Automotive Simulation System

Steven Bradley

Student Number: 10122985

University of Western Australia

School of Electrical, Electronic and Computer Engineering

Centre for Intelligent Information Processing Systems (CIIPS)

Supervisor: Associate Professor Thomas Bräunl

Submitted: 28 May 2009

Steven Bradley
12 Lowanna Way
City Beach WA 6015

May 29, 2009

The Dean
Faculty of Engineering, Computing and Mathematics
The University of Western Australia
35 Stirling Highway
Crawley, Western Australia 6009

Dear Sir,
I submit to you this dissertation entitled "Automotive Simulation System" in partial fulfilment for the Bachelor of Computer Science and Bachelor of Engineering degree at the University of Western Australia.

Yours faithfully

Steven Bradley

1 Acknowledgements

I would like to thank the University of Western Australia, the faculty of Engineering, Computing and Mathematics for the opportunity and support during my course work. My thanks grow exponentially to and for Thomas Bräunl, and the CIIPS group, for all their support and advice.

I must acknowledge the inspiration, love and support of my family Anne, Rod and Claire. Much love and thanks to you.

-Steven

2 Abstract

The Automotive Simulation is a large and complex software project with the goal of bringing a realistic driving simulator to the workbench. Many attributes of this a large system require reform and redesign to improve performance and stability. To this end, this thesis discusses the attempt to move AutoSim to a new operating system platform so as to make better use of hardware and leverage the Client/Server architecture of the design.

3 Content Page

3.1 List of Figures

3.2 List of Tables

3.3 Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CIIPS	Centre of Intelligent Information Processing Systems
COLLADA	COLLAborative Design Activity
GCC	GNU C/C++ Compiler
GNU	a recursive acronym for “GNU's Not Unix”
GUI	Graphical User Interface
KDE	K Desktop Environment
PAL	Physics Abstraction Layer
RARS	Robot Auto Racing Simulator
TORCS	The Open Racing Car Simulator
UWA	University of Western Australia
XML	Extensible Markup Language

4 Introduction

The Automotive Simulator, described hereafter as *AutoSim*, is a software simulation and development platform for the design and calibration of engineering projects. The Centre for Intelligent Information Systems (CIIPS) is a research group in the school of Electrical and Electronic Engineering at the University of Western Australia and has the goal of researching robotic and automation systems as they apply to modern vehicles. AutoSim reinforces that objective by consisting of a software realistic world simulation using programmable robots that provides visualisation and instrumentation sensors around a virtual road network environment. The software is developed for Microsoft Windows™ with the goal of having a cross platform, distributable and extensible modular system. These objectives are to be met in part by moving also onto the Debian Linux™ operating system while improving reliability and tweaking performance. AutoSim is built around the use of open source technologies so that the software can be distributed to other research institutions and is designed to communicate over an Ethernet network.

4.1 Thesis Structure

This thesis will examine some of the examples of simulation and the work already completed into AutoSim in Chapter 5. Chapter 6 conducts a review and explanation of the current implementation. Followed by Chapter 7 contains discussion of the implementation of improvement objectives.

5 Literature Review.

5.1.1 Examples of Other Simulators

There are various examples of automotive simulators currently being developed for the purposes of entertainment, for specific applications like driver training and for product development and research purposes.

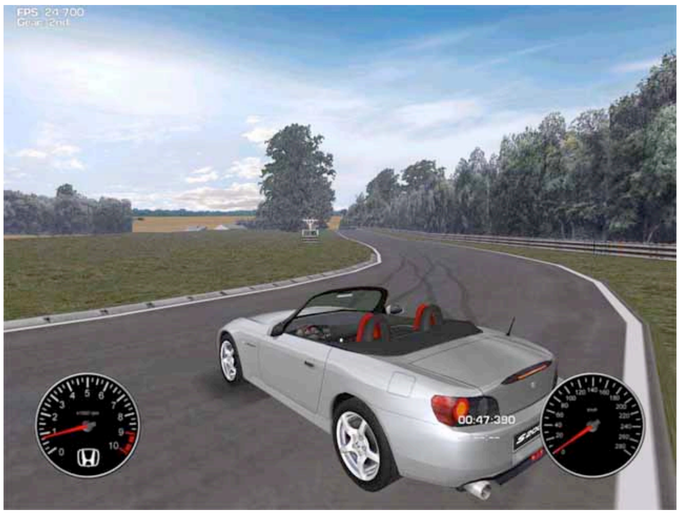


Figure 1 : Example screenshot of Racer (used from [1])

Racer

This project is a free, as in cost, cross-platform arcade car simulation project. Racer Author Ruud van Gaal in the Netherlands describes his project as,

> *"Racer is a free cross-platform car simulation project (for non-commercial use), using professional car physics to achieve a realistic feeling and an excellent render engine for graphical realism. Cars, tracks and such can be created relatively easy (compared to other, more closed, driving simulations). The 3D, physics and other file formats are documented. Editors and support programs are also available to get a very customizable and expandable simulator.*
> *OpenGL is used for rendering." [1]*

Racer is similar in many functional ways to AutoSim however the project source is not free enough to be distributable. This makes it unsuitable as AutoSim intends to develop into a research and development platform that may be distributed around the world. Racer also lacks any design consideration for use of robotic programming or instrumentation.

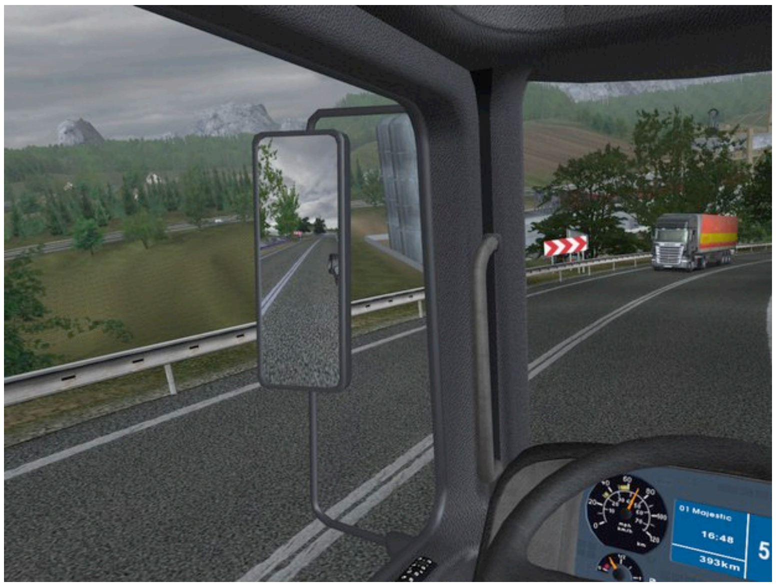


Figure 2 : Euro Truck has a realistic cabin (screenshot from [2])

Euro Truck Simulator

The 'Euro Truck Simulator' is an immersive simulator experience of driving a truck or various large transport vehicles around the cities and roads of Europe. It aims to provide the most realistic experience with detailed graphics and rich interactive environments. In particular of the driving cabin with items such as wing mirrors. This proprietary simulator is designed to share the truck driving experience around Europe with the patrons of the company SCS Software.

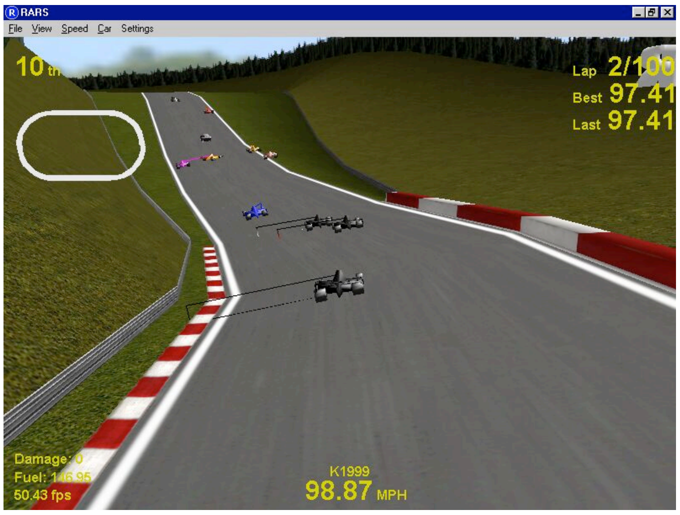


Figure 3 : RARS in action (screenshot from [3])

Robot Auto Robotic Simulator (RARS)

Built with the idea of providing a simple 3d robot simulator for competitive racing and educational purposes for the subjects of Artificial Intelligence and real-time adaptive optimal control. This project has been superseded by The Open Racing Car Simulator (TORCS) project.

Figure 4 : TORCS a modern AI championship (screenshot from [4])

The Open Racing Car Simulator (TORCS)

Using more modern graphics technologies over the previous project. TORCS continues RARS effort in AI competition by focusing on the competitive programming of compliant programmed routines to handle a given track layout.

TORCS is built around a community of fun seeking technological savvy fans who want to see their racers exceed, the project does allow users to create events for their AI programmed racing teams and allows tracks to be customised for the development of real world tracks. Users contribute based on their want to participate in an event based on a particular scenario.

5.1.2 Motivation

Road transport safety is an ongoing concern as the number of road transport vehicles increases nationally [5], and this importance for Australia, continues to grow as does the proportion of road transportation in Australian society [6]. Car manufacturers are seeking ways to prevent traffic accidents in a many new directions, often with the inclusion of a driver assistance product [7].

Systems have also been developed for the use of simulating hardware through a hardware-in-the-loop vehicle simulation. This is a vehicle on a workbench approach in order to develop complex adaptive systems like traction assist [8]. AutoSim will bring the software bench-top to a device. Visual, interface and monitoring devices that can help manage risks in the national road transport system are important technologies that need innovative and sometimes complex solutions. At present driver assistance and enhancement innovations are slow to market and expensive to develop. The steep start-up costs of providing real hardware for testing and the rigorous testing requirements mean significant time delays between development iterations that inhibit research. A worthwhile goal therefore is to provide a targeted testbed for driver-assistance functions. By building on the existing work of AutoSim the CIIPS group will be able provide realistic simulated data and instruments for use in driver-assistance technology projects.

