hempster hampster

a multiplayer car simulation engine

Supervisors

Magnus Bondesson

Computing Science Department, Chalmers University

Joakim Eriksson

Department of Design Sciences, Lund University

Eero Piitulainen

Magnus Österlind

ISRN: LUTMDN/TMAT-5071-SE

To our supervisors, past and present, for their infinite patience, thank you.

Abstract
This thesis describes the implementation of a multiplayer car simulator. The purpose of this work is to create a realistic car simulator framework. The framework can easily be extended to a platform for different test environments. Possible implementations could be behavior research and driver training.

Advantage has been taken of the continuing increase in computing power of personal computers to make the simulations of the vehicle dynamics as realistic as possible. The graphics engine takes advantage of the unprecedented increase in the capabilities of graphics cards over recent years.

1 INTRODUCTION 6

1.1 LAYOUT OF THE DOCUMENT 6
1.2 GOAL 6
1.3 RELATED WORKS 6
1.4 LIMITATIONS 7

2 METHOD 8

3 RESULTS 9

3.1 APPLICATION 9
3.2 MAIN LOOP 9
3.2.1 CONTEXTS AND TRANSITIONS 9
3.3 THE CLIENT-SERVER MODEL 11
3.3.1 CLIENT 11
3.3.2 SERVER 11
3.4 OTHER IMPORTANT COMPONENTS 11
3.4.1 ID DATABASE 11
3.4.2 GAME DATABASE 12
3.5 PHYSICS 13
3.6 STRUCTURE OF PHYSICS OBJECTS 13
3.6.1 DATA STORAGE 13
3.6.2 CURRENT OBJECTS DERIVED FROM CENTITY 13
3.6.3 ENTITIES AND ENTITY OWNERSHIP 13
3.7 MAIN PHYSICS LOOP 14
3.8 OBJECT CREATION 14
3.8.1 VEHICLE 14
3.8.2 CMYFIRSTCAR 14
3.8.3 CGEARBOX 14
3.8.4 CSIMPLEENGINE 14
3.8.5 CSUSPENSION 15
3.8.6 CPRISMATICSUSPENSION 15
3.8.7 FACTORY 15
3.9 WORLD SPACE SYSTEM 15
COLLISION SYSTEM 17
3.9.1 WORLD MESH STRUCTURE 17
3.9.2 COLLISION OBJECT STRUCTURE 17
3.9.3 COLLISION PRIMITIVES 17
3.9.4 COLLISION OBJECT HANDLE 18
3.9.5 CONTACTS 18
3.9.6 CONTACT KEY. 18
3.10 RIGID BODY DYNAMICS 19
3.10.1 CALCULATION OF BODY ACCELERATION 19
3.11 TYRE MODEL 20
3.12 PRISMATIC SUSPENSION MODEL 23

4 THE GRAPHICS ENGINE 25

4.1 INTRODUCTION 25

4.2 COMPONENT DESCRIPTION 25
4.2.1 WORLD 25
4.2.2 ANIMATIONMANAGER 25
4.2.3 CAMERA MANAGER 25
4.2.4 SHADERBREAKER 26
4.2.5 SCENE LOADER 26
4.3 3D OBJECT STRUCTURE 26
4.4 RENDERING 26
4.5 NOTABLE TECHNOLOGIES 27
4.5.1 SHADOW VOLUMES 27
4.5.2 ATMOSPHERIC SCATTERING 27
4.6 EXPORTING OBJECTS FROM 3DSTUDIO MAX 28

5 DISCUSSION 29

5.1 CONCLUSION 29
5.1.1 PHYSICS 29
5.1.2 COLLISION SYSTEM 29
5.1.3 NETWORK 29
5.1.4 ENGINE ONE 29
5.2 FUTURE WORK 29
5.2.1 PHYSICS 29
5.2.2 COLLISION SYSTEM 30
5.2.3 NETWORK 30
5.2.4 ENGINE ONE 30

6 REFERENCES 31

7 APPENDIX 32

7.1 BUILDING HEMPSTER HAMPSTER 32
7.2 EXTERNAL DEPENDENCIES 32
7.3 USERS GUIDE 33
7.4 USED ABBREVIATIONS 36
7.5 BLUEPRINT FORMAT 36
7.5.1 CAR ASSEMBLY 36
7.5.2 CAR 37
7.5.3 WHEEL WITH TYRE 38

List of Figures

FIGURE 1. APPLICATION MAIN LOOP 9
FIGURE 2. SINGLE PLAYER LOOP 10
FIGURE 3. MULTIPLAYER LOOP 10
FIGURE 4. ARRANGEMENT OF CELLS IN THE WORLD SPACE SYSTEM 16
FIGURE 5. RIGID BODY WITH ATTACHED REFERENCE SYSTEM, THE BODY FRAME. 19
FIGURE 6. LATERAL DEFLECTION OF TYRE. 20
FIGURE 7. SIDEVIEW OF TYRE, SHOWING THE TORSION OF THE CARCASS. 20
FIGURE 8. FORCE AND SLIPPAGE RELATIONSHIP OF TYRE. 22
FIGURE 9. MODEL OF A PRISMATIC SUSPENSION ATACHED TO A RIGID BODY. 24
FIGURE 10 IN GAME 25
FIGURE 11. DISPLAY OPTIONS 33
FIGURE 12. MAIN MENU 33
FIGURE 13. MAIN MENU 34
FIGURE 14. HOST GAME MENU 34
FIGURE 15. REMOTE GAME MENU 35

1 Introduction

1.1 Layout of the document

We begin by describing the goals, methods and limitations of Hempster Hampster (HH). A conclusion and final thoughts section rounds off the first chapter.

This is followed by a technical description of the three main parts of HH, the application, the physics, and the graphics engine.

Finally, an appendix containing instructions for building HH, along with a user's guide, and the format of the XML [2] files is presented.

Doxygen[1] generated documentation for the source code is included on the source code CD.

1.2 Goal

The goal of Hempster Hampster (HH) was to create a working multiplayer car simulator, that was easily extendable, both in terms of physics, logic and graphics.

We have implemented a simple multi player racing game as a proof of concept, but we think that the framework we have created can be used as the base of a lot more interesting projects.

