.h>
#include "MapDB.h"
#include "ModelMgr.h"
#include "MapVisual.h"
#include <qfile.h>

class QLabel;
class QLineEdit;
class QPushButton;
class QPopupMenu;
class QListBox;
class QScrollBar;

class QPlayback : public QWidget
{
Q_OBJECT
public:
QPlayback(QWidget * parent = NULL, const char * name = 0, Qt::WFlags f = 0);
virtual ~QPlayback();
virtual void SetPlaybackInitialized(bool m_bLoaded);
protected slots:
void signalPlay();
void SET_TIMER_PLAY();
void SET_TIMER_STOP();
void SET_TIMER_FAST();
void SET_TIMER_PAUSE();
void SET_TIMER_SCROLLER();
void moveScroller();
void signalMoved(int);
protected:
QHBoxLayout * p_layout;
QPushButton * p_play;
QPushButton * p_fforward;
QPushButton * p_rewind;
QPushButton * p_pause;
QPushButton * p_stop;
```

```
QWidget * p_listClients;
QScrollBar * p_scroll;
bool stop;
MapVisual * ptrModelV;
Model*  ptrModel;
QString strFilename;
QTextStream reader;
QTimer * play_time;
int scroll_value;
};

#endif
#include <qlayout.h>
#include <qlabel.h>
#include <qlineedit.h>
#include <qpushbutton.h>
#include <qpopupmenu.h>
#include <qlistbox.h>
#include <qwidget.h>
#include <qtimer.h>
#include <qdialog.h>

#include "QPlayback.h"
#include "HistoricalInfo.h"
#include "MainWindow.h"
#include "QMapWidget.h"
#include "Visualizer.h"
#include "Simulator.h"

#define FFW 8

QPlayback::QPlayback(QWidget * parent, const char * name, Qt::WFlags f)
: QWidget(parent, name, f)
{

QHBoxLayout * button_p_layout;

button_p_layout = new QHBoxLayout(this,8,8,"layout");    //declare layout for buttons

p_play = new QPushButton("PLAY", this);
p_play->setEnabled(false); //disable the button

connect(p_play, SIGNAL(clicked()), this, SLOT(signalPlay()));

p_stop = new QPushButton("STOP", this);
p_stop->setEnabled(false);

p_fforward = new QPushButton("FFW", this);
p_fforward->setEnabled(false);
```

```
p_rewind = new QPushButton("RW", this);
p_rewind->setEnabled(false);

p_pause = new QPushButton("PAUSE", this);
p_pause->setEnabled(false);

p_scroll = new QScrollBar(Qt::Horizontal, this);
p_scroll->setEnabled(false);
p_scroll->resize(1000, p_scroll->height()); //resize the scroll bar

play_time = new QTimer(this);

button_p_layout->addWidget(p_play, 0, Qt::AlignCenter); //add the widget to the layout
button_p_layout->addWidget(p_pause, 0, Qt::AlignCenter);
button_p_layout->addWidget(p_fforward, 0, Qt::AlignCenter);
button_p_layout->addWidget(p_rewind, 0, Qt::AlignCenter);
button_p_layout->addWidget(p_stop,0, Qt::AlignCenter);

connect(p_play, SIGNAL(clicked()), this, SLOT(SET_TIMER_PLAY()));
connect(p_stop, SIGNAL(clicked()), this, SLOT(SET_TIMER_STOP()));
connect(p_fforward, SIGNAL(clicked()), this, SLOT(SET_TIMER_FAST()));
connect(p_pause, SIGNAL(clicked()), this, SLOT(SET_TIMER_PAUSE()));
connect(p_play, SIGNAL(clicked()), this, SLOT(SET_TIMER_SCROLLER()));
connect(play_time, SIGNAL(timeout()), this,  SLOT(signalPlay()));
connect(play_time, SIGNAL(timeout()), this, SLOT(moveScroller()));
//connects a timer to the play slot; emits timeout signals every 1 second

p_scroll->setRange(0,51);
p_scroll->setSteps(1, 0);

connect(p_scroll, SIGNAL(sliderMoved(int)), this, SLOT(signalMoved(int)));
}
void QPlayback:: signalMoved(int temp)
{
        //get the current value of the slider when press
        if ( scroll_value < p_scroll->value( ) )
        {
                signalPlay(); //read forward in the data file
        }
        else
        {
                //read backward in the data file
        }
}
void QPlayback:: SetPlaybackInitialized(bool m_bLoaded)
{
        if(m_bLoaded)
        {
```

