T.O.R.C.S.
Manual installation
and
Robot tutorial

Author by:

Bernhard Wymann

Transcription:

Gonzalo Antonio Aranda Corral

Contents

Chapter 1

TORCS Installation

This section will guide you through the download and installation of TORCS 1.2.4 from the sources on Linux (required for the robot tutorial). You can simply follow the links on the bottom of the pages. Installation instructions for the Linux and Windows binaries are available from the official TORCS site. If you get stuck, don't hesitate and mail to the torcs-users mailing list. There is also an archive of the mailing list. If you have comments or suggestions for improving the installation instructions, send me a mail.

1.1 Conventions

During the installation you have to run several commands in a shell. If a command starts with a hash (#) you should run this command as superuser (root), if it starts with a $ you should run it as normal user.

1.2 Hardware Requirements

You need a 3D-accelerator with OpenGL support for your platform. I recommend a CPU with 800 MHz or more, 256MB RAM and e.g. a Nvidia GeForce 2MX AGP or better (or similar ATI, Kyro,...). Below I comment some systems where I tested TORCS so that you can get a feeling what works well and what does not. Your z-buffer should have a depth of 24 bits or more to avoid glitches.

1.2.1 K6-III-400

K6-III-400MHz (66MHz memory bus) with 256MB RAM, motherboard Asus TX-97E, graphics card Nvidia Riva TNT with 16MB RAM (PCI). I ran it with 16 bit color depth and came to the following conclusions: This system is ok to practice without opponents (8-45 frames per second). Running races with robots alone is possible, but driving manually is difficult because the 6-20 frames per second (with 4 opponents) are not enough for a smooth ride. So the system is the bare minimum sufficient for developing your driver (robot).

1.2.2 SMP-PIII-550MHz

Two PIII-550MHz (100MHz memory bus) with 512 MB RAM, motherboard Asus P2B-DS, graphics tested with GeForce DDR, GeForce 2 GTS, GeForce 3. I ran it with 32 bit color depth. TORCS runs almost as good on the GeForce DDR like on the GeForce 3, so you don't need the latest and greatest graphics card. What is more important is working AGP, recent drivers and a "fast" CPU. TORCS runs just a single process (no threads), so it doesn't take advantage of SMP systems. In practice mode, I get more than 100 frames per second and in races with 20 cars 10-50 frames per second. This system works very well for developing a driver (robot) and to drive manual races with up to 10 opponents.

1.2.3 PIII-800MHz

PIII-800MHz (100MHz memory bus) with 512 MB RAM, motherboard Asus P2B, GeForce 2 GTS, 32 Bit color depth. Runs almost always smooth above 20 frames per second, so with 800MHz or more you should be able to enjoy racing.

1.3 Software

You need a working OpenGL/DRI driver, development tools and some additional libraries. Most often you can go into your Linux distribution setup and simply click something like ”development machine” and all necessary tools will be installed. You need gcc, header files for OpenGL, GLUT, GLU, XFree86 and libc. OpenGL driver setup is most often supported by the distribution, consult the documentation. Below we will check if the basic requirements are fullfilled. If you have already compiled OpenGL applications and you are sure your hardware is set up correct, skip the rest of this page.

1.3.1 Checking OpenGL/DRI

Start your XFree86 (if not already done), open a terminal and run as normal user (the $ means the prompt of a normal user, # the prompt of root).

```
$ glxinfo | grep direct
```

The result should look like this:

```
direct rendering: Yes
```

If that’s not the case you have to check your OpenGL setup.

1.3.2 Checking GLUT

For rpm based distributions, run the following command (we run a query over the whole package database):

```
$ rpm -qa | grep glut
```

The result should look something like this (this depends on the package names):

- mesaglut-3.4.2-42
- mesaglut-devel-3.4.2-42

If the result lists nothing you have to install GLUT. Use the tool provided from your distribution. Make sure that you also install glut.h. If just one package shows up, you can list the package content and search for glut.h:

```
$ rpm -ql NAME | grep glut.h
```

where NAME means in the above example mesaglut or mesaglut-devel, without version numbers. If nothing shows up, glut.h is not installed, so search your distribution and install it.

1.3.3 Checking Libpng

For rpm based distributions, run the following command:

```
$ rpm -qa | grep png
```

The result should look something like this (this depends on the package names):

```
libpng-2.1.0.12-153
```

If the result lists nothing you have to install libpng with header files (eg. png.h).