Possible expansions include using the simulator as an aid for driving students and instructors etc. The simulation can also be expanded to conduct various types of research, such as, simulating different traffic environments, the effects of impaired peripheral vision [1].

The simulation could also provide a safe environment for otherwise dangerous tests, such as the effects of alcohol on a driver.

1.3 Related Works

There are a few other open-source game engines available, for example

- Crystal Space (http://crystal.sourceforge.net/tikiwiki/tiki-view_articles.php)
- Genesis 3D (http://www.genesis3d.com)
- Racer (http://www.racer.nl)

Despite this, we choose to implement everything in Hempster Hampster from scratch. The reasons for this choice are

- Various limitations in existing packages such as lack of control of numerical precision in large worlds.
- Control of the code base
- Many packages are geared mainly towards entertainment applications.

[1] http://www.stack.nl/~dimitri/doxygen/

1.4 Limitations

When we first drew up a list of features we wanted to implement in HH, it soon became apparent that this list was beyond the scope of the thesis.

Instead we focused on core functionality, while making sure the system was open ended, and could easily be extended in the future.

The core of Hempster Hampster consists of the following

- An application frame work, and a simple game model.
- A client/server implementation allowing for network play.
- A physics simulation
- A collision system
- A rendering engine, capable of rendering the car and the environment on different graphics cards.

The ideas and features we didn't implement are specified under each section's "Future Work" heading under conclusion.

2 Method

HH was coded in C++ under Microsoft Visual Studio .NET 2003. The 3d models were created in 3dsmax 5/6.0 and exported with a custom exporter. All parameters that are tweakable are stored in .XML files.

We choose Direct3D [4] as the graphics API to use, because it was the one we were most familiar with.

FMOD[2] was used for the sound playback. This made it possible to get a stable sound system running with very little effort.

We also used a number of free source code libraries to avoid reinventing several wheels. See the appendix for a complete list of libraries used.

When writing the code, we used a special commenting format to enable Doxygen to generate better source code documentation, along with class hierarchies and call graphs. The generated documentation is useful both in getting an overview of Hempster Hampster, and in the case of expanding upon this work.

[2] http://www.fmod.org/

3 Results

The HH program is split into three different parts, the application, the physics and the graphics engine.

3.1 Application

The application handles most of the “essential” functions, networking, updating inputs and sound and game logic. It also tells the physics to run the simulation and the 3d-engine to render.

3.2 Main Loop

The applications main loop is as follows (see figure 1):

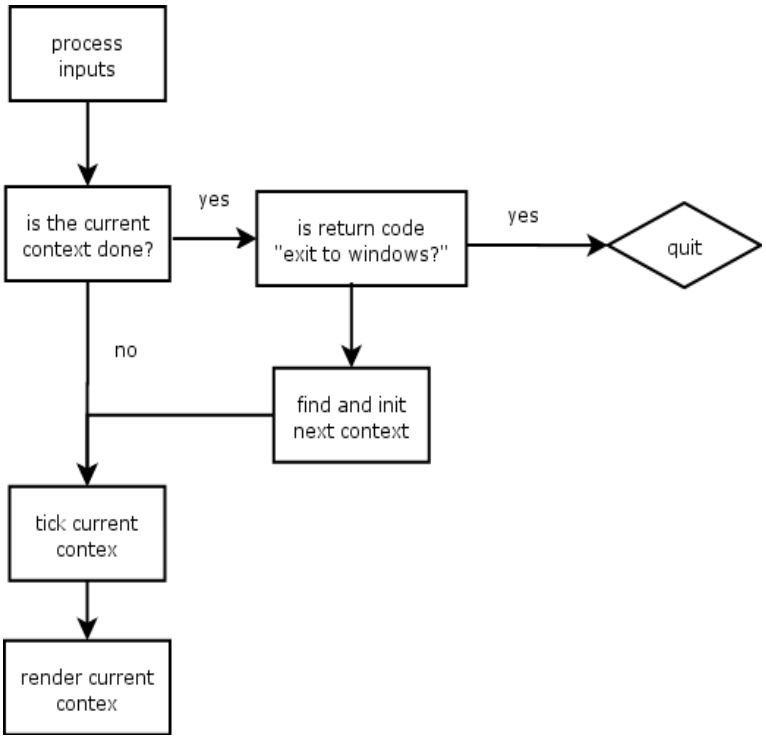

Figure 1. Application main loop

The application can be thought of as a state machine, and at any given point in time, is in exactly one of many different states (or contexts). Today “many” is just two.

3.2.1 Contexts and transitions

A context is a well defined state in the application. Each context has method to update (tick) and a method to render. The contexts also have a method for reporting if they are done (along with a return code).

Context Transitions
When a context reports that it is finished, it also returns a return code. This return codeis used to determine the next context.

Main Menu Context
When HH starts, the application is in the “Main Menu” context.

This context handles drawing the main menu, and handling the user input until the context exists. The main menu context can with either a “Quit game” return code, in which case we exit HH, or with the “Start” return code, which will transfer control to the “Game” context.

Game Context
The game context handles everything game related. Its update method ticks the simulation, processes network data and updates player information. Its render method calls Engine One’s World to render the environment.

Two slightly different loops are run, one for single player mode, and one for multi player (See figure 2 and 3).

The single player loop:

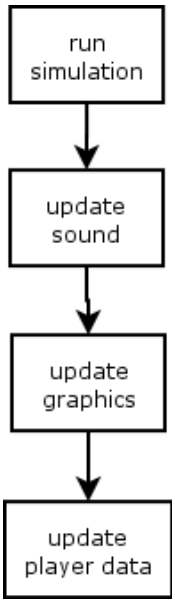

Figure 2. Single player loop

The multi player loop:

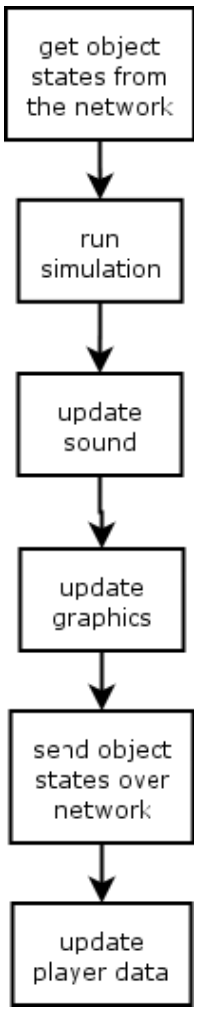

Figure 3. Multiplayer loop

3.3 The client-server model

HH implements a simple client-server model for the network play.