```
                p_play->setEnabled(true);
                p_fforward->setEnabled(true);
                p_rewind->setEnabled(true);
                p_pause->setEnabled(true);
                p_scroll->setEnabled(true);
                p_stop->setEnabled(true);
        }
        else
        {
                p_play->setEnabled(false);
                p_fforward->setEnabled(false);
                p_rewind->setEnabled(false);
                p_pause->setEnabled(false);
                p_scroll->setEnabled(false);
                p_stop->setEnabled(false);
        }
        stop = false;
        g_pMapDB->Path4=true;
        g_pSimulator->m_ModelMgr.GetModel("MapVisual0", ptrModel);
        if(ptrModel)
        {
                ptrModelV = (MapVisual *)ptrModel;
        }
        printf("CONSTRUCTOR CALLED \n");
}
QPlayback:: ~QPlayback()
{
        delete p_pause;
        delete p_play;
        delete p_rewind;
        delete p_fforward;
        delete p_scroll;
        delete p_layout;
        p_pause = p_play = p_rewind = p_fforward = NULL;

}
void QPlayback:: SET_TIMER_PLAY()
{
        play_time->start(1000, FALSE);
}
void QPlayback:: SET_TIMER_SCROLLER()
{
        play_time->start(1000, FALSE);
}
void QPlayback:: SET_TIMER_PAUSE()
{
        play_time->stop();
        scroll_value=p_scroll->value();
        printf("The value of the slider is: %i\n", scroll_value);
```

```
}
void QPlayback:: SET_TIMER_FAST()
{
	int msec_time;
	msec_time= 1000/FFW;
	play_time->start(msec_time , FALSE);
}

void QPlayback:: SET_TIMER_STOP()
{
	play_time->stop();
	g_pMainWindow->m_pHist->file->reset();
	p_scroll->setValue(0);
}
void QPlayback:: signalPlay()
{
	g_pMainWindow->m_pHist->HistoricData_Read(); //read new data
	ptrModelV->recenterVal();
}
void QPlayback:: moveScroller()
{
	//find the the current position
	//add one to the current position
	//store the new poision
	p_scroll->setValue(p_scroll->value() + 1);
}


#ifndef _HISTORICALINFO_H
#define _HISTORICALINFO_H

#include "CarModel.h"
#include "MapDB.h"
#include "CarRegistry.h"
#include "StreetSpeedModel.h"
#include "QMapWidget.h"
#include "MapVisual.h"
#include "Visualizer.h"
#include "QPlayback.h"

#include <qfile.h>
#include <qtextstream.h>

#define PARAM_MODEL "MODEL"
#define PARAM_TYPE "TYPE"
#define PARAM_DEPENDS "DEPENDS"
```

```
typedef struct HistoricalData {
	int CongId;
	int RecordId;
	int SpeedDirection;
	int SegmentSpeed;
	int NumVehicles;
}HistoricalData;

class HistoricalInfo
{
public :
	HistoricalInfo();
	~HistoricalInfo();

	int Load(const QString & strFilename);
	int  Getspeed(int RecID, int direction, struct timeval CurTime);
	int  Extractinfo(const QString & strLine, std::vector<HistoricalData > & recInfo);
	int  GetIndex(struct timeval CurTime);
	int  HistoricData_Count(QTextStream * reader );
	int  GetRecIndex(int RecID);
	void Save(QTextStream & writer, double & timeStamp);
	void Setspeed(int RecID, int direction, int speed, struct timeval CurTime);
	void HistoricData_Read();
	void LoadHistoricalinfo();
	void ClearList();
	void returnListRec(std::list<unsigned int> &, std::list<unsigned int> &, std::list<unsigned int>
&, std::list<unsigned int> &);

	std::vector<HistoricalData> recSpeed;
	std::list<unsigned int> records1;
	std::list<unsigned int> records2;
	std::list<unsigned int> records3;
	std::list<unsigned int> records4;

	QFile * file;
	QString h_strFilename;
	QTextStream * reader;

	bool m_bLoaded;
	int countStars;
	int h_count;
};
#endif

#include "HistoricalInfo.h"
#include "MainWindow.h"