Testing and development of driver assistance and safety systems can be an expensive, dangerous and time intensive iterative process. The development of a useful simulation is required therefore to provide an abstract way of dealing with real world situations and hence reduce the expense, danger and length of time involved with development iterations.

Advances in computation and electronics has shown promise to improve the safety and lives of individuals with the introduction of quality cheap cameras and microcontroller capability. With the assistance of this development platform the opportunity to leverage those technologies in the improvement of safety, through research projects such as Active Advance Warning Systems [9] and assistance Adaptive Cruise Control [10].

Existing platforms catering for the simulation of automotive systems have been focused on engine performance and car dynamics. Vehicle manufacturers in general do research in house and keep their development tools proprietary. The use of virtual reality environments to simulate data for use in further research has strong potential to accelerate improvements in driver assistance technology, and vehicle instrumentation research [11]. This is where AutoSim is positioned, to provide an adaptable and available workstation to efficiently and rigorously test new automotive devices.

5.2 Software Theory

Software engineering concepts applied to AutoSim and discussed later in this thesis are described in the following section.

5.2.1 Singleton

The singleton design pattern uses methods to ensure a class only has one instance or a specified number of instances and provides a global point of access for them.

Figure 5 is a class diagram of a singleton design, here the function getInstance() returns a reference to the unique singleton attribute. The class Singleton has a constructor that initialises the unique singleton variable or returns it if already established.

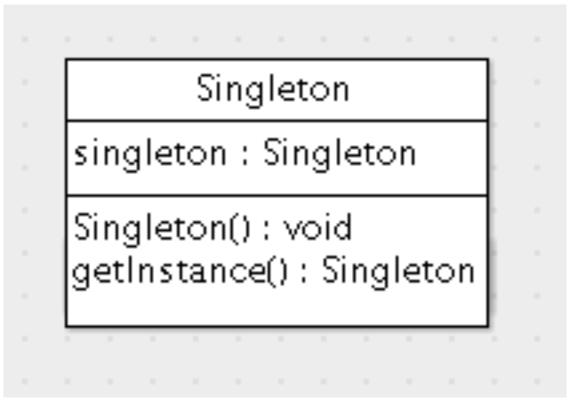


Figure 5 : A Singleton Class Diagram

This design pattern of a unique ubiquitous object has the advantage of controlling access. Managing and encapsulating the sole instance, due to the Singleton being a class, it has complete and strict control over when and how other classes access it. The same principles may be achieved by using global variables however the Singleton design pattern avoids complicating the name space by polluting it with global variables to store sole instances. Another advantage comes from the flexibility this approach provides over class methods of software control, including allowing a variable and dynamic number of instances. Singleton classes, as a class can have their operation and structure refined by use of inheritance.

5.2.2 State Design Pattern

A state design pattern uses class methods to convert objects between classes. The object becomes the state and this allows an object to alter behaviour based on the internal state changes in that object. This is achieved by implementing an abstract state class that acts as a proxy for explicitly defined state classes that inherit the abstract class (see Figure 6).

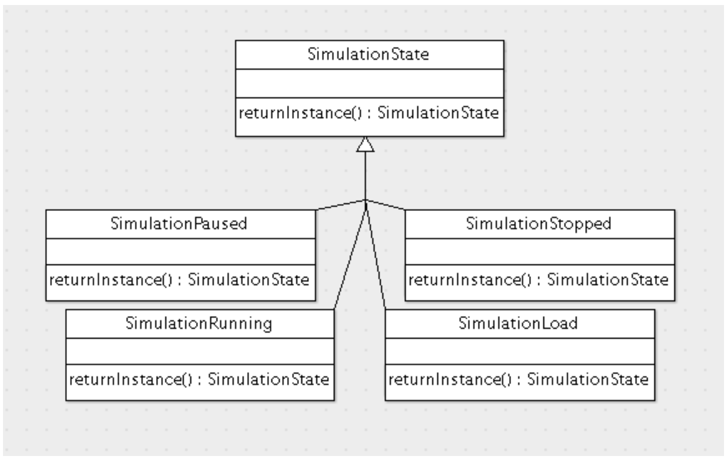

SimulationState
returnInstance() : SimulationState
SimulationPaused
returnInstance() : SimulationState
SimulationStopped
returnInstance() : SimulationState
SimulationRunning
returnInstance() : SimulationState
SimulationLoad
returnInstance() : SimulationState

Figure 6 : State Design Inheritance

This design modality has the consequence of localising behaviour of a state to the relevant class which partitions behaviour for different states. The alternative is to use variables and data within a class to define internal states and have context operations validate data and hence the current state explicitly. This complication must be implemented repeatedly and throughout the implementation. Further without this design pattern including a new state would require changing several large conditional operation statements.

An advantage of the state design pattern is it ensures state transitions are explicit, when an object defines a state by the internal data values of a class the change of state has no representation in the data until such time as assignment to those variables has been complete. By having state objects, the context is protected from invalid and inconsistent internal states due to context shifting. These state transitions are atomic and will be independent of interruptions from another task and hence protected from circumstances of race conditions.

5.2.3 Chain of Responsibility

The chain of responsibility design pattern avoids coupling the sender of a request to a receiver by using more than one object as an intermediately to handle the request. These chains receiving objects do pass requests along the chain until an object handles it or drops it.

The reasons for having a decoupled sender and receiver can be because a class in a chain may not have to know how the object will be handled. A handler can only interpret where to further process an object but can only know within its own context what an object is and is required of it. In a chain of responsibility, see Figure 7, each class is an object handler sorting or manipulating until the object is handled.

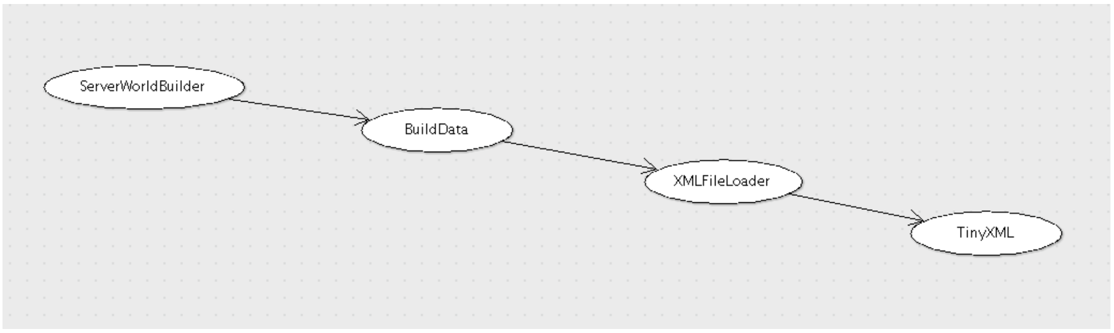


Figure 7 : Chain of Responsibility

A chain of responsibility has the advantage of loose coupling because this frees an object from having to know which other object handles a request. An object requires only that it will be handled appropriately for an object of that class. The sender, ServerWorldBuilder, and receiver, TinyXML, need not know of each other while maintaining software integrity. This allows for increased flexibility when handling object responsibilities. Objects can be replaced with various alternatives at run-time and various requests can be interpreted by another single object.

A disadvantage of the chain design pattern, is a receipt of a request is not guaranteed. Since the sender does not know explicitly of the receiver an object may be handled inappropriately or go unhandled when a chain is not established properly during runtime.

6 Review of the current implementation

AutoSim is an open source simulation platform. An environment is replicated with models that include vehicles, roads, intersections, buildings and various obstacles including trees road signs. The effects of gravity and wind are replicated on the virtual vehicle which a user can drive using automated scripting, GUI controls and a joystick control steering wheel and pedals.

Maps, model positions, car parameters are read from XML files generated from user input as well as information freely available from the Open Street Map Project [12]. Open Street Map is a public collaboration effort to provide an open and free database of geographical data for public access. AutoSim uses this as the basis of the maps which it uses running a simulation.

AutoSim has responsibilities to create a visualisation of a virtual three-dimensional environment and to maintain track of a virtual world of elements and obstacles. This is broken up into a client and server executables to allow the simultaneous execution on multiple computers and at different locations can be connected to the same virtual world via a network. This client server architecture is an important part of the transition to a multiple node network simulation.

The two executables are broken up into several key responsible classes.

6.1 *AutoSimClient*

The AutoSimClient has the responsibility of creating the graphical interface and visualisation of the models and environment of the physical simulation. It has a graphical user interface shown below in Figure 8.