One player starts HH in host mode, selecting the track to be played, and waits for other players to connect using HH's remote game mode. When the players have connected, the one acting as server presses "start", and initial user and track data is sent to all the connected players.

Each players HH handles the simulation of the local player, and sends the local player's position to the server. The server sends each player's position to all the connected clients. There is thus no direct client to client communication.

3.3.1 Client

The client is only used when HH is connected to a remote game (i.e. the local application isn't acting as a server). Each frame, the client sends the local player's position to the server, and also to collects incoming player data, and sends it to the simulation (for collision detection, dead reckoning etc).

3.3.2 Server

The server acts as the hub, collecting all the player and race information and sending it on the connected clients. The server is also responsible for deciding who wins the race, and sending this data to the clients.

3.4 Other important components

Here we describe two other vital components used in the application.

3.4.1 Id Database

Because both the physics and the 3d-engine need to talk about the same objects, we need a way to ensure that both components have the same id for the same object. This is done via the id database.

Before an object is created, the id database's CreateIds method is called, which will create a number of ids for the specified object type. A car, for example, has five ids; one for the body, and one for each wheel. Components can then query the id database for a specific object id by sending in the objects name and group id[3].

[3] A group id is just a construct to be able to load several instances of the same object. Each player has a unique group id, so that several players can have the same car etc.

3.4.2 Game Database

The game database is a global store where the player and track data is kept. Having a central store saves us a lot of headache by ensuring that there is only one instance of important data to keep up to date. This also makes loading and saving a lot easier to implement.

3.5 Physics

The main physics class, CPhysics can be seen as the physical world. It contains all objects that reside in the world, as well as methods for accessing and manipulating these objects.

3.6 Structure of physics objects

All objects residing in the world derive from CEntity. This class contains identification and type information about the object. The entity has an interface that is used in the main physics loop for integration.

3.6.1 Data storage

All objects in the physics world that should use the following naming convention for internal structures that store runtime data.

SBlueprint
Contains all data needed to create the object, such as mass and inertia for a rigid body.

SState
The canonical state of the object. All redundant information can be resolved from this state together with the blueprint.

SAuxState
Redundant state information. If a property is frequently derived from the canonical state it can be added to this structure. This is mainly to avoid doing the same calculations multiple times.

3.6.2 Current objects derived from CEntity

Rigid body
An object that behaves as a rigid body, it is derived from an Entity.

MyFirstCar
This is a test class for a simple car. It is derived from a rigid body.
It can be controlled by user inputs.

Wheel
The wheel class is derived from a rigid body.

3.6.3 Entities and entity ownership

All references to all created entities are contained in a binary tree, called the entity map.
If an object, under local server control, is activated, it's added to the active entity list.
The physics engine is free to manipulate these objects without any restrictions.

An activated object that is not under the control of the local server is added to the non owned active entity list. These objects are also manipulated by the physics engine, mainly for dead reckoning purposes. Their states will however be replaced by incoming states from their respective owners.

3.7 Main physics loop

The physics state is progressed in a discreet time step, the size of which is set at the initialization of the world.

For each tick, the following is done in the given order:

- The state derivative is calculated for all active objects.
- All active objects are integrated.
- Collisions between objects and between the world mesh and objects are detected.
- Collisions are handled.

3.8 Object creation

Objects are created by calling the CreateObject function, and supplying object identification and state information.

The object database is used to find the blueprints required to build the object. The blueprints are sent to the object factory, which creates the objects and returns a reference to the object.

3.8.1 Vehicle

At the moment there is only one vehicle class, CMyFirstCar. It is however relatively easy to expand this to other types of vehicles.

3.8.2 CMyFirstCar

CMyFirstCar consists of:

- A finite number of suspensions. CSuspension.
- An engine. CSimpleEngine.
- A gearbox. CGearbox.

3.8.3 CGearBox

A gearbox has a finite number of gears, each with their own gear ratio, and a final drive. The velocity of the outgoing axis is engine velocity * gear ratio * final drive.

3.8.4 CSimpleEngine

This is a simple engine class, which will return an output torque as a function of engine speed and throttle position.

To determine this, it uses two torque curves, i.e. a table with output torque from the engine as a function of velocity. One of the curves is for full throttle, and the other is for zero throttle input. The final output torque is an interpolated value between the two curves, depending on throttle input and engine velocity.

3.8.5 CSuspension

CSuspension is an abstract base class for all supported types of suspensions. The suspension can be mounted on any rigid body. It contains functions for calculating the state derivative and to forward project the state, as well as functions for mounting a wheel onto the suspension.

3.8.6 CPrismaticSuspension

This is a simple suspension class derived from CSuspension. The mounted wheel can move along a prismatic joint. A spring and a damper are working on the wheel along the joint axis.

3.8.7 Factory

All physics objects are created in the physics factory, which is a set of public functions in the NPFactory namespace. The factory functions are called with a blueprint, and will return the objects created.

The blueprint is an XML document, containing information needed to create the object.
There are two types of blueprints.
The object blueprint, this type contains the definition for a specific object to be created.
The assembly blueprint, this is a collection of object blueprints with instructions on how to assemble the created objects.

Factory functions called with an object blueprint will return a pointer to the created object, while functions called with an assembly blueprint will return a list of created objects.

Each object type requires its own factory function. The function holds information on how to parse the XML file and how to fill out the SBlueprint struct for the created object.

Factory functions currently exist for these objects

- CMyFirstCar, created with BuildCar
- CWheel, created with BuildWheel.

Only the AssembleCar assembly function is currently supported. It assembles a CMyFirstCar and a finite number of wheels.

If new entity classes are created, factory functions that support the creation of them have to be implemented.

3.9 World space system

To be able to handle very large worlds and at the same time keep numerical precision errors at acceptable levels we have chosen to implement a world space system (WSS).

WSS is basically a grid of fixed size boxes in space, each containing their own local coordinate system (See figure 4). By representing points in the local frame of its container box, we can maintain an acceptable numerical precision.

A point in WSS is represented by a 32 bit index and a vector of three floats. The index can easily be converted to integer offsets in the direction of each axis. Thus points are conveniently transformed between the world frame and the local frame of a box or between two boxes.