1.4 TORCS Packages

1.4.1 Required Packages

The following packages are required to follow the tutorial (download from here):

- TORCS-1.2.4-src.tgz
- TORCS-1.2.4-data.tgz
- TORCS-1.2.4-data-tracks-road.tgz
- TORCS-1.2.4-data-tracks-dirt.tgz

- TORCS-1.2.4-src-robots-base.tgz
- TORCS-1.2.4-data-cars-extra.tgz
- TORCS-1.2.4-data-cars-nascar.tgz

1.4.2 Optional Packages

There are also optional packages available with additional cars, robots and tracks. I recommend to download them all, of course.

- TORCS-1.2.4-src-robots-berniw.tgz
- TORCS-1.2.4-src-robots-bt.tgz
- TORCS-1.2.4-src-robots-olethros.tgz
- TORCS-1.2.4-data-cars-Patwo-Design.tgz
- TORCS-1.2.4-data-cars-kcendra-gt.tgz
- TORCS-1.2.4-data-cars-kcendra-roadsters.tgz
- TORCS-1.2.4-data-cars-kcendra-sport.tgz
- TORCS-1.2.4-data-cars-VM.tgz
- TORCS-1.2.4-data-tracks-oval.tgz

You have to be careful with the packages from Patwo-Design, kcendra and VM, they are for free, but not GPL. So you can use the package for free, but you are not allowed to change it. License details are included in the readme.txt of the cars. That means if you develop a robot which is driving such a car, you are not allowed to change the texture.

1.5 Plib Installation

1.5.1 Checking Plib

First we check if there is already the required plib version installed (TORCS 1.2.4 has been tested with plib version 1.8.3):

```
$ rpm -qa | grep plib
```

In case we find plib 1.8.3 everything is fine and you can skip the rest of this section. If you find plib in any other version (e. g. 1.4.2., 1.5.0 or 1.6.1) you have to remove the packages:

plib-1.4.2-30 plib-examples-1.4.1-32

Remove first the plib-examples package and then plib (you need to be root):

```
# rpm -e plib-examples
# rpm -e plib
```

There is no danger in removing plib, because it contains just static libraries, so if something goes wrong you can force the remove.

1.5.2 Download and Unpack Plib

Download plib-1.8.3.tar.gz from here. TORCS has been tested against 1.8.3, for other versions we guarantee for nothing! Stay as root and do

```
# cd /usr/src
# mkdir torcs
# cd torcs
# tar xfvz /path_to_downloaded_files/plib-1.8.3.tar.gz
# cd plib-1.8.3
```

1.5.3 Compiling and Installing Plib

You are still root and in /usr/src/torcs/plib-1.8.3, if you run a 64 bit version of Linux export the following variables:

```
# export CFLAGS="-fPIC"
# export CPPFLAGS=$CFLAGS
# export CXXFLAGS=$CFLAGS
```

Now for all platforms run:

```
# ./configure
# make
# make install
```

Just to play safe clear the above defined variables:

```
# export CFLAGS=
# export CPPFLAGS=
# export CXXFLAGS=
```

If something fails you need to resolve it. In case the configure script complained about something you can find some additional information in the file config.log. The cause for problems are usually missing header files or libraries, wrong versions or multiple versions installed.

1.6 OpenAL Installation

1.6.1 Checking for OpenAL

First we check if there is already an openal version installed.

```
$ rpm -qa | grep -i openal
```

In case you find an installed version uninstall it. Remove first the openal-devel package and then openal (you need to be root):

```
# rpm -e openal-devel
# rpm -e openal
```

There is a certain danger in removing OpenAL, because it is a shared library. If the package management system refuses to remove it, try to stick with the installed version, perhaps TORCS will compile with it (but you will expirience some sound glitches with that).

1.6.2 Download and Unpack OpenAL

Download OpenAL from here. TORCS has been tested against this version, for other versions we guarantee for nothing! Stay as root and do

```
# cd /usr/src/torcs
# tar xfvj /path_to_downloaded_files/torcs-1.2.4-openal.tar.bz2
# cd openal/linux
```

1.6.3 Compiling and Installing OpenAL

You are still root and in /usr/src/torcs/openal/linux:

```
# ./configure
# make
# make install
```

If something fails you need to resolve it. In case the configure script complained about something you can find some additional information in the file config.log. The cause for problems are usually missing header files or libraries, wrong versions or multiple versions installed.