#include <qfile.h>
#include <qtextstream.h>
```

```
#include <qiodevice.h>
#define MAXSIZE 60

HistoricalInfo::HistoricalInfo()
{
	m_bLoaded = false;
	countStars = 0;
}

HistoricalInfo::~HistoricalInfo()
{
	recSpeed.clear();	//might not be needed
}
int HistoricalInfo::Load(const QString & strFilename)
{

	file = new QFile(strFilename);		//create new file, attached to a file name

	reader = new QTextStream(file);	//point reader to the file

	if (!file->open(IO_ReadOnly | IO_Translate))		//test that file is opens sucessfully
	{
		errno = ENOENT;			//exit if not opened correctly
		return 0;
	}

	reader->setDevice(file); //set device

	if (g_pMainWindow != NULL && g_pMainWindow->m_pLblStatus != NULL) //test main
window pointer
		g_pMainWindow->m_pLblStatus->setText("Loading Historical Info...");

	if(!m_bLoaded)		//if playback tab is not loaded
	{
		g_pMainWindow->m_pPlayback = new QPlayback(g_pMainWindow->tabsContainer,
"playback");
		//create new playback tab
		g_pMainWindow->tabsWidget->addTab(g_pMainWindow->m_pPlayback,
QString("Playback"));
		//add the playback table to the window
		g_pMainWindow->m_pPlayback->SetPlaybackInitialized(true);
		//enable playback tab
		m_bLoaded = true;	//set playback tabe loaded
	}
	return 1;
}
void HistoricalInfo::HistoricData_Read()
{
	QString line;
```

```
	while (!(line = reader->readLine()).isNull())//read lines from the file until there is no more data
	{
		if (line.isEmpty())	//if line is empty then go to the next line
			continue;
		if(line == "*")		//if line is a star(delimiter) then ...
		{				//we have found a whole data set then break
			break;
		}
		if(!Extractinfo(line, recSpeed))  //we can parse the line
			printf("error occured while extracting historical data\n");    //or error out!
	}
	LoadHistoricalinfo(); //load segement data into a list
}
int HistoricalInfo::HistoricData_Count(QTextStream * reader)
{
	QString line;
	int h_count = 0;
	while (!(line = reader->readLine()).isNull())
	{
		if (line.isEmpty())
			continue; //skip the next if, go to while
		if(line == "*")
		{
			++h_count; //track the number of historical data sets in file
		}
	}
	if(h_count>0)
		return h_count;
	else return 0;
}
int HistoricalInfo::Extractinfo(const QString & strLine, std::vector<HistoricalData > & recInfo)
{
	//parse one line of input and dump it into respective element in to a vector of historical data
	HistoricalData temp;
	QStringList list = QStringList::split(" ",strLine);
	temp.CongId = list[0].toInt();
	temp.RecordId = list[1].toInt();
	temp.SegmentSpeed = list[2].toInt();
	temp.SpeedDirection = list[3].toInt();
	//printf("%d, %d \n",temp.CongId, temp.RecordId,temp.SegmentSpeed, temp.SpeedDirection);
debugging
	recInfo.push_back(temp);	//save data into datastructure
	return 1;		//return sucess
}
void HistoricalInfo::Save(QTextStream & writer, double & fTime)
{
	std::map<in_addr_t , CarModel* >::iterator iterCar;		//declare iterators for maps and a list
	std::map<in_addr_t, CarModel *>::iterator iterCar1;
	std::list<int>::iterator iterSegment;
```

```
	std::map<in_addr_t, CarModel *> * pCarRegistry = g_pCarRegistry->acquireLock(); //not sure if needed

	std::list<int> segment;			//
	std::list<int>recSizes;
	std::list<int>::iterator iterSizes;