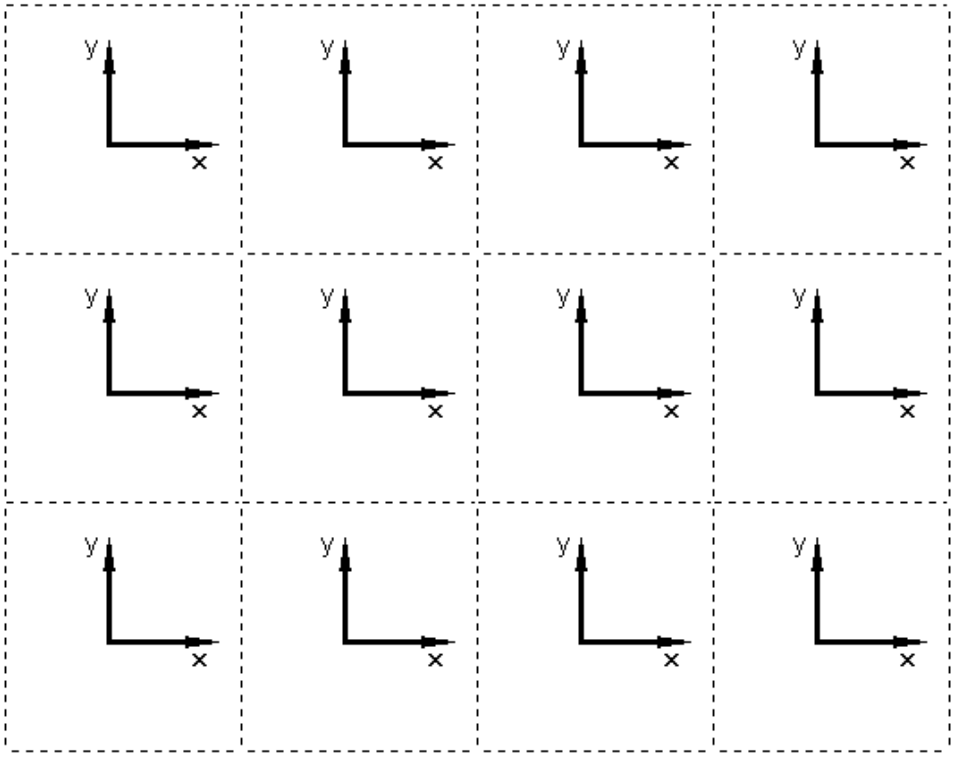

Figure 4. Arrangement of cells in the world space system

Collision system

The collision system contains all collision objects and performs collision tests between objects.

The system holds a static polygon mesh representing the world and dynamic objects consisting of collision primitives representing rigid bodies.

3.9.1 World mesh structure

A world chunk is a set of triangles all represented by the same WSS index.

The triangles in a world chunk are stored in a binary tree of axis aligned bounding boxes (AABB). A leaf consists of a bucket containing a fixed number of polygons.

All the world chunks are also stored in binary AABB tree, which makes up the world mesh.

When collision queries are made, the tree is traversed and tested against the AABB of the test object. A candidate set is retrieved and all shapes in the candidate set are then tested against the object.

3.9.2 Collision object structure

All collision objects are derived from the CColObj base class. It holds type and identification information about the collision object.

Currently we support the following collision types:

- Sphere — a simple sphere object.
- Sphere cluster — a cluster of spheres.
- Box — an object aligned box.

Additional types require implementation of new collision routines for the added types.

3.9.3 Collision Primitives

All collision objects are built by collision primitives. The collision primitives are simple geometric shapes.

Currently these types are implemented.

- Sphere — a point and a radius.
- Triangle — three vertices.
- Axis aligned bounding box — a max and min vector.
- Object aligned bounding box — a position, extent vector and a transform.

Collision tests

A brute force method is used at this time to test collisions between objects. All interactive objects are tested against each other. This is not very efficient method, but at this stage there are so few interactive objects in the world that it does not matter.

All interactive objects are also tested against the static world mesh, through bounding box tests and traversal of the AABB tree of the mesh.

3.9.4 Collision object handle

When the physics interacts with the collision system it does so via a handle to a collision object, CColObjHandle. Among other things the handle is used when registering and removing objects from the collision system. Object transforms are also updated via the handle.

3.9.5 Contacts

When a collision occurs a contact is created. The contact holds information about which objects collided, where the collision point is located and a collision normal.

3.9.6 Contact key.

To aid sorting of contacts, a contact key is created. The key is a 64 bit integer created by a merger of the indices of the colliding objects, with the lowest index first.

3.10 Rigid body dynamics

The rigid body is idealized as an object with a fixed mass and a fixed inertia.

The body is not deformable, i.e. the relations of points on the body are fixed in a reference system attached to the body. If not constrained the rigid body has six degrees of freedom. It can both move and rotate around all three axes of a reference system. Torques and forces acting on arbitrary points can be applied to the body.

To aid operations on the rigid body, a body frame is defined.

The origin of the body frame coincides with the body's center of gravity. The xyz-unit axes coincide with the three principal axes of inertia of the rigid body (See figure 5). In this frame the inertia matrix becomes a diagonalized matrix.

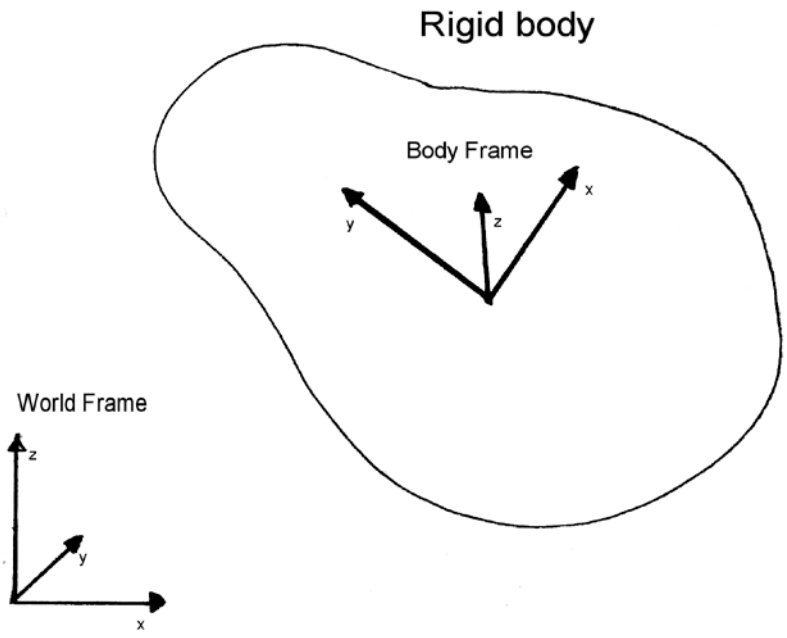

Figure 5. Rigid body with attached reference system, the body frame.