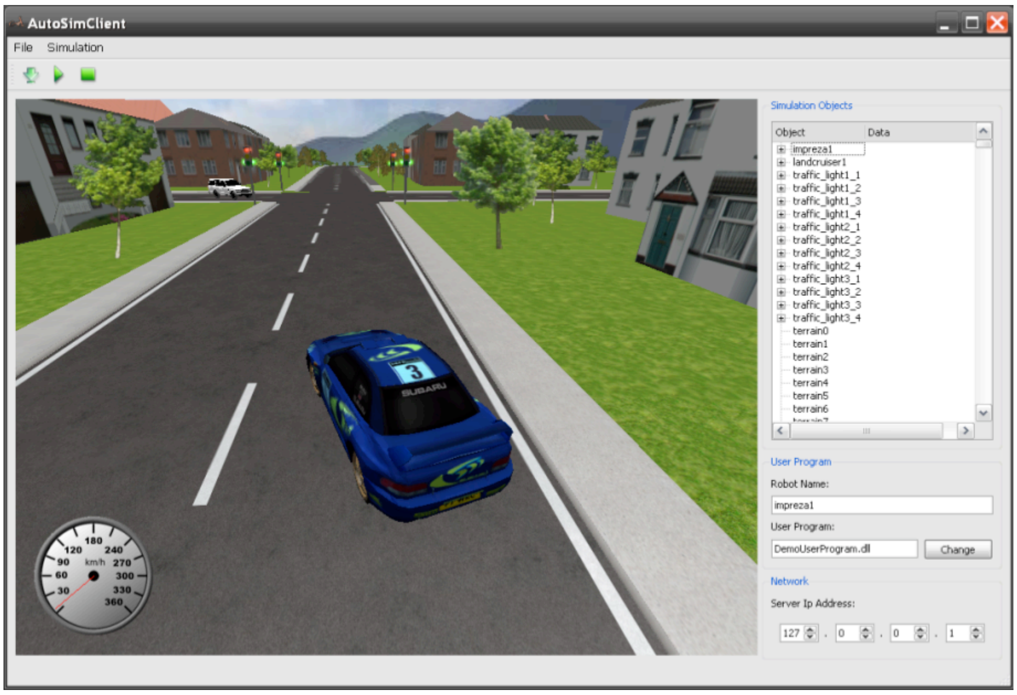


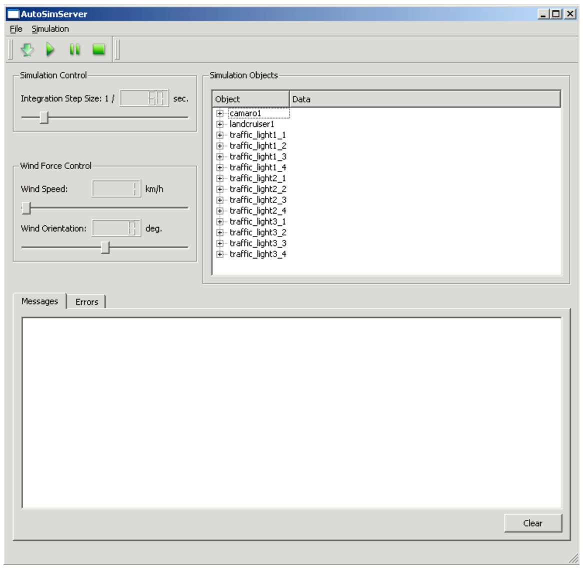
Figure 8 : The Original AutoSimClient Interface, (used image from [13])

The AutoSimClient dominant software architecture design is the Singleton. A Singleton design pattern, described by Figure 5, restricts the instantiation of a class to a single object. In software a single instance is useful for co-ordinating actions across a system. [14]

The classes that are key singletons in AutoSimClient are ClientController, GraphicsManager, RakNetClient and UserProgramDLL. They are all implemented to be individual threads to execute their individual part. They are regularly referring back to another singleton of other threads for triggers to enact load and run behaviours over shared memory.

6.2 AutoSimServer

The AutoSimServer has the responsibility of handling the status of the environment in the simulation. It produces interactions of the models through the physics engine.



camaro1
landcruiser1
traffic_light1_1
traffic_light1_2
traffic_light1_3
traffic_light1_4
traffic_light2_1
traffic_light2_2
traffic_light2_3
traffic_light2_4
traffic_light3_1
traffic_light3_2
traffic_light3_3
traffic_light3_4

Figure 9 : AutoSimServer Interface

The simple interface of AutoSimServer displays a expandable tree of robots in the simulated world. Expanding each item shows which body the robot is connected to and the attributes of the body in the world.

6.3 Used Software Libraries

AutoSim adopts several free, open-source libraries to provide features and functionality for the simulation. The libraries relevant to this thesis are:

6.3.1 Qt

Qt, to be pronounced like the word "cute", is the GUI and application development framework that AutoSim is built upon. This well documented library is used to create the GUI of applications and to provide some, XML parsing and cross platform file handling features. Qt is well known for being used in the Linux desktop environment KDE and several leading applications including Google Earth and the browser Opera [15].

Qt runs on all major platforms including Java based Qt Jambi and support for Embedded Linux with Qt Extended. Qt Software, formally known as Trolltech

Software, releases the software. The library uses C++ and has the notable exceptions to a normal library by the use of non-standard tools to produce standard C++ source code before compilation.

6.3.2 RakNet

RakNet is game network engine developed for the rapid development of multiplayer gaming software. Based on the UDP protocol this C/C++ library is a protocol with low overhead and low response time. The project written in large part and administered by Jenkings Software, the tagline of RakNet is *"Multiplayer on the Deadline"* [16].

In AutoSim RakNet is used to provide the application network stack and to communicate between the core applications the client and server. During the development of AutoSim this communication has been done through localhost, the loopback address of a computer.

6.3.3 TinyXML

TinyXML is a super lightweight and simple open source XML parser written in C++. Without support for features such as Extensible Stylesheet Language and ignoring Document Type Definitions TinyXML focuses on ease of implementation. Developed by Lee Thomason as a cross platform freely available open source project TinyXML is used in AutoSim for the parsing and storage of simulation information XML files. Thus providing AutoSim with human readable and editable data files.

6.3.4 PAL

Physics Abstraction Layer or PAL is an open source physical simulation API abstraction system. It works as an interface between the low-level physics engines and a high-level simulation application allowing flexibility for an application to switch physics engines such as Bullet, Newton Game Dynamics Engine and the Tokamak Physics Engine to name a few. In AutoSim, PAL acts as the interface for physics engines for AutoSimServer where all the physics calculations take place. PAL provides an extensive range of features including support for XML, the Sycthe Physics Editor file format, COLLADA and benchmarking tools.

6.4 Limitations of the implementation

AutoSim is a large and complex project. It has had the participation of several students over the time it has been in development. This includes Torsen Sommer who designed the physics and server implementation [17], Johannes Brand who created the graphics virtualisation and road generation features [13] and Pål Ruud who built on the vehicle models and physics with a focus on vision [18] these three contributed the foundations of the project. And the project has been and continues to be built on by students.

On inheriting the project in 2008 the project faced some major issues.
One included the lack of documentation of the project that can be expected in any large software project. Another was the performance of the software; the simulation had grown in complexity in such a way as to become unstable in various ways and the slow progression of iterations of the physics simulation itself. It appeared complexity was introduced to the software system by the lack of development in the overall software design.

In the source code of AutoSim, the implementation began to rely too heavily on the singleton pattern for application control. In these cases each thread of control is a singleton and presumes an overall objective and attempts to enact that behaviour. The crossing of responsibilities is inevitable as is creating race conditions between responsibilities. This issue is further exemplified with the use of a chain of responsibility whose classes handle requests with no receipt or reply to update the singleton master class. A singleton itself is useful in defining a global state because it can be referenced uniquely and globally without complication. However with several singleton 'global states' and with no validation, by the use of objects in a chains of responsibility, rebuttal of a forward global state transition in runtime is impossible. Unchecked transitions internally will allow the simulation to lead into to an unknown and invalid working condition. When this occurs a terminal segmentation fault or invalid access error will occur as internal references executing in one of the singleton classes has become stale and invalid. Once a global inconsistent state occurs there is no means to move back to a previous state or into a correcting state.

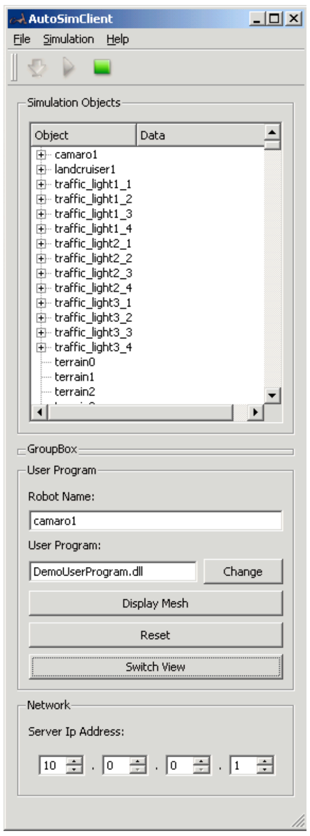
At the time of writing steps were being made to address some of the software design issues identified. The lack of any clear state design made execution complex. The singleton designs were being enhanced to deal with that complexity by implementing state design patterns with their internal state information. Validation of state transitions will be discussed in this thesis.

6.4.1 Objectives

Improvement objective would be to test and implement cross platform feature of the project by building AutoSimServer for the Linux platform. The problem starts is two fold, to replicate the build process already in place and to introduce more stability to the application by introducing state transitions.

7 Implementation and Outcomes

The client interface has changed quite a bit since the previous version of AutoSim.



Object	Data
camaro1	
landcruiser1	
traffic_light1_1	
traffic_light1_2	
traffic_light1_3	
traffic_light1_4	
traffic_light2_1	
traffic_light2_2	
traffic_light2_3	
traffic_light2_4	
traffic_light3_1	
traffic_light3_2	
traffic_light3_3	
traffic_light3_4	
terrain0	
terrain1	
terrain2	

Robot Name: camaro1

User Program: DemoUserProgram.dll

Server Ip Address: 10 . 0 . 0 . 1

Figure 10 : Image of the AutoSimClient side panel

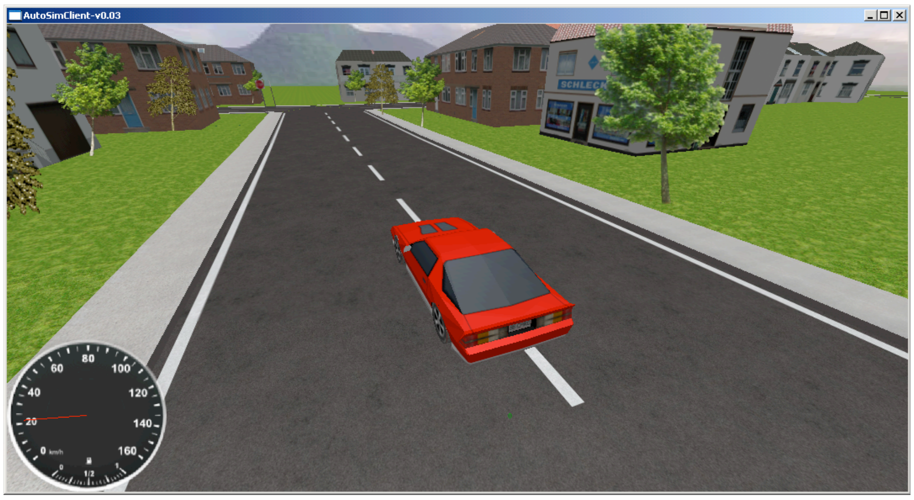


Figure 11 : The stand alone AutoSimClient visualisation window

The client now consists of two windows. The second, the visualisation window can be maximised for use with a second monitor or projector. The two-window interface of AutoSimClient provides an uncluttered view of the controls of the client. The controls of the client are gathered into a separate dialog that provides input for the connection process over the network.