	MapRecord * psRec;
	StreetSpeedModel speedLim;
	short roadSpeedLimit = 0.0;
	double fSeconds(0);
	float mSec(0);
	mSec=modf(fTime, &fSeconds);
	int iSeconds = (int)fSeconds;

	for (iterCar = pCarRegistry->begin(); iterCar != pCarRegistry->end(); ++iterCar)
	{
		unsigned int record = iterCar->second->GetCurrentRecord();

		bool foundSeg = false;
		//search list of used segments
		for(iterSegment= segment.begin() ; iterSegment != segment.end(); ++iterSegment)
		{
			if( *iterSegment == record)
			{
				foundSeg=true;		//segment found in list
			}
		}

		if(!foundSeg) //segment not in list
		{

			segment.push_back(record); //save segment in list

			unsigned int totalspeed = 0;

			unsigned int count = 0;

			short roadAverage = 0;

			bool SpeedL1 =0, SpeedL = 0;

			short direction1 = 0;

			short direction = iterCar->second->GetCurrentDirection();

			int vecSpeed = iterCar->second->GetCurrentSpeed();
```

```
roadSpeedLimit = GetRoadSpeed(record);

if(direction>= -4500 && direction < 13500)
{
	SpeedL = true;
}
if(direction< -4500 || direction >= 13500)
{
	SpeedL = false;
}
for (iterCar1 = pCarRegistry->begin(); iterCar1 != pCarRegistry->end(); +
+iterCar1)
{
	if ( iterCar1->second->GetCurrentRecord() == record ) //found a vehicle
	on the same segment
	{
		direction1 = iterCar1->second->GetCurrentDirection();

		if((direction1 >= -4500 && direction1< 13500) && SpeedL)
		{
			totalspeed = totalspeed + iterCar1->second-
			>GetCurrentSpeed();
			++count;
		}
		if((direction1 < -4500 || direction1 >= 13500) && !SpeedL)
		{
			totalspeed = totalspeed + iterCar1->second-
			>GetCurrentSpeed();
			++count;
		}
	}
}
roadAverage = totalspeed/count;

if (roadAverage >= .70*roadSpeedLimit && roadAverage <
0.80*roadSpeedLimit) //record only vechicles that are traveling below 80% of
the speed limit
{
	writer<<1<<" "<<record<< " "<< roadAverage <<" "<<direction<<"
	"<<count<<" "<<iSeconds<<"\n";
}
if (roadAverage >= .60*roadSpeedLimit && roadAverage <
	0.70*roadSpeedLimit) //record only vechicles that are traveling below
	80% of the speed limit
{
	writer<<2<<" "<<record<< " "<< roadAverage <<" "<<direction<<"
	"<<count<<" "<<iSeconds<<"\n";
}if (roadAverage >= .50*roadSpeedLimit && roadAverage <
	0.60*roadSpeedLimit) //record only vechicles that are traveling below
```

```
                        80% of the speed limit
                {
                        writer<<3<<" "<<record<< " "<< roadAverage <<" "<<direction<<"
                        "<<count<<" "<<iSeconds<<"\n";
                }if (roadAverage < 0.50*roadSpeedLimit) //record only vechicles that are
                        traveling below 80% of the speed limit
                {
                        writer<<4<<" "<<record<< " "<< roadAverage <<" "<<direction<<"
                        "<<count<<" "<<iSeconds<<"\n";
                }

        }
    }
    writer<< "*\n";
    ++countStars;
    //find how many bytes have been read so far
    //recSizes.push_front(0); //save this value in a list
    //write the list to an index file
    if(countStars == MAXSIZE) //we have captured 1 minute of data
    {
        //
        writer.unsetDevice();  //detach writer from data file and close the data file
        //detach writer from index file
    }
}
void HistoricalInfo::LoadHistoricalinfo()
{

    for (int n = 0; n < recSpeed.size() ; n++)
    {
        switch (recSpeed[n].CongId)
        {
            case 1: //70..80
                records1.push_back(recSpeed[n].RecordId);//copy every record element
                from recSpeed into a list
            break;
            case 2: //60..80
                records2.push_back(recSpeed[n].RecordId);//copy every record element
                from recSpeed into a list
            break;
            case 3: // ..
                records3.push_back(recSpeed[n].RecordId);//copy every record element
                from recSpeed into a list
            break;
            case 4: //.
                records4.push_back(recSpeed[n].RecordId);//copy every record element
                from recSpeed into a list
            break;
        }
```

```
        }


}
void HistoricalInfo :: returnListRec(std::list<unsigned int> & recordsa, std::list<unsigned int>
&recordsb, std::list<unsigned int> &recordsc, std::list<unsigned int> &recordsd)
{
        recordsa = records1;
        recordsb = records2;
        recordsc = records3;
        recordsd = records4;
}
void HistoricalInfo::ClearList()
{
        records1.clear();
        records2.clear();
        records3.clear();
        records4.clear();
        recSpeed.clear();
}
```