3.10.1 Calculation of body acceleration

The linear acceleration of the body is determined by a = F / m along all axes.
The angular acceleration is determined by Euler's equations. [5]

3.11 Tyre model

In our approximation of the tyre, all forces and torques are applied to the wheel via a spring system.

The system consists of a rigid ring, which approximates the carcass tread. If this ring is in contact with the ground there exists a contact point. The ring can be displaced with regard to the wheel in three ways, vertical-, lateral- and torsion-displacement (See figures 6 and 7). All these are attached to a damped spring and will result in reaction forces and torques.

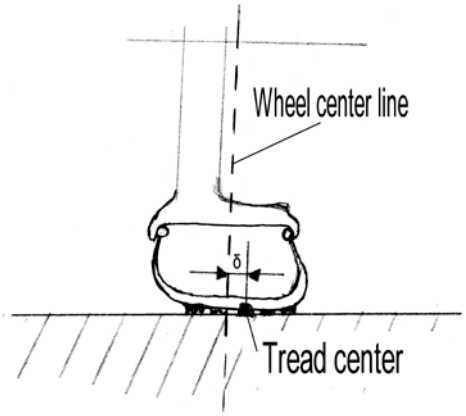

Figure 6. Lateral deflection of tyre.

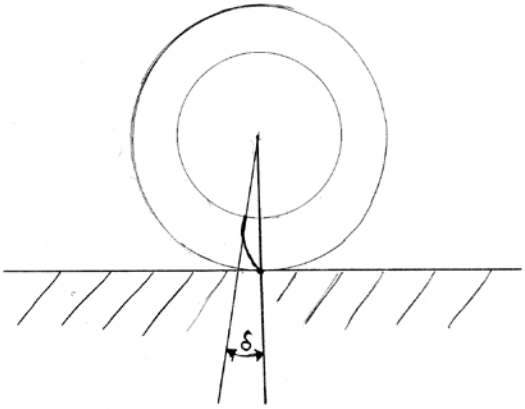

Figure 7. Sideview of tyre, showing the torsion of the carcass.

The system of springs governs the dynamic properties of the wheel and tyre. However they do not dictate the maximum force between the tyre to ground contact. This is governed by the steady state tyre model.

In the steady state model the forces on the tyre are a function of slippage. Two definitions are introduced. The lateral side slip angle (α) is the angle between the forward direction of the hub and the velocity direction of the hub. The longitudinal slip ratio (SR) is the ratio between the longitudinal velocity of the contact patch in relation to the ground and the longitudinal

velocity of the hub. The lateral force can be plotted as a function of slip angle and respectively the longitudinal force can be plotted as a function of slip ratio (See figure 8) [3]

Empirical tests have been carried out to gather data to better understand the properties of these relations.

Longitudinal forces
The tyre is studied under constant vertical load and changing slip ratio. For modest slip ratios the relationship between longitudinal force and slip ratio is linear.
$Fy = Ky * SR$
Where Fy is longitudinal force, and Ky is the traction/braking stiffness. If the slip ratio is increased the relationship will no longer be linear, and eventually the longitudinal force will reach a maximum. A continued increase of slip ratio will decrease the force and eventually move the tyre into a sliding state.

Lateral forces
The relationship between slip angle and cornering force is analogous to the relationship between longitudinal force and slip ratio. The cornering force is the lateral force exerted on the tyre.
$Fx = Kx * \alpha$
Where Fx is the cornering force and Kx is the cornering stiffness.

Two sets of forces and torques are generated in this scheme, one set for the spring model, and one set for the steady state model.

The steady state model is a closer approximation to reality during steady state motion[4], but does not work very well for sudden changes. Therefore the two are coupled together.

The forces transmitted by the spring model are clamped to not exceed the values obtained from the steady state model. This way the springs are the dominant model during transitional states[5], but as soon as a steady state has settled in the slip method is dominant.

[4] The monitored state does not change with time.
[5] The monitored state changes with time.

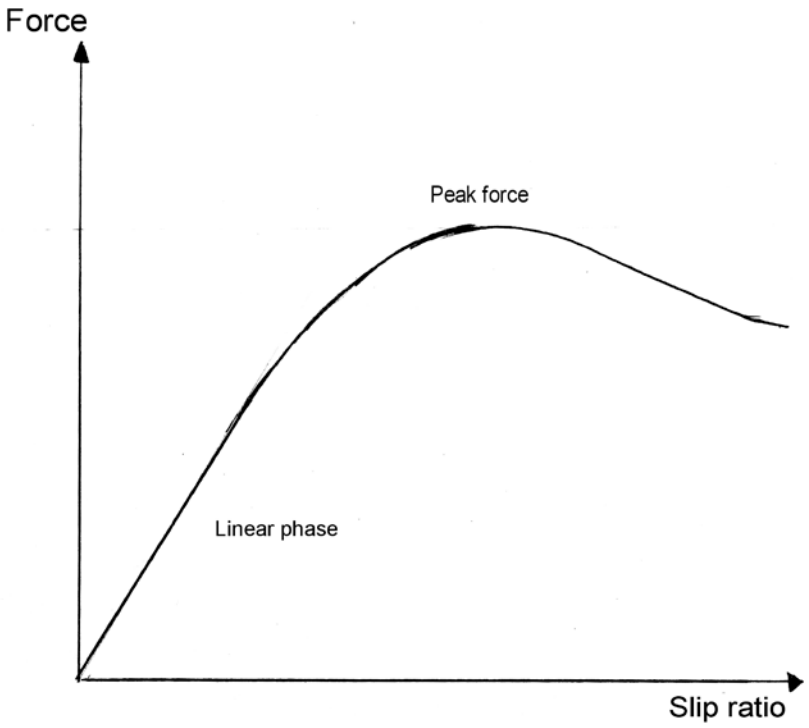

Figure 8. Force and slippage relationship of tyre.