1.7 TORCS Installation

1.7.1 Unpacking Source Packages

If you didn't download a certain optional package, simply skip the command. Still as root do:

```
# cd /usr/src/torcs
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-src.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-src-robots-base.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-src-robots-berniw.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-src-robots-bt.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-src-robots-olethros.tgz
```

1.7.2 Building TORCS

So, fasten your seatbelts... we will run the configure script:

```
# cd /usr/src/torcs/torcs -1.2.4
# ./configure
```

Optionally you can pass –enable-debug (to build a debug version) and/or –without-xrandr (does not use xrandr extension) as argument to configure. If there are any problems reported, you have to resolve them now. Remember the config.log file which contains detailed information. If you get stuck, don't hesitate and mail to the torcs-users mailing list. Now start building:

```
# make >& error.log
```

When compiling is done, we check the error.log for... errors, yep. TORCS doesn't stop on compile errors, so you have to be careful. You can search the file for the string "error", if there is no hit, everything is fine. Because configure checkes very well there will be most often no errors, which is good, or quite bizare errors, which is not so good. In case you have SuSE 7.3 and upgraded to XFree86 4.2.0 from SuSE, you have a library in two versions, which will cause a linking error against GLU. Simply delete the file /usr/X11R6/lib/libGLU.a (the right version is in /usr/lib). This is rather strange, because it's a static library.

1.7.3 Installing TORCS

If everything worked fine till now, you can relax, the hard part is over. Run the following commands to install TORCS and unpack the data.

```
# make install
# cd /usr/local/share/games/torcs
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-tracks-road.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-tracks-dirt.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-tracks-oval.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-cars-extra.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-cars-nascar.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-cars-Patwo-Design.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-cars-kcendra-gt.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-cars-kcendra-roadsters.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-cars-kcendra-sport.tgz
# tar xfvz /path_to_downloaded_files/TORCS-1.2.4-data-cars-VM.tgz
```

1.8 TORCS Setup

1.8.1 Permissions

Because we want later compile and install your driver, we have to change some permissions. We assume that the username is uname and it belongs to the group ugroup. Simply replace uname and ugroup with your real values. Do (still as root)

```
# cd /usr/src/
# chown -R uname:ugroup torcs
# cd /usr/local/share/games
# chown -R uname:ugroup torcs
# cd /usr/local/lib
# chown -R uname:ugroup torcs
```

If the sound doesn't work, you have perhaps to put your user in a group like "audio". Look up your distribution manual for more information about sound. You're finished with administrator stuff, so you can go back to your normal user account.

1.8.2 Environment

Now we have to set up some environment variables. That you don't need to type this all the time, you put it best in your shell's rc file. For bash on SuSE that is .bashrc in your home directory. Copy the following on the end of the file. Don't forget to put a newline after the last line in the file:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
export TORCS_BASE=/usr/src/torcs/torcs-1.2.4
export MAKE_DEFAULT=$TORCS_BASE/Make-default.mk
```

Open a new terminal with a shell to check if the .bashrc did its job. Check if the variables have the right values. We will do now a final check if everything is set up correct by recompiling a single driver (remember, you are working now as normal user in a shell where the above variables are set):

```
$ cd $TORCS_BASE
$ cd src/drivers/tanhoj
$ make clean
$ make
$ make install
```

If everything worked without errors, you are ready now! Execute