7.1 Objective: Stability by Exception Handling and State Transitions

The internal state of independent threads is reliant on the memory of what they have executed explicitly as well as the implicit memory of the position of that thread. As an example, when loading the SimulationManager initialises from new simulation data loaded by ServerWorldBuilder an independent thread that may or may not have parsed the world attributes correctly. If ServerWorldBuilder, which runs on a separate thread and received an error and so has not loaded yet or is invalid, a fatal error occurs.

Before the transition to the simulation state 'SimulationLoaded', a validation of completion must occur. The following describes what happens without class transition validation.

From SimulationStopped.cpp

```
void SimulationStopped::Load(Simulation* pSimulation, QProgressDialog
*pQProgressDlg, STRING worldFileName)
{
  MSG_LOG << "loadSimulation()";
  ServerWorldBuilder::buildAll(pQProgressDlg, worldFileName);
  ChangeState(pSimulation, SimulationLoaded::Instance());
}
```

Figure 12 : SimulationStopped without validation

Here the buildAll() function is unchecked and executes to create the objects for the simulation. The change of state occurs when buildAll() returns, and SimulationStopped becomes SimulationLoaded.

From ServerWorldBuilder.cpp

```
void ServerWorldBuilder::buildAll(QProgressDialog *pQProgressDlg, STRING
fileName) {
 ServerWorldBuilder *builder = new ServerWorldBuilder();
 BuildDataCreator* buildDataCreator = new BuildDataCreator(new
XMLFileLoader(fileName));
builder->buildWorld();
builder->buildGraphics(buildDataCreator->createBDGraphics());
builder->buildPhysics(buildDataCreator->createBDPhysics());
builder->buildObjects(buildDataCreator->createBDObjects());
builder->buildTerrain(buildDataCreator->createBDTerrain());
  }
delete buildDataCreator;
delete builder;
}
```

Figure 13 : ServerWorldBuilder blind to the results

Here function buildAll() will always return quietly as the chain of responsibility in handling objects will drop unhandled requests such as invalid XML. This is not good as the behaviour of SimulationStopped is no longer blocked and inevitably the application will achieve an invalid state. What is assumed here is an implicit understanding between ServerWorldBuilder and the rest of the simulation towards the required creation of simulation data containing the nodes for each of the XML objects required.

From SimulationStopped.cpp

```
void SimulationStopped::Load(Simulation* pSimulation, QProgressDialog
*pQProgressDlg, STRING worldFileName)
{
  MSG_LOG << "loadSimulation()";
  if(!ServerWorldBuilder::buildAll(pQProgressDlg, worldFileName)) return;
  ChangeState(pSimulation, SimulationLoaded::Instance());
}
```

Figure 14 : SimulationStopped with validation

In the modified SimulationStopped Load() method below control is diverted from the transition if the buildAll() function returns false.

Returning of a Boolean is only possible when a process can state success or otherwise.

From ServerWorldBuilder.cpp

```
bool ServerWorldBuilder::buildAll(QProgressDialog *pQProgressDlg, STRING
fileName)
{
 ServerWorldBuilder *builder = new ServerWorldBuilder();
 BuildDataCreator* buildDataCreator = new BuildDataCreator(new
XMLFileLoader(fileName));
 try
  {
      pQProgressDlg->setValue(10);
      builder->buildWorld();
      pQProgressDlg->setValue(20);
      builder->buildGraphics(buildDataCreator->createBDGraphics());
      pQProgressDlg->setValue(40);
```

```
            builder->buildPhysics(buildDataCreator->createBDPhysics());

            pQProgressDlg->setValue(60);

            builder->buildObjects(buildDataCreator->createBDObjects());

            pQProgressDlg->setValue(80);

            builder->buildTerrain(buildDataCreator->createBDTerrain());

            pQProgressDlg->setValue(99);

        }

        catch(string e)

        {

          cerr << "exception: " << e << endl;

          delete buildDataCreator;

          delete builder;

          return false;

        }

        delete buildDataCreator;

        delete builder;

        return true;

    }
```

Figure 15 : ServerWorldBuilder able to indicate success

Validity checking is established through the use of a try catch method and the use of exceptions. This is a generic way to provide feedback without requiring passing of references between functions. The exception collected is a string so that an error can be reported, further this exception is used to inform the state transition of the failure by returning false which in turn prevents the change of state to an invalid one. Any object in a chain of responsibility can create an exception. However the cost to performance of using such an approach makes it inappropriate for highly used functions. In the case of high performance sections of code, it is better to use functions with a neutral realisation to the validity of context relevant to the next executed commands. For this example of loading XML, the only time penalty is for the user when there is an invalid data set being loaded which is not time critical.

7.2 Objective: building on a Linux system

Towards the objective of achieving the outcome of bringing AutoSim to the Linux platform a build process was required that would produce the required binary files using a Linux system compiler. GCC is a GNU C/C++ compiler GCC was the logical choice with almost universal availability on Unix based systems. GNU Make is a build scripting toolset to provide automation for the compilation process and is similarly widely supported.
AutoSim being of client/server architecture allowed movement of the Server executable over to the Linux platform. This was achieved and is described in the following section.

7.2.1 Porting AutoSimServer

Building AutoSim on another platform was made easier by the design choices and contribution of previous work on the project. This includes the work of Ruud, Sommer and Brand with references in their development process to choosing popular open source libraries, and in large part platform independent libraries [18] [17] [13].

The process of preparing AutoSim involved many tweaks and rewriting differences between what is allowable on the Microsoft compiler and standard C/C++ or varient of implemented in the GNU GCC compiler.

7.2.2 Dynamic Makefile - Idea

The Microsoft Visual Studio integrated developer environment and compiler handled and organised building executables under the Windows environment with modifications for the use of QT source generation tools. This process was roughly followed in order to replicate the build process under Linux. The build process was attempted as follows:

1. Compile the third party Libraries.
2. Build all the source files.
3. Link all the required source binaries, link dependent libraries.
4. Create the executable binaries AutoSimServer and AutoSimClient to their respective locations.

It was clear from the failure that more conditions and scripting must be implemented to achieve this objective. The extra steps included are:

1. Run the QT source generation tools to generate source files for the GUI.
   This is required as QT uses an XML file structure to transparently contain the GUI design specification. This allows programmers to read, modify and implement their own changes using the tools provided by QT or even a basic text editor. QT provides tools to 'generate' C++ source files that can be called and in turn call functions within AutoSim that are linked to them. Thus providing the cross-platform and transparent GUI module that AutoSim needs to also be cross-platform.
2. Check dependencies of source files & compile dependencies first.
   In Microsoft Visual Studio, an integrated software development environment, this is handled internally as links and dependencies between source files are handled as development occurs. For an independent build script this is not true. The choices made by the script must be dynamic in order to prevent long build times and frequent modification.
3. Building dependencies from project locations outside of AutoSimServer.
   Both AutoSimClient and AutoSimServer share source files that reproduce functionality and are identically used. It is important to be able to build each executable independently when contributing to a group project as client code may be unstable and it is unnecessary to build both projects completely.

The implementation of a script to achieve these steps was required. Building on a Linux platform can vary widely based on the tools available and supported for various distributions. To keep AutoSim widely distributable the very wildly supported set of tools, GNU Make and GCC were chosen to be the build platform.

7.2.3 Makefile Design Theory

GNU Make is an application for automated scripting of the build process. It uses a 'Makefile' script to execute commands based on rules and dependencies.
A Makefile is a script of explicit or implicit rules, variable declarations, directives and comments [19]. Rules define how and when a target, for example a source file, is built. Directives execute commands and control logic within make while comments are ignored lines for the purpose of annotation and begin with the hatch symbol '#'.

Rules are based on the standard example as follows:

```
targets : prerequisites
        command
        ...
```

This is interpreted as, when making *target*, ensure *prerequisites* are made, if so execute *command*. Any command that returns an error will halt the make process.

An explicit list of commands can be written to build the executables of AutoSim however such a list must be modified with every addition or removal of source files in the project. And also every explicit target must list each and every dependency required so that the build command can be executed with an assumed success. This is an unwieldy complication.

Implicit rules in Makefiles describe a type of rule for building a type of file, usually based on a partial filename. The set of implicit rules in Make provide useful utility when compiling simple source structures and checking prerequisites. Many of the implicit rules are described from the built-in rules of the GNU Make program.

Variable declarations are written in the form of:

```
VARIABLE=SomeValue
```

And can be included in a future part of the script using the following syntax:

```
all :
echo $(VARIABLE)
```

The above example would when running make execute ‘echo SomeValue’.

Directives are the useful control and command features of GNU Make. These allow inclusion of source and program flow and stop from within a Makefile itself.

Comments are the useful descriptors that help summarise information for a human reader and can be used to ignore or block sections of code from executing.

To be used productively in a multiple programmer environment a Makefile to build AutoSim must make use of the following properties: it will

- Balance simplicity while being general enough for use with little need for modification as source code and dependencies change.
- Be dynamic so that each source file binary will be compiled separately

- Be specifically described in the process so that errors can be understood and that each executable can be compiled separately.

Clearly the complexity of the code leaves the process of writing a completely explicit build script will be nearly impossible, as dependencies follow dependencies like a rabbit warren. To overcome these difficulties a dynamic approach was taken and is described in the following chapter.