3.12 Prismatic suspension model

This is a very simple suspension model.

Contrary to what its name might suggest, it is not a pure prismatic joint between the wheel and vehicle body.

In relation to the vehicle, the wheel can revolve around the hub axis and, in the case of a steered wheel, around the prismatic joint axis as well. The wheel can also move along the prismatic joint within certain limits. These constraints govern the position and orientation of the wheel in relation to the vehicle.

The motion of the wheel along the joint axis is governed by an imaginary constraint, the body roll center and body roll axis (See figure 9). The reason for not using the prismatic joint to constrain the motion is to allow the roll center to be altered, without introducing more complicated suspension models. The relation between the roll centers of the front and rear axles can be used to alter the balance of the car, i.e. making it understeer or oversteer.

All torques and forces applied to the wheel which are not forming a torque around the body roll axis are directly transmitted to the vehicle. The remaining forces and torques accelerate the wheel in relation to the vehicle body. These are used for solving the equations of motion of the wheel.

Mounted along the joint axis is spring damper system, which also exerts forces on the wheel. The spring is a simple spring with a relaxation length, and a constant stiffness.

The dampers have two coefficients of damping, one value for negative velocities along the axis, bump damping, and another for positive velocities, rebound damping. The right relationship between these is important in achieving a smooth ride over irregularities in the road surface.

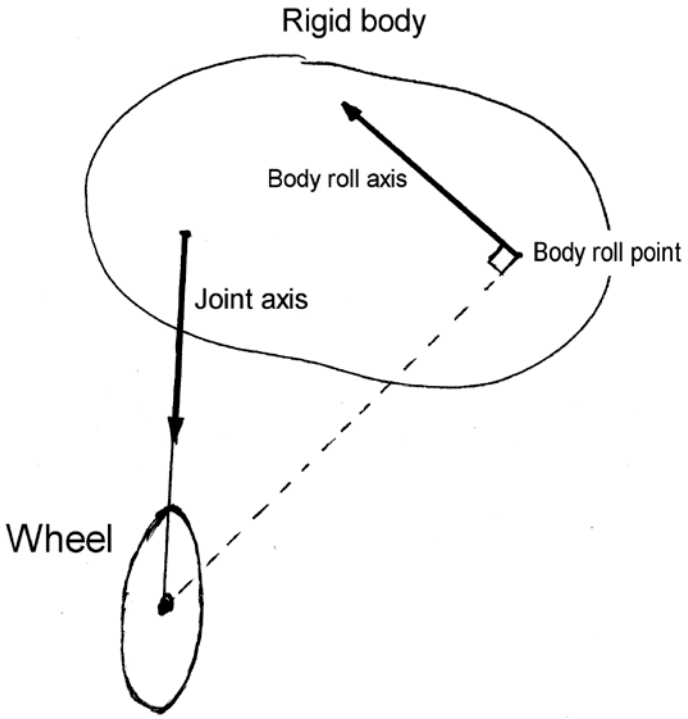

Figure 9. Model of a prismatic suspension atached to a rigid body.

4 The Graphics Engine

4.1 Introduction

The graphics engine, called Engine One, is the rendering and animation part of HH. It takes care of the loading, updating and rendering of all 3d objects in HH.

4.2 Component description

Next follows a description of the principal components of Engine One.

4.2.1 World

The world is the main engine. It contains the shaders and objects, and has public methods for updating and rendering. All external manipulation of the 3d-engine is also done through the World.

Figure 10 In game

4.2.2 AnimationManager

The animation manager serves as the link between the 3d objects and the “real world”.

The objects per se don’t have any code for updating their own positions in the world, instead they rely on the animation manager to do this. When the objects are created, they are given an animation handle and every frame, the objects use this handle to ask the animation manager for their current transformation.

By gathering all the transformation data in one place, it becomes trivial to gather data both from the network and from the simulation, and the objects don’t have to know how they were modified.

4.2.3 Camera Manager

The camera manager contains the cameras, along with logic for sending user input to the cameras.

There are two types of cameras in HH, free fly and target cameras.

Free fly cameras take user inputs and move accordingly. Free fly cameras can also be locked in a certain position.

Target cameras are created with function pointers to functions returning a position, a look at and an up vector. At each frame, the camera calls these functions and gets new data. By connecting the position and look at to the position stored in the player data, we can easily create a camera that follows the players car.

4.2.4 Shaderbreaker

A shaderbreaker is a combination of a shader (read from an .FX file) and the face blocks that use that shader. All the setting of parameters, textures etc needed for rendering is taken care of by the shaderbreaker. For more information, see the truly excellent article in ShaderX3 [6].

4.2.5 Scene Loader

The scene loader handles the loading of scenes, and adding the loaded objects to the world.

The file format is node based, where each node has a super type and a sub type. The super type specifies the type of node in general, for example “geometry”, “shape”, “light”, while the sub type has more detail: “tri_mesh”, “poly_line”, “directional”.

To use the loader, we specify callback methods that handle the particular type of nodes we’re interested in.

4.3 3D Object Structure

All 3d entities (including cameras and dummy[6] objects) in HH derive from a common base class, CNode. CNode contains transformations between local (object) and world space, as well as hierarchy information, and handles for referencing animation data.

Objects containing actual renderable geometry are called GeomObjects. GeomObjects contain the actual vertex and index buffers used for rendering, along with bounding volumes for culling, and eventual shadow geometry.

Because a single object in 3dstudio max can have multiple materials, each GeomObject in Engine One has one or more face blocks, where each face block is a collection of faces that share the same material.

4.4 Rendering

To get proper interaction between lights and shadows, the objects are rendered in several passes [7]:

- Render the ambient pass of all the shaderbreakers. This in turn calls the render method of all the visible face blocks using that shader.
- For each light (we only have a single directional light in HH at the moment)

[6] A dummy object is an object with no geometry, but it just serves as a node in an object hierarchy.