```
$ torcs
```

If TORCS hangs or something went wrong, double check the stuff and look up the torcs mailing list archives. And you know, if you get stuck... the force may be with you.

On the official TORCS site is a nice guide how to set up your joystick with TORCS. Look at "Sections", "How-To Drive".

1.8.3 Command Line Options

- -s: disable multitexturing (for users of older graphics boards, e. g. i810, ATI 3D Rage LT Pro or Matrox G400/G450/G550).
- -m: use the system mouse pointer and do not hide it.
- -l: list the dynamically linked libraries.
- -d: run under gdb and print stack trace on exit, makes most sense when compiled with –enable-debug.
- -e: display the commands to issue when you want to run under gdb.

1.9 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

TORCS Robot Tutorial

The goal of this tutorial is to motivate you to write your own TORCS robot and to guide you through the first steps. So what does the term ”robot” here mean exactly? A robot is a program that drives a car (technically it's a function encapsulated into a shared object in Linux, into a DLL in Windows). It is executed from TORCS and gets as input information about the current status of its car and the situation on the track. Based on this it can compute how much it wants to steer, to brake or accelerate, which gear it needs and if it wants to pit. Your robot returns the data to TORCS and the next simulation step will be performed.

For me it's great fun to develop my robot, because I can implement my ideas and check it in the simulation. I can just say, it didn't really often happen what I expected, because it is a more complex and interesting environment as it seems on the first glimpse. You don't have just to fight with the control of your car, you have also to avoid collisions, to overtake and to resolve other situations.

So, if you like to watch races on TV here is something more exciting for you, you can let your driver enter the race and participate yourself. Take your chance and enter the race!

Requirements

You need to have TORCS installed exactly according to the installation[1] section (if you have not read the installation instructions, then do it before you start, even if TORCS runs perfectly). It helps if you already know a programming language and a bit about data structures like linked lists.

[1]See installation manual

You have to write your robot in C or C++, but that's easy if you already know another programming language, because you just need very easy stuff like basic data types, structures, pointers, arrays and functions. So if you are not familiar with C or C++, simply read further and look up a C or C++ tutorial on demand. You can find such tutorials on the internet, pick one that fits your needs.

Resources

There is a reference manual and much more on the official TORCS site. Let us know about your robot and make it available, so that the world can enjoy your work. If you want discuss about TORCS, mail to the torcs-users mailing list.

Robot Tutorial Download

If you prefer to read the tutorial offline, *it's this*.

Chapter 2

Build a Very Simple Robot

2.1 Robot Skeleton

2.1.1 Generate the Robot Skeleton

Before you start make sure that you have installed TORCS according to the installation[1] section, because that is where it is explained how to set up some required environment variables and permissions. If you do not have a proper setup you will not be able to compile or install your robot. To generate the initial set of files for your robot you have to download the robotgen script (you need exactly this version to follow the tutorial). Put it in your TORCS source directory and make it executable.

```
$ cd $TORCS_BASE
$ cp /path_to_downloaded_files/robotgen.gz .
$ gzip -d robotgen.gz
$ chmod 755 robotgen
```

You have to tell the script the robots name, your name, the car you choose, and optional a description and if you want GPL headers. You can look up the available cars with

```
$ ls /usr/local/share/games/torcs/cars

155-DTM      CORW61     acura-nsx-sz   cg-nascar-rwd  lotus-gt1     porsche-gt3rs
206W10       EVOWRC61   baja-bug       clkdtm         mclaren-f1    torcs
306W61       FOCW61     buggy          corvette       p406          viper-gts-r
360-modena   SWRC62     cg-nascar-fwd  gt40           porsche-gt1   xj-220
```

[1]See installation manual

The result depends on what TORCS car packages you have installed. To follow the tutorial you should choose cg-nascar-rwd for the car. It is very easy to change that later to your favourite car, so don't worry. I would run now the robotgen script with

```
$ cd $TORCS_BASE
$ ./robotgen -n "bt" -a "Bernhard Wymann" -c "cg-nascar-rwd" --gpl

Generation of robot bt author Bernhard Wymann
Generating src/drivers/bt/Makefile ... done
Generating src/drivers/bt/bt.xml ... done
Generating src/drivers/bt/bt.cpp ... done
Generating src/drivers/bt/bt.def ... done
Generating src/drivers/bt/bt.dsp ... done
```

You should run the script with your name and a fancy name for your robot. Keep in mind on further instructions to replace bt by your robots name. To check if everything worked we compile and install now your driver. Don't worry about the files, I'll explain them on demand.

2.1.2 Summary

- You have generated the files.
- You are aware to replace bt with your robots name.

2.2 Build the Robot

So, you want to build your robot? Ok, here is some basic information about that. As you know from the installation section, everything is already set up fine (remember the test compilation of tanhoj). The directory where your robots sources are is $TORCS_BASE/src/drivers/bt. To build your robot you will use "make". Make is a powerful tool which will track dependencies for you and much more. But don't worry, everything is already set up.

You will recognise that we have to do stuff in more than one directory, so it makes sense to use more than one shell. I use the Konsole from KDE and have one shell for every directory.