7.2.4 Dynamic Makefile – Practice

The following section is a detailed examination of the concepts involved in the execution of the dynamic build process.

```
##
## AutoSimServer Makefile
## Written to replicate the build process used in VS, to build under Linux
## Assumes all libraries are built beforehand this includes:
##        Raknet, PAL, Qt, Bullet, tinyXML
##  Author       :            Steven.John.Bradley+AutoSim@gmail.com
##
##        Requires: binutils
##
```
Table 1 : Introductory remarks

Beginning this script is a description of the purpose and author of the script. Some useful notes are also attached to remind users of the requirements and expectations that may accompany the script. Everything here is put into comments so they will not interfere with the make process. A second hatch symbol was produced to allow the comment to stand out away from any line of ignored code.

In designing this script there are a few requirements before it can become useful. Declarations are used to specify information relevant to the process. These are placed near the top of the script so that they can be observed easily for error checking, and also relatively easy editing. They include locations of important files and resources and also the flags used in the build process. In the code below in Table 2 we see the declaration '$(CURDIR)' this is an inbuilt function of Make that refers to a string containing the absolute current working directory.

```
CXX=/usr/bin/g++

MAKE=/usr/bin/make

INTERNAL_PATH=$(CURDIR)/../release

CFLAGS=-D STATIC_CALLHACK -Wall -Wextra -ansi

##-pedantic

##-pedantic ##libraries headers are not compiled with pedantic and will fail

CONFIG=release


LIBDIR=$(CURDIR)/../../lib

QTDIR=$(LIBDIR)/Qt

RAKDIR=$(LIBDIR)/RakNet

PALDIR=$(LIBDIR)/PAL

XMLDIR=$(LIBDIR)/tinyxml

BULDIR=$(LIBDIR)/bullet


CLIENTDIR=$(CURDIR)/../AutoSimClient-v0.03

SERVERDIR=$(CURDIR)
```

Table 2 : Useful Declarations

Table 2 contains declarations of the compiler flags and references to applications used in the build process which are the 'make' binary and the compiler binary 'g++'. Also included are a number of declarations with arrays of paths to the install location of every library used in this build process. It is assumed that every library is installed under 'LIBDIR' and have the normal name as their directory.

```
##
##      Paths
##

INCLUDE=-I../AutoSimCommon -I$(RAKDIR)/Source -I$(PALDIR) -
I$(PALDIR)/pal_i -I./GeneratedFiles -I./GeneratedFiles/Release -I$(XMLDIR) -
I$(SERVERDIR) -I$(CLIENTDIR) -I$(QTDIR)/include -I$(QTDIR)/include/QtCore -
I$(QTDIR)/include/QtGui -I$(BULDIR)/src
##Include has a reference to AutoSimClient/Server source files but they should be
made mostly independent
LIB_PATHS=-L$(PALDIR)/lib/debug -L$(BULDIR)/src -L$(RAKDIR)/Lib/GNU-
Linux-x86 -L$(QTDIR)/lib
##-L$(XMLDIR)

##
##      Libs
##

##QTLIBS=-lQt3Support -lQtScript -lQtWebKit -lQtCore -lQtSql -lQtXml -lQtGui -
lQtSvg -lQtXmlPatterns -lQtNetwork -lQtTest -lQtOpenGL -lQtUiTools
QTLIBS=-lQtCore -lQtGui
LIB_NAMES=-lbulletcollision -lraknetd $(QTLIBS)
##-llibpal -ltinyxml
LIBS=$(LIB_PATHS) $(LIB_NAMES)
```

Table 3 : Compiler Library flags

The contents of Table 3 show flags used when pre-processing headers for compilation and declaration of paths for linking libraries to the main executable. The Libs section builds the declaration 'LIBS' so that it can be used during the linking process. The lines in ignored comment tags contain all the Qt libraries that are not required but are listed for future reference as they are installed on the system. PAL and tinyXML are not included in the library list as they will be statically linked. They should be included here for dynamic linking as well as removed from inclusion in the static object list before compiling.

```
##
##        AutoSim Builds
##



all: AutoSimServer
        @echo "***** $< is now built ******"
```

Table 4 : Script Catch-all

Table 4 contains the default explicit rule. The rule for 'all' will be executed if the script is run with no parameters and is the part of the script that triggers the default behaviour to continue the rest of the build process. If it is satisfied, i.e. that AutoSimServer is built without error, then a message informing the user is displayed.

```
##
##	Qt MOC and UIC
##

##Makefile path == $(CURDIR)
GEN=$(CURDIR)/GeneratedFiles/$(CONFIG)
GENPATH=$(CURDIR)/GeneratedFiles

MOC_FLAGS=-DUNICODE -DQT_THREAD_SUPPORT -DQT_CORE_LIB -DQT_GUI_LIB
-I$(CURDIR)/GeneratedFiles -I$(QTDIR)/include -I$(GENPATH)/$(CONFIG) -
I$(CURDIR) -I$(QTDIR)/include/QtCore -I$(QTDIR)/include/QtGui

##	Using our own Qt's binaries prevents compatibility issues.
MOC=$(QTDIR)/bin/moc
UIC=$(QTDIR)/bin/uic

##	Typical System Locations with QT-Dev packages installed
##MOC=/usr/bin/moc
##UIC=/usr/bin/uic
```

Table 5 : Qt declarations

The Qt requires that the XML files be processed by tools provided as part of Qt to generate source files related to the GUI. To facilitate this the above declarations in Table 5 will be used in following build rules.

```
##
## BUILDING QT -> UIC -> MOC
##
## Qt:
##	cd $(QTDIR); ./configure -prefix $(QTDIR) -static -qt-sql-sqlite -no-openssl
< yes && make -j 5
##	Remove -j 5 to stop parallel processing and enable clear reading of output
##	Example:	cd $(QTDIR); $(MAKE) -f Makefile
## ^^^ above is taken out, unnecessary, as long as Qt built beforehand
## if included add to the dependencies of MOC below

MOC_FILES : $(GENPATH)/ui_mainwindow.h\
	$(GEN)/moc_mainwindow.cpp\
	$(GEN)/moc_treemodel.cpp\
	$(GEN)/moc_serializabletree.cpp\
	$(GEN)/moc_messagelist.cpp\
	$(GEN)/moc_controller.cpp
MOC: $(GEN) MOC_FILES

$(GEN):
	@echo "*****Creating Qt Release Directory"
	mkdir $(GEN)

$(GEN)/moc_controller.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/Controller.h -o
$(GEN)/moc_controller.cpp
```

```
$(GEN)/moc_messagelist.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/MessageList.h -o
$(GEN)/moc_messagelist.cpp

$(GEN)/moc_controller2.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/Controller2.h -o
$(GEN)/moc_controller2.cpp

$(GEN)/moc_simulation2.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/Simulation2.h -o
$(GEN)/moc_simulation2.cpp

$(GEN)/moc_serializabletree.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/SerializableTree.h -o
$(GEN)/moc_serializabletree.cpp

$(GEN)/moc_treemodel.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/TreeModel.h -o
$(GEN)/moc_treemodel.cpp

$(GEN)/moc_mainwindow.cpp:
	@echo "****Moc'ing $@"
```

```
        @$(MOC) $(MOC_FLAGS) $(CURDIR)/mainwindow.h -o
$(GEN)/moc_mainwindow.cpp


$(GENPATH)/ui_mainwindow.h: $(GEN)/moc_mainwindow.cpp mainwindow.ui
        @echo "***Uic'ing $@"
        @$(UIC) -o $(GENPATH)/ui_mainwindow.h mainwindow.ui
```

Table 6 : Qt Tools Source Generation

The specialist command rules in Table 6 are explicit. While this would normally be an issue as any change to the name, number or structure of these files would require manual editing, however they are unlikely to change as frequently and are an unusual build process they do not fit into the pattern of dependency checking accomplished in the later part of the script. Therefore they can be kept as a reference in an explicit rule. The directories are specifically made to maintain the layout as used by the Windows build process. Commented out is code for building Qt from source, both the libraries and the source generation tools, when it has been extracted from an archive into QTDIR.

For each case as this set of explicit rules is executed, if required, a notice is printed to the console indicating the activity it is processing.

```
##

##        Object Compiling

##


## All source files must be listed here, wildcards include new files automatically

SOURCES = $(wildcard ../AutoSimCommon/*.cpp) $(wildcard $(GEN)/*.cpp)
$(wildcard *.cpp)

## Static link libpal due to PAL breaking dynamic linking support for GCC


OBJECTS = $(SOURCES:.cpp=.o) $(PALDIR)/lib/liblibpal.a

MAKEDEPEND = $(CXX) -MM $(INCLUDE) -o $*.d $<


##

##        Required Objects from another location.

##

DEPEND =        $(CLIENTDIR)/RoadFactory.o\

                $(CLIENTDIR)/MapInfo.o\

                $(CLIENTDIR)/TerrainBuilder.o\

                $(CLIENTDIR)/node.o\

                $(CLIENTDIR)/OsmParser.o\

                $(CLIENTDIR)/RectNode.o\

                $(CLIENTDIR)/segment.o\

                $(CLIENTDIR)/hvector.o\

                $(CLIENTDIR)/kbspline2d.o
```

Table 7 : Listing Target Objects

To build a process dynamically we require that targets be listed without requiring an authored list. The wildcard function achieves this where asterisk alone would fail because when used without the wildcard function; in part of a dependency with a file which has an extension for example, file.o, does already exists, the normal behaviour is to then look for the incorrect string '*.o' as a filename. Which then usually fails, the wildcard function correctly handles the use of asterisks to avoid this confusion. SOURCES now contains the list of source files to be compiled and linked, however each file may depend on others and require the building of some other parts of the project. OBJECTS also now contain a list of file names to be built by compiling SOURCES using a simple extension substitution. Some binary object files from AutoSimClient are required to build AutoSimServer, and so they are listed here.