    - Render the shadow volumes into the stencil buffer for all objects, using the current light
    - Add the diffuse and specular contribution for each shaderbreaker for the current light, using the stencil buffer to mask off pixels that aren't lit.
- Render the transparent parts of each shaderbreaker.

4.5 Notable technologies

4.5.1 Shadow Volumes

To achieve real time shadows, we use the shadow volume technique. Shadow volumes work by extruding a volume based on the silhouette of the object from the light source, along the light direction. By counting how many times a ray from the eye to a pixel on screen enters and leaves the shadow volume, we can determine if that pixel is shadowed or not.

To solve the case where the camera is inside the shadow volume, we use the technique commonly know as "Carmack's reverse".

Links
http://developer.nvidia.com/object/robust_shadow_volumes.html

References
"Real Shadows, Real Time", Tim Heidmann, *IRIS Universe, #18*
"Shadow Algorithms for Computer Graphics", Frank Crow, *Proceedings of Siggraph 1977*

4.5.2 Atmospheric Scattering

In order to get a more realistic lighting model, we use a model called atmospheric scattering.

This model takes into account the interaction between particles in the air and the light heading towards the eye. The model gives us two terms, a multiplicative and an additive.

The multiplicative term, called the extinction term, tells us to what degree the particles either absorb the light, or reflect (scatter) it another direction, hence reducing the amount of light that reaches the eye.

The additive term, called inscattering, is the amount of scattered light that has been reflected into the current viewing direction, and is added to the final pixel color.

Links
http://www.ati.com/developer/demos/r8000.html

References
"Rendering Outdoor Light Scattering in Real Time", N. Hoffman, A. Preetham, *Proceedings of Game Developer Conference 2002*

4.6 Exporting objects from 3dstudio max

Exporting objects and tracks from 3dstudio max should not cause too many problems, but there are a few caveats.

Scale

When exporting, the scale used is one unit equals one meter. Also, don't forget to use the "Reset Xform" button before exporting. This will propagate all transformations stored in the objects transformation matrix onto the actual vertices. This is needed because some objects receive transformation matrices from the simulation, and will thus loose any previous transformations.

Required Objects

Car

When exporting a car, the following objects need to be present:

- Body, the body of the car
- Wheel_FL, Wheel_FR, Wheel_RL, Wheel_RR, the front left, front right, rear left and rear right wheels of the car.
    - Optional shadow meshes. Any object starting with "s_" will be treated as a shadow mesh, and be used in the shadow volume extrusion. Shadow meshes should be child objects under the meshes they represent.

Neither the body nor the wheel objects should have any parents, as the transforms for these objects are in world space.

Track

The following objects must be present for a track to be exported correctly:

- DriveLine, a poly line (called "DriveLine") representing the actual track. This is used to track the progress of the cars. The first knot of the line should be just after the starting point. The line must also be closed
- Starting positions. Point helpers that mark the starting position of the players. These must be called "startposXX" where XX goes from 00 to the number of players allowed.
- A sky dome. This is just a simple sphere enclosing the whole track, called "Sky".

Setting the shaders

Texture mapped and diffuse shaded objects are exported correctly from 3dstudio max.

To get the correct shader on the sky dome, the track's .ini file should be edited. The FaceBlock0_Shader part of the sky should be set to SkyDome (this will make the sky dome object use the SkyDome.fx shader).

```
[Sky]
FaceBlock0_Material = 13 - Default
FaceBlock0_Shader = SkyDome
```

5 Discussion

5.1 Conclusion

5.1.1 Physics

A basic system for physics simulation is in place and works as intended. Expansion of new types of objects should be rather straight forward. Existing packages such as ODE (http://www.ode.org) were evaluated. Most packages however are mainly geared towards rigid body dynamics, and multi body dynamics. In a vehicle simulator the rigid body dynamics is only a small part of the total simulation. With this in mind together with other issues, such as control of numerical errors in large worlds, we came to the conclusion to create a completely new physics system.

5.1.2 Collision system

A basic collision system has been implemented, and works as intended. Again the need for control of numerical errors in large worlds made us implement our own system.

5.1.3 Network

DirectPlay was a pain to get working. A lot of the problems had to do with unclear documentation and samples. In hindsight, we should probably have evaluated RakNet (http://www.rakkarsoft.com) or HawkNL (http://www.hawksoft.com/hawknl/) first.

5.1.4 Engine one

We're quite happy with the structure of Engine One. It does what it's supposed to do, in a way that balances generality with efficiency, and should be easy to expand upon when the need arises.

5.2 Future Work

Possible expansions include using the simulator as an aid for driving students and instructors etc. The simulation can also be expanded to conduct various types of research, such as, simulating different traffic environments, the effects of impaired peripheral vision [Sara04].

The simulation could also provide a safe environment for otherwise dangerous tests, such as the effects of alcohol on a driver.

5.2.1 Physics

More accurate suspension models, such as MacPherson struts and double A-arm could be implemented. Camber effects and velocity effects could be added to the tyre model.

5.2.2 Collision system

Use of temporal coherence could be used to speed up collision queries. More primitives and more complex object types should be added.

5.2.3 Network

The network code should be made more robust, being able to handle disconnected players, players connecting mid game etc.

Stand alone server mode, were the server doesn't have its own car, but only acts as an observer should also be added.

Synchronized starting of races. At the moment, each player is able to drive when his track and car are loaded, but there should really be some kind of synchronization going on here.

5.2.4 Engine one

The first thing that comes to mind is more specialized shaders. Shaders for car paint and glass, along with shaders for the road surface (complete with bump mapping) should be straight forward to implement and add.
Handling lost devices is also something that should be handled gracefully.

6 References

[1] S. Rowell, Perifera seendets rörelsedetektion i trafiksimulator TSim, 2004
[2] Don Box, *Essential XML*, 1999
[3] G. Genta, *Motor Vehicle Dynamics – Modeling and Simulation*, World scientific publishing 1998
[4] K. Gray, *DirectX 9 Programmable Graphics Pipeline*, 2003
[5] J.L Meriam, *Engineering Mechanics - Dynamics*, Wiley 1998
[6] "Shader breaker", Magnus Österlind, *ShaderX3*, 2004
[7] A. Watt, 3D Computer Graphics, 2000

7 Appendix

7.1 Building Hempster Hampster

Hempster Hampster compiles “out-of-the box”, providing that DirectX Summer Update 2004 is installed (included on the CD).