```
/usr/local/share/games/torcs to run TORCS and look up if the installation worked.
$TORCS_BASE/src/drivers/bt to work on the robot.
$TORCS_BASE/export/include/ to search in the header files.
```

I would suggest using an editor which supports at least syntax highlighting, easy to use are nedit and kate (choose C++ mode). I prefer kate.

2.2.1 Compile and Install Your Robot

```
$ cd $TORCS_BASE/src/drivers/bt
$ make
```

It should now be compiled. If you get errors, check the installation and the environment variables. If everything worked fine install your robot with

```
$ make install
```

In case you get errors here check if the compilation worked and if the permissions are correct in /usr/local/share/games/torcs. If everything worked fine, start TORCS and choose practice, choose your robot and start a practice session. Here it is and stays and stays and... Cool!

2.2.2 Summary

- You know where your robots sources are located.
- You know how to compile and install your robot.
- You know how to start TORCS and run a practice session.
- The build environment is set up properly.
- You decided which editor you want to use.

2.3 Make it Drive

2.3.1 The Track

Before we can start implementing simple steering we need to discuss how the track looks for the robot. The track is partitioned into segments of the following types: left turns, right turns and straight segments. The segments are usually short, so a turn that looks like a big turn or a long straight is most often split into much smaller segments. The segments are organized as linked list in the memory. A straight segment has a width and a length, a turn has a width, a length and a radius, everyting is measured in the middle of the track. All segments connect tangentially to the next and previous segments, so the middle line is smooth. There is much more data available, the structure tTrack is defined in $TORCS_BASE/export/include/track.h.

2.3.2 The Car

You can obtain car data by the tCarElt structure, which contains all you need. It is defined in $TORCS_BASE/export/include/car.h. But now lets implement something.

2.3.3 Implementing Simple Steering

As first challenge we will implement a very simple steering function, that simply tries to follow the middle line on the track. On a high level it could look like that:

- First steer the front wheels parallel to the track.
- If we are not in the middle of the track add a correction to the steer value.

You can get the tangent angle of the track with the following function call (look up $TORCS_BASE/export/include/robottools.h):

```
RtTrackSideTgAngleL(&(car->_trkPos))
```

The angle of the car is:

```
car->_yaw
```

The initial steering angle is then the difference:

```
angle = RtTrackSideTgAngleL(&(car->_trkPos)) - car->_yaw;
NORM_PI_PI(angle); // put the angle back in the range from -PI to PI
```

The distance to the middle line is:

```
car->_trkPos.toMiddle (+ to left, - to right)
```

The distance to the middle is measured in meters, so the values are too big to add it directly to the steering angle. So we divide it first through the width of the track and multiply it with a "tuning" constant. Now we put all the stuff together:

```
float angle;
const float SC = 1.0;
angle = RtTrackSideTgAngleL(&(car->_trkPos)) - car->_yaw;
NORM_PI_PI(angle); // put the angle back in the range from -PI to PI
angle -= SC*car->_trkPos.toMiddle/car->_trkPos.seg->width;
```

We need also to change to the first gear and apply a bit of accelerator pedal. The accelerator and brakes ranges from [0..1], the steer value from [-1.0..1.0]. Now this should raise a question. In the upper code snippet I computed a steering angle, so how do we convert this to [-1.0..1.0]? For that there is a constant in the tCarElt struct:

```
car->_steerLock
```

This value defines the angle of 100 % steering (1.0), so we need to divide our steering angle by car− >_steerLock.

```
car->ctrl.steer = angle / car->_steerLock;
car->ctrl.gear = 1; // first gear
car->ctrl.accelCmd = 0.3; // 30% accelerator pedal
car->ctrl.brakeCmd = 0.0; // no brakes
```

You can see that we return all our values in the car− >ctrl structure to TORCS. Insert this code in the bt.cpp file in your robots source directory into the drive function. It should look like this after you changed it:

```
/* Drive during race. */
static void
drive(int index, tCarElt* car, tSituation *s)
{
    memset(&car->ctrl, 0, sizeof(tCarCtrl));

    float angle;
    const float SC = 1.0;

    angle = RtTrackSideTgAngleL(&(car->_trkPos)) - car->_yaw;
    NORM_PI_PI(angle); // put the angle back in the range from -PI to PI
    angle -= SC*car->_trkPos.toMiddle/car->_trkPos.seg->width;