Importantly the declaration of MAKEDEPEND is a command used later in the build process that uses the GCC pre-processor to create a list of dependencies for a given source file. The argument is supplied through the $< macro and the output generates text files from each C++ source file with the extension .d containing the paths to all dependencies. This is used in the build process and is required for it to complete without complaint of missing dependencies. This automatic process is key to making this dynamic script work. With each source file now with a similarly named .d dependency file all that remains is to manipulate this information into a form usable by GNU Make.

```
##

##      TinyXML library objects

##

TINYXML =       $(XMLDIR)/tinyxml.o $(XMLDIR)/tinystr.o
$(XMLDIR)/tinyxmlerror.o $(XMLDIR)/tinyxmlparser.o


testing:

        @echo $(SOURCES)


%.o: %.cpp

        @echo "**Compiling Object $@ for $<"

        @$(MAKEDEPEND); \

        cp $*.d $*.D; \

        sed -e 's/#.*//' -e 's/^[^:]*: *//'\

        -e 's/ *\\$$//' \

        -e '/^$$/ d'\

        -e 's/$$/ :/' < $*.d >> $*.D; \

        rm -f $*.d

        @$(CXX) $(CFLAGS) $(INCLUDE) -c $< -o $@
```

Table 8 : Implicit Build Rules

Now the information required for the dependencies is located in a .d file with a lot of otherwise useless information references to system files and includes.

The '.d' file will contain lists of dependencies in the form:

/pathtodependency/filename.extension

/anotherpathtodependency/filename.extension

To transform those lists into useful explicit dependency rules the list of commands in Table 8 uses a stream editor tool to read and manipulating the files contents (simply put it removes unwanted parts, adds colon characters and leaving useful parts [20]) which is then redirected to a file with the extension '.D'.

The files contents will then be of the form:

```
<filename.o> : <dependency of filename> \
 <dependency of filename>\
...
<filename.cpp> : <dependency of filename>\
<dependency of filename>\
...
```

Which is later included onto the bottom of the running script. In doing so, the explicit rules for all the source files have been defined dynamically and are in to be used in the compile command (the last line of Table 8) as the rules are executed again. The '%.o : %.cpp' is a pattern matching rule so that the commands to generate dependencies and compile is only run when a new file matching the pattern '*.o' is required or a dependency of that rule, the '%.cpp' , is changed.

```
##
##	Linking
##

AutoSimServer: MOC $(OBJECTS) clientdepend
	@echo "*Linking $@"
	$(CXX) $(CFLAGS) $(INCLUDE) -o $(INTERNAL_PATH)/AutoSimServer
$(OBJECTS) $(DEPEND) $(TINYXML) $(LIBS)

clientdepend:
	cd ../AutoSimClient-v0.03; $(MAKE) -f Makefile dependAutoSimServer

clean:
	@echo "*!*!*!*Cleaning..*!*!*!*"
	@/bin/rm -rf *.o $(INTERNAL_PATH)/AutoSimServer $(GENPATH)/*



-include $(SOURCES:.cpp=.D)
```

Table 9 : Linking, Cleaning and Including

In this final section are the rules for linking the binary object files into a complete executable. The rule 'clientdepend' executes a more simple script to build the components required of AutoSimClient. The last line is of interest because this directive includes all .D files generated from the dependency pre-processing command. Providing the dynamic list of explicit dependencies. This dependency map allows the script to compile and link the final executable easily.

7.2.5 Dynamic Makefile Conclusion

The objectives were to produce a build process that handles source code and dependency changes, to have a process that compiles objects and projects independently, to be specific so where problems occur they can be identified and to include the extra build steps, running Qt tools, dependency checking and building projects independently. These objectives have been met by a combination of specific explicit rules, and a dynamic approach of pre-processing the source to build rules so that the script can mostly modify itself as the project changes.

7.3 Improved Network Execution

One of the largest issues confronting AutoSim is the lack of performance in the simulation. Moving to separate the client and server processes on separate machines does help the overall throughput however refinement and reform of the approach taken to communicate over the network should yield results given the recent move away from the chain of responsibility singleton class structure to a more organised approach of state based control.

Benchmarking around the network classes is the easiest way to measure simulation throughput, as the code is highly predictable and subject to the performance of the rest of the application structure.

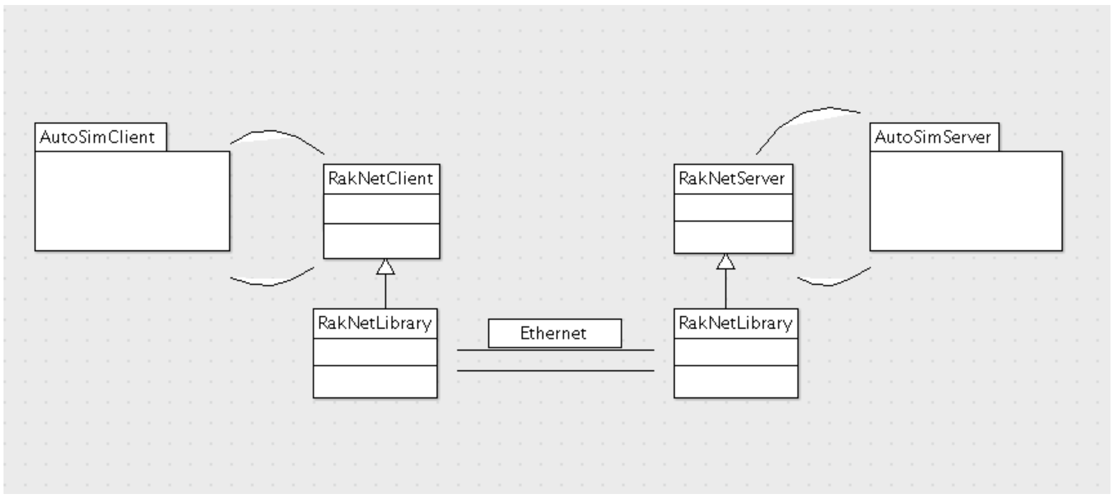


Figure 16 : Program control flow diagram.

In both the client and the server are implementations from an extension of the class RakPeer from the RakNet Library API. RakPeer is the main interface for network communications and contains all the major functions for the RakNet library. Both RakNatClient and RakNetServer are singleton classes in their own executable that also implement threading so that they can run in their own loop of execution. In this loop are functions to the rest of AutoSim that pass information updates and trigger code execution through the Singleton class chain of responsibility in order to update and maintain coordination throughout the system. Figure 16 shows the relationships between RakNetServer and RakNetClient classes to the rest of the project; they both call functions into the rest of the binary and program control returns to them. Which then loops over RakNet API calls checking for more data from the network. The class RakNetServer evaluates the passing of real time to ensure the simulation is not processed too quickly and in doing so limits the packet throughput.

When active the RakNetClient checks for a received packet as the active thread and new packet is received, it is de-serialised before being passed on for use in other singleton classes like GraphicManager. This acts as a chain of responsibility from between different types of request.

7.3.1 Networking Principles Theory

AutoSim uses a physics engine to handle interactions with the models in the simulation. To evaluate their movement physics engines adopt a Euler equation approach to integration of the realistic physical relationships they uphold [21].

To do so requires providing them with a delta of time to run the simulation over from the current state. This means the lower the delta of time the higher the precision but the longer the calculation time. The inverse is also true that the longer the delta the less ability to correct changes to the simulation and hence a loss of accuracy. The value designed for in AutoSim is a thirty millisecond time delta. This means that the resolution of the Simulation is thirty milliseconds and that no new information between the client and server is developed earlier than this time. The time the simulation states is known as a simulation tick. Computer cycles and time are referred to as tocks.

AutoSim networking stack is handled by the use of the Raknet Network Multiuser Network library. Raknet is open source UDP and C++ based network library designed to provide low response time with little overhead.

The simulation maintains a tree data structure with the simulation status information. It is serialised for storage and transmission of simulation data between the client and server.

The simulation processes the simulation in specific finite quantum of time or ticks; these ticks represent a specific amount of simulated time. In AutoSim the base tick time represents thirty milliseconds of simulation time. This is one simulation iteration and the physics engine. The process of communicating this evolution of the physics simulation and the position of models in AutoSim occur by the connection of an AutoSimClient to the Server. This occurs when the AutoSimClient establishes a connection through Raknet by sending a datagram to AutoSimServer. The server proceeds to modify simulation state information each tick, which is transferred by a serialised simulation tree as a message to the client in a datagram via Raknet and the network stack.

To assess the performance of the software attention was given to the process of communication between the two major executable, the client and server. In the Server, the responding thread is of class 'NetworkThread', this is defined as a wrapper for the class 'RakNetServer' which implements functionality for the communication with Raknet and code to process the serialised simulation information tree.

7.3.2 Network Performance Practise

The movement of work away from any unnecessary responsibility held by the network threads in both the client and the server is expected to help network threads adapt better when computer cycle conditions change. Outcomes cannot be determined by the subjective experience of the simulator as the simulation behaves differently when simulated changes are not reflected on by user input. The following section conducts to evaluate the code reform.

7.4 Experiments to evaluate outcomes.

It was clear from the sluggish response of AutoSim that the simulation execution was not occurring efficiently. To evaluate the usefulness of the implemented reforms a log was collected of events in the AutoSimServer RakNetServer class in the control loop, which governs the evolution speed. When events take place so that the simulation is executed more than the simulation step size apart, and event and time was recorded. The actual time was printed to a log and is pictured in the following performance chart, Figure 17.

The experiment was conducted using default settings and the 'residential_area.xml' world file. This allowed the simulation to drive the car straight forward through the town models and then out onto a space of empty grass fields with little input from the driver. Events in the single digits would be influenced by the connection of the client, and similarly events at the tail end of the simulation represent server load once the client has disconnect. Each time the same motion was completed by the experiment under the exact operating system conditions and experiment run was completed in two minutes. A number of trial runs were completed to ensure that the log used was a representative sample of the simulation performance.

These experiments were all conducted using the following hardware.