Just open the solution file Server.sln, under HempsterHampster/Server, and build.

7.2 External Dependencies

To avoid having to reinvent several wheels, Hempster Hampster uses a few external libraries (all of which are included on the CD):

Boost
http://www.boost.org
A collection of invaluable C++ templates and classes, smart pointers, signal systems, type safe formatting etc.

Loki
http://www.moderncppdesign.com/
Andrei Alexandrescu’s library of useful stuff, singletons, factories etc.

Crazy Eddies GUI System
http://crayzedsgui.sourceforge.net/
An open-source, cross platform GUI system. Under development, but pretty stable, none-the-less.

TinyXML
http://www.grinninglizard.com/tinyxml/
A simple, small and free C++ XML parser. Just four files needed to be added to the project, and we had XML support.

7.3 Users guide

When HH is first started, the user is presented with a display options screen. It usually suffices just to press enter, using the default settings.

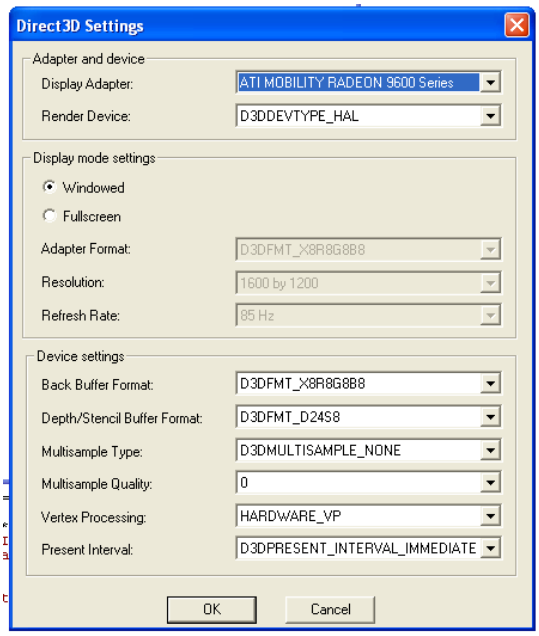

Figure 11. Display options

The main menu is pretty intuitive.
Choose “Start Local Game” to play by yourself.
“Host Game” if you’re going to be the server, and “Remote Game” to connect to someone who’s already hosting a game.
“Quit” will exit back to Windows.

Figure 12. Main menu

To play a local game, just select the car and track, and press start.

Figure 13. Main menu

To host a game, select the track to play, and the car you want to use. Press “Accept Players” to start hosting a game. As remote players connect, their names will appear in the “Connected Players” list. When everyone has connected, press the “Start” button to start the game.

Figure 14. Host game menu

To connect to a remote game, choose the car you want (along with your player name), and press “Enum Servers” to look for servers hosting games. As servers are found, they appear in

the servers list. To select a server, highlight it, and press connect. Note that it might take a while (around 30 seconds) to look for servers.

Figure 15. Remote game menu

7.4 Used abbreviations

HH	–	Hempster Hampster

7.5 Blueprint format

Here follows examples of supported blueprints. With blueprint syntax in verbatim and comments in italic.

7.5.1 Car assembly

```
<?xml version='1.0'?>
<HH>
<Blueprint type="assembly" name="CarAssembly1">

          <Car name="Body" file="Cars/Blueprints/p917.xml"/>

          <Wheels>
                    <Wheel
                    name="Wheel_FL"
                    file="Cars/Blueprints/WheelFront.xml"
                    mount ="Left Front"
                    />

                    <Wheel
                    name="Wheel_FR"
                    file="Cars/Blueprints/WheelFront.xml"
                    mount ="Right Front"
                    />

                    <Wheel
                    name="Wheel_RL"
                    file="Cars/Blueprints/WheelRear.xml"
                    mount ="Left Rear"
                    />

                    <Wheel
                    name="Wheel_RR"
                    file="Cars/Blueprints/WheelRear.xml"
                    mount ="Right Rear"
                    />
          </Wheels>

</Blueprint>
</HH>
```

7.5.2 Car

```
<?xml version="1.0" ?>
<HH>
    <Blueprint id="p917">
        <MyFirstCar name="Porsche 917">

            <RigidBody
                      mass="1000.0"
                      inertia="900.0 300.0 1000.0" />

            <Engine
                      torque="700.0"
                      maxspeed="700.0" />

             <Gearbox autogears="1" />

             <Suspension>
                 <Prismatic
                       name="Left Front"
                       vJointMount_VRF="-0.78 1.22 0.25"
                       vJointAxis_VRF="0.0 0.0 -1.0"
                       vRollPoint_VRF="0.0 1.42 0.3"
                       vRollAxis_VRF="0.0 1.0 0.0"

                       DefaultRotation="3.14659265"

                       MaxSteeringRotation="0.2"
                       Drive="0.0"
                       Brake="1.0"
                       HandBrake="0.0"
                       BrakeTorque="1500.0"
                       Inertia="1.0"
                       Mass="30.0"
                       SpringLength="0.3"

                       SpringStiffness="50000.0"
                       DampingBump="3000.0"

                       DampingRebound="5000.0" />

                 <Prismatic … />
                 <Prismatic … />
                 <Prismatic … />

             </Suspension>
        </MyFirstCar>
    </Blueprint>
</HH>
```

7.5.3 Wheel with tyre

```
<?xml version='1.0'?>
<HH>
<Blueprint id="FrontTyre">
          <Wheel
                    name                    ="Front Wheel"

                    Width                   ="0.2"
                    HubOffset  ="0.0"
                    Radius                  ="0.3"
          />

          <Tyre
                    name                    ="A Front Tyre"
                    Radius                  ="0.3"
                    LateralStiffness        ="2.3e5"
                    VerticalStiffness       ="2.0e5"
                    TorsionalStiffness      ="3.0e4"

                    MyLat                   ="1.4"
                    MyLong                  ="1.5"
                    MySlide                 ="1.2"

                    SlipPeakLat             ="0.2"
                    SlipSlideLat            ="0.5"
                    StiffnessLat            ="3.5e4"

                    SlipPeakLong            ="0.15"
                    SlipSlideLong           ="0.5"
                    StiffnessLong           ="8.0e4"
          />

</Blueprint>
</HH>
```