Processor	Intel Core Duo 2.0GHz
System Memory	2GB of DDR2
Operating System	Microsoft Windows XP SP3
DirectX Version	9.0c

Table 10 : Benchmarking Hardware Platform

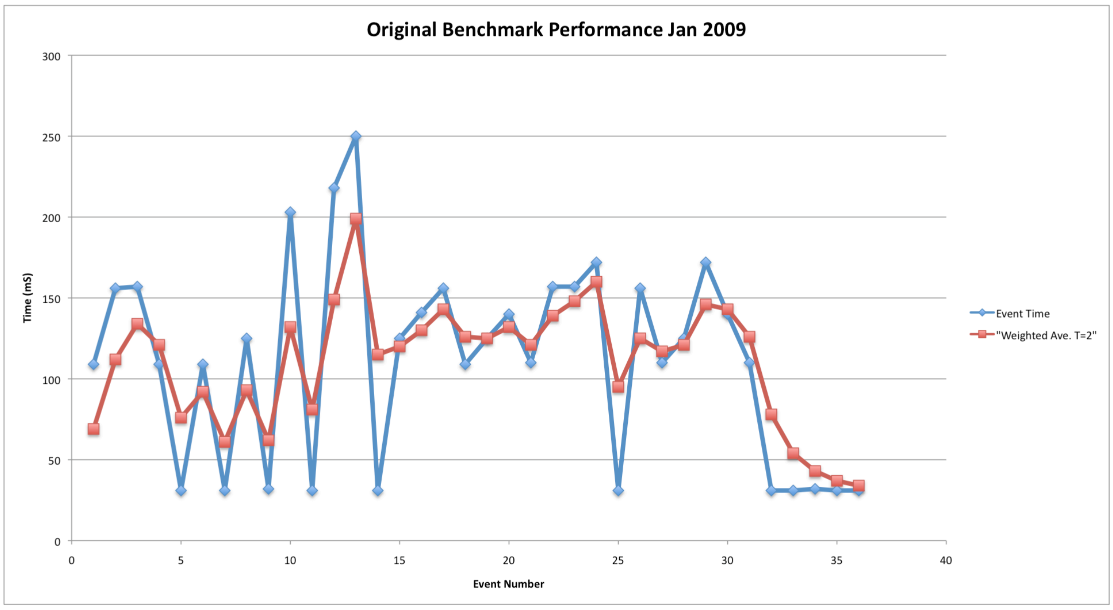


Figure 17 : Performance before implementation

Each event only occurs when the period between simulation ticks or iterations is over thirty milliseconds. This may be due to the length of execution time within the simulation governing control loop of AutoSimServer. These events are indicated as an overload reporting errors such as “Can Not Keep Up!” to the console.

The first series is the raw record of the simulation lag event. The second series is a weighted average over the last two records. From the graph it can be stated that the average lag event that occurs is one hundred milliseconds. While this does not seem like much, it should be remembered that each point on the graph represents successive lag events, and multiple events may occur together. This results in the user experiencing what is referred to as chugging as an apt comparison to a struggling and slowly moving motor engine.

In general the longest lag event occurred near the beginning of the simulation, and the lag events stopped when the client disconnected.

Figure 17 is the benchmark that this work aims to improve on.

7.5 Analysis of the Second Experiment

After the changes to code in the shift towards a state design pattern and the removal of work on the network thread. The exact context of the experiment trails was repeated. And a representative sample result created the following chart:

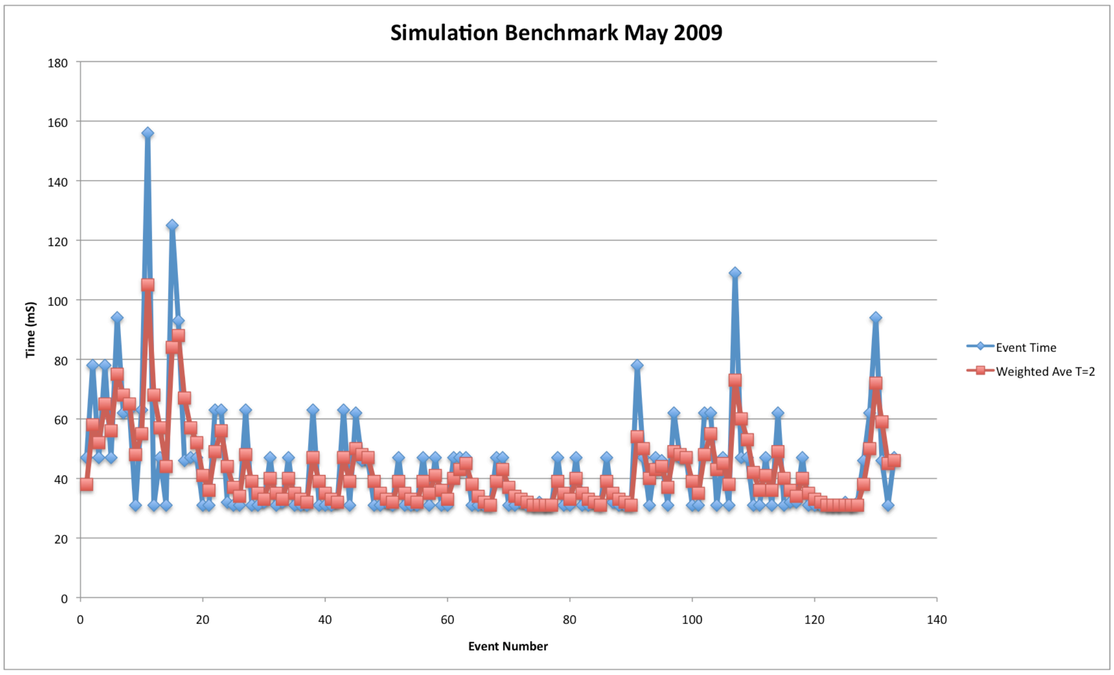


Figure 18 : Performance after implementation

In Figure 18 we can see a good reduction in the average intensity of lag events. The average time for a lag event is around forty-five milliseconds. The number of events has increased two and a half fold. The increased number of events may indicate that the simulation is actually completing more of the actual physical states between the start and finish of the experiment. Chugging appeared to be reduced. Figure 18 also represents well the early lag seen in the earlier experiment. And an unusual spike at the end of the experiment is an outlier but before that point the clear drop-off of events has occurred as before.

8 Conclusion

This thesis attempted to achieve three objectives, one to bring AutoSimServer to the Linux platform, to improve stability with the improvement of transitions to help prevent invalid internal states and all while improving the network performance of the simulation so that the simulation was more realistic. It has mostly achieved what it set out to do. AutoSim remains a large and complex piece of software however now the implementation now stands on two platforms with improved stability and more efficient processes for producing an automotive simulation.

9 References

[1] "Racer Car and Racing Simulator", Retrieved May 2009 http://www.racer.nl/

[2] "Euro Truck Simulator", Retrieved April 2009 http://www.eurotrucksimulator.com/

[3] "Robot Auto Racing Simulator", Retrieved February 2009 http://rars.sourceforge.net/

[4] "Welcome to The TORCS Racing Board", Retrieved February 2009 http://www.berniw.org/trb/index.php

[5] "Road fatalities and fatality rates - 1925 to 2003", Retrieved 10 September 2008 http://www.abs.gov.au/AUSSTATS/ABS@.NSF/Previousproducts/1301.0Feature%20Article302005?opendocument&tabname=Summary&prodno=1301.0&issue=2005&num=&view=

[6] Australian Bureau of Statistics, "Use of urban public transport in Australia", Retrieved 10 September 2008 http://www.abs.gov.au/Ausstats/abs@.nsf/90a12181d877a6a6ca2568b5007b861c/d81efef6e2252cf4ca256f7200833049!OpenDocument

[7] W. D. Jones, "Keeping Cars from Crashing," *IEEE Spectrum,* vol. 38, pp. pp. 40-45, Sept. 2001.

[8] S. Alles, C. A. Swick, M. E. Hoffman, S. M. Mahmud, and L. Feng, "The hardware design of a real-time HITL for traction assist simulation," *Vehicular Technology, IEEE Transactions on,* vol. 44, pp. 668-682, 1995.

[9] "Mobileye - Advanced Warning Systems", Retrieved January 2009 http://www.reverseinsafety.co.uk/advanced-warning-systems/mobileye.html

[10] Retrieved January 2009 http://www.toyota.eu/06_Safety/03_understanding_active_safety/03_cruise_control.aspx

[11] T. Guo, X. Wang, and G. Yu, "Virtual-Reality-Based Instrument Development in a Virtual Vehicle Simulation System," in *Electronic Measurement and Instruments, 2007. ICEMI '07. 8th International Conference on*, 2007, pp. 3-215-3-219.

[12] "OpenStreeMap", Retrieved http://www.openstreetmap.org/

[13] J. G. Brand, "Graphics for a 3D Driving Simulator," in *Center for Intelligent Information Processing Systems*: University of Western Australia, 2008.

[14] R. H. Erich Gamma, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Publishing Company, Inc, 1996.

[15] "Qt - A cross-platform application and UI framework", Retrieved February 2009 http://www.qtsoftware.com/

[16] "RakNet - Multiplayer game network engine", Retrieved October 2009 http://www.jenkinssoftware.com/
[17] T. Sommer, "Physics for a 3D Driving Simulator," University of Western Australia 2008.
[18] Pål Simen Ruud, "Evaluation of a Vision-
-based Driver Assistance System in Simulation," in *Centre for Intelligent Information Processing Systems (CIIPS)*: University of Western Australia, 2008.
[19] "GNU `make'", Retrieved February 2009 http://www.gnu.org/software/make/manual/make.html
[20] "sed, a stream editor", Retrieved March 2009 http://www.gnu.org/software/sed/manual/sed.html
[21] T. B. A. Boeing, ""Evaluation of Real-Time Physics Simulation Systems"," *Int'l Conf. Comp. Graphics Interaction Techniques Australia and Southeast Asia,* pp. pp. 281-288, 2007.

Appendix

9.1 AutoSimServer Makefile

```
##
## AutoSimServer Makefile
## Written to replicate the build process used in VS, to build under Linux
## Assumes all libraries are built beforehand this includes:
##	Raknet, PAL, Qt, Bullet, tinyXML
##  Author		:	Steven.John.Bradley+AutoSim@gmail.com
##
##	Requires: binutils
##

CXX=/usr/bin/g++
MAKE=/usr/bin/make
INTERNAL_PATH=$(CURDIR)/../release
CFLAGS=-D STATIC_CALLHACK -Wall -Wextra -ansi
##-pedantic
##-pedantic ##libraries headers are not compiled with pedantic and will fail
CONFIG=release

LIBDIR=$(CURDIR)/../../lib
QTDIR=$(LIBDIR)/Qt
RAKDIR=$(LIBDIR)/RakNet
PALDIR=$(LIBDIR)/PAL
XMLDIR=$(LIBDIR)/tinyxml
BULDIR=$(LIBDIR)/bullet

CLIENTDIR=$(CURDIR)/../AutoSimClient-v0.03
SERVERDIR=$(CURDIR)
```

```
##
##      Paths
##

INCLUDE=-I../AutoSimCommon -I$(RAKDIR)/Source -I$(PALDIR) -
I$(PALDIR)/pal_i -I./GeneratedFiles -I./GeneratedFiles/Release -I$(XMLDIR) -
I$(SERVERDIR) -I$(CLIENTDIR) -I$(QTDIR)/include -I$(QTDIR)/include/QtCore -
I$(QTDIR)/include/QtGui -I$(BULDIR)/src
##Include has a reference to AutoSimClient/Server source files but they should be
made mostly independent
LIB_PATHS=-L$(PALDIR)/lib/debug -L$(BULDIR)/src -L$(RAKDIR)/Lib/GNU-
Linux-x86 -L$(QTDIR)/lib
##-L$(XMLDIR)

##
##      Libs
##

##QTLIBS=-lQt3Support -lQtScript -lQtWebKit -lQtCore -lQtSql -lQtXml -lQtGui -
lQtSvg -lQtXmlPatterns -lQtNetwork -lQtTest -lQtOpenGL -lQtUiTools
QTLIBS=-lQtCore -lQtGui
LIB_NAMES=-lbulletcollision -lraknetd $(QTLIBS)
## -llibpal -ltinyxml
LIBS=$(LIB_PATHS) $(LIB_NAMES)
```

```
##
##	AutoSim Builds
##


all: AutoSimServer
	@echo "***** $< is now built ******"



##
##	Qt MOC and UIC
##

##Makefile path == $(CURDIR)
GEN=$(CURDIR)/GeneratedFiles/$(CONFIG)
GENPATH=$(CURDIR)/GeneratedFiles

MOC_FLAGS=-DUNICODE -DQT_THREAD_SUPPORT -DQT_CORE_LIB -DQT_GUI_LIB
-I$(CURDIR)/GeneratedFiles    -I$(QTDIR)/include    -I$(GENPATH)/$(CONFIG)    -
I$(CURDIR) -I$(QTDIR)/include/QtCore -I$(QTDIR)/include/QtGui

##	Using our own Qt's binaries prevents compatibility issues.
MOC=$(QTDIR)/bin/moc
UIC=$(QTDIR)/bin/uic

##	Typical System Locations wiht QT-Dev packages installed
##MOC=/usr/bin/moc
##UIC=/usr/bin/uic
```

```
##
## BUILDING QT -> UIC -> MOC
##
## Qt:
##      cd $(QTDIR); ./configure -prefix $(QTDIR) -static -qt-sql-sqlite -no-openssl <
yes && make -j 5
##      Remove -j 5 to stop parallel processing and enable clear reading of output
##      Example:        cd $(QTDIR); $(MAKE) -f Makefile
## ^^^ above is taken out, unnecessary, as long as Qt built beforehand
## if included add to the dependancies of MOC below

MOC_FILES : $(GENPATH)/ui_mainwindow.h\
        $(GEN)/moc_mainwindow.cpp\
        $(GEN)/moc_treemodel.cpp\
        $(GEN)/moc_serializabletree.cpp\
        $(GEN)/moc_messagelist.cpp\
        $(GEN)/moc_controller.cpp

MOC: $(GEN) MOC_FILES

$(GEN):
        @echo "*****Creating Qt Release Directory"
        mkdir $(GEN)

$(GEN)/moc_controller.cpp:
        @echo "****Moc'ing $@"
        @$(MOC)         $(MOC_FLAGS)        $(CURDIR)/Controller.h          -o
$(GEN)/moc_controller.cpp

$(GEN)/moc_messagelist.cpp:
        @echo "****Moc'ing $@"
```

```
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/MessageList.h -o $(GEN)/moc_messagelist.cpp

$(GEN)/moc_controller2.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/Controller2.h -o $(GEN)/moc_controller2.cpp

$(GEN)/moc_simulation2.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/Simulation2.h -o $(GEN)/moc_simulation2.cpp

$(GEN)/moc_serializabletree.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/SerializableTree.h -o $(GEN)/moc_serializabletree.cpp

$(GEN)/moc_treemodel.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/TreeModel.h -o $(GEN)/moc_treemodel.cpp

$(GEN)/moc_mainwindow.cpp:
	@echo "****Moc'ing $@"
	@$(MOC) $(MOC_FLAGS) $(CURDIR)/mainwindow.h -o $(GEN)/moc_mainwindow.cpp

$(GENPATH)/ui_mainwindow.h: $(GEN)/moc_mainwindow.cpp mainwindow.ui
	@echo "***Uic'ing $@"
	@$(UIC) -o $(GENPATH)/ui_mainwindow.h mainwindow.ui
```

```
##
##      Object Compiling
##

## All source files must be listed here, wildcards include new files automatically
SOURCES = $(wildcard ../AutoSimCommon/*.cpp) $(wildcard $(GEN)/*.cpp)
$(wildcard *.cpp)
## Static link libpal due to PAL breaking dynamic linking support for GCC

OBJECTS = $(SOURCES:.cpp=.o) $(PALDIR)/lib/liblibpal.a
MAKEDEPEND = $(CXX) -MM $(INCLUDE) -o $*.d $<



##
##      Required Objects from another location.
##
DEPEND =        $(CLIENTDIR)/RoadFactory.o\
                $(CLIENTDIR)/MapInfo.o\
                $(CLIENTDIR)/TerrainBuilder.o\
                $(CLIENTDIR)/node.o\
                $(CLIENTDIR)/OsmParser.o\
                $(CLIENTDIR)/RectNode.o\
                $(CLIENTDIR)/segment.o\
                $(CLIENTDIR)/hvector.o\
                $(CLIENTDIR)/kbspline2d.o
```

```
##
##	TinyXML library objects
##
TINYXML =	$(XMLDIR)/tinyxml.o	$(XMLDIR)/tinystr.o
$(XMLDIR)/tinyxmlerror.o $(XMLDIR)/tinyxmlparser.o

testing:
	@echo $(SOURCES)

%.o: %.cpp
	@echo "**Compiling Object $@ for $<"
	@$(MAKEDEPEND); \
	cp $*.d $*.D; \
	sed -e 's/#.*//'\
	 -e 's/^[^:]*: *//'\
	 -e 's/ *\\$$//' \
	-e '/^$$/ d' -e 's/$$/ :/' < $*.d >> $*.D; \
	rm -f $*.d
	@$(CXX) $(CFLAGS) $(INCLUDE) -c $< -o $@


##
##	Linking
##

AutoSimServer: MOC $(OBJECTS) clientdepend
	@echo "*Linking $@"
	$(CXX)	$(CFLAGS)	$(INCLUDE)	-o	$(INTERNAL_PATH)/AutoSimServer
$(OBJECTS) $(DEPEND) $(TINYXML) $(LIBS)

clientdepend:
	cd ../AutoSimClient-v0.03; $(MAKE) -f Makefile dependAutoSimServer
```

```
clean:
	@echo "*!*!*!*Cleaning..*!*!*!*"
	@/bin/rm -rf *.o $(INTERNAL_PATH)/AutoSimServer $(GENPATH)/*


-include $(SOURCES:.cpp=.D)
```

9.2 AutoSimClient Makefile

```
##
## AutoSimClient Makefile
## Written to replicate the build process used in VS, to build under Linux
## Assumes all libraries are built beforehand this includes:
## Assumes all libaries are built beforehand this includes: Raknet, PAL, Qt,
Bullet, tinyXML
## Author : Steven.John.Bradley+AutoSim@gmail.com
##
CXX=/usr/bin/g++
MAKE=/usr/bin/make
INTERNAL_PATH=$(CURDIR)/../release
CFLAGS=-Wall -Wextra -ansi
##-pedantic ##libraries are not compiled with pedantic and will fail
CONFIG=release

LIBDIR=$(CURDIR)/../../lib
QTDIR=$(LIBDIR)/Qt
RAKDIR=$(LIBDIR)/RakNet
PALDIR=$(LIBDIR)/PAL
XMLDIR=$(LIBDIR)/tinyxml
BULDIR=$(LIBDIR)/bullet

INCLUDE=-I../AutoSimCommon -I$(RAKDIR)/Source -I$(PALDIR) -
I$(CURDIR)/GeneratedFiles -I$(CURDIR)/GeneratedFiles/Release -I$(XMLDIR) -
I$(CURDIR)/../AutoSimServer-v0.03 -I$(CURDIR)/../AutoSimClient-v0.03 -
I$(QTDIR)/include -I$(QTDIR)/include/QtCore -I$(QTDIR)/include/QtGui
##Include has a reference to AutoSimClient/Server source files but they should
be made mostly independent

all: dependAutoSimServer
```

```
	@echo "*****$< has successfully compiled Objects for AutoSimServer"

##OBJECTS
dependAutoSimServer:  RoadFactory.o  depend-RoadFactory  MapInfo.o  depend-
MapInfo TerrainBuilder.o node.o depend-Node OsmParser.o

##	This indicates the depth of dependency of the Server on the Client
depend-MapInfo:			RectNode.o
depend-Node:			segment.o hvector.o
depend-RoadFactory:		kbspline2d.o

%.o: %.cpp
	@echo "**Compiling Object $(CURDIR) / $@"
	@$(CXX) $(CFLAGS) $(INCLUDE) -c $< -o $@

clean:
	@echo "*!*!*!*Cleaning..*!*!*!*"
	@/bin/rm -rf *.o
```