    // set up the values to return
    car->ctrl.steer = angle / car->_steerLock;
    car->ctrl.gear = 1; // first gear
    car->ctrl.accelCmd = 0.3; // 30% accelerator pedal
    car->ctrl.brakeCmd = 0.0; // no brakes
}
```

Compile and install your robot, and play a bit with it. In case you don't understand the algorithm, take a sheet of paper and a pencil and draw the situation with the track and the car, this will help. You will discover some problems which you have to solve later. Change some of the values, run it on different tracks, add better steering correction, whatever you like. You don't need to restart TORCS every time you did "make install", because if you start a new practice session, your robot module is reloaded. If you practice very often you should have an xosview or something similar running, because TORCS has memory leaks, so you have to restart it from time to time...

2.3.4 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

2.3.5 Summary

- You know the drive function and the file bt.cpp.
- You know how to access car data.
- You know where to look up the structure of tCarElt and tTrack.
- You know about the robottools.
- You know how to return your data to TORCS.
- You know about about dynamic loading of your robot.
- You know about the memory leaks.

2.4 What's next?

So, you had your first taste of robot development, and I hope you are keen to improve it. There are also some features of TORCS, which I would like to introduce now. As you will see in the next chapter, your robot module is able to handle up to 10 robots. That's like e. g. the "Damned" robots are implemented.

I already mentioned that your robot is able to pit. You can then tell the simulation how much fuel you need and how much damage you want to repair. We will deal with pit stops in a later chapter, because it's not that easy to stop in the pit.

Keep in mind, this is just a tutorial, so the solutions presented here are not perfect. But it should be a good starting point for you. The coming chapters deal with the following topics:

- Chapter 2 will discuss how to get unstuck and how to deal with more than one car.
- In chapter 3 we will start braking and gear changing.
- In chapter 4 we try to steer better.
- We also need a way to setup the cars, e. g. spoilers, dampers and more. That's chapter 5.
- Paint your car and a pit logo in chapter 6.
- We overtake and try to avoid collisions in chapter 7.
- And finally in chapter 8 we will do pit stops.

2.4.1 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

Chapter 3

Getting Unstuck and Handle Multiple Cars

3.1 Basic Getting Unstuck

So far our robot is just able to drive forward. That means if we hit the wall with the front of the car, we still try to drive forward and stuck in the wall. The only way to resolve the situation is to recognize it, and then to back up. Such a recovery procedure is essential even if we drive perfect, because we can be hit by opponents and spin of the track into the wall. It's also important that it works always and fast, because every second counts in the race.

3.1.1 Basic Algorithm

Here is the draft for the algorithm to detect if we are stuck:

1. Init a counter to 0.
2. If the absolute value of the angle between the track and the car is smaller than a certain value, reset counter to 0 and return ”not stuck”.
3. If the counter is smaller than a certain limit, increase the counter and return ”not stuck”, else return ”stuck”.

The algorithm to drive is then:

- If we are stuck get unstuck else drive normal.

```
// counter
static int stuck = 0;

/* check if the car is stuck */
bool isStuck(tCarElt* car)
{
    float angle = RtTrackSideTgAngleL(&(car->_trkPos)) - car->_yaw;
    NORM_PI_PI(angle);
    // angle smaller than 30 degrees?
    if (fabs(angle) < 30.0/180.0*PI) {
        stuck = 0;
        return false;
    }
    if (stuck < 100) {
        stuck++;
        return false;
    } else {
        return true;
    }
}


/* Drive during race. */
static void
drive(int index, tCarElt* car, tSituation *s)
{
    float angle;
    const float SC = 1.0;

    memset(&car->ctrl, 0, sizeof(tCarCtrl));

    if (isStuck(car)) {
        angle = -RtTrackSideTgAngleL(&(car->_trkPos)) + car->_yaw;
        NORM_PI_PI(angle); // put the angle back in the range from -PI to PI
        car->ctrl.steer = angle / car->_steerLock;
        car->ctrl.gear = -1; // reverse gear
        car->ctrl.accelCmd = 0.3; // 30% accelerator pedal
        car->ctrl.brakeCmd = 0.0; // no brakes
    } else {
        angle = RtTrackSideTgAngleL(&(car->_trkPos)) - car->_yaw;
        NORM_PI_PI(angle); // put the angle back in the range from -PI to PI
        angle -= SC*car->_trkPos.toMiddle/car->_trkPos.seg->width;
        car->ctrl.steer = angle / car->_steerLock;
        car->ctrl.gear = 1; // first gear
        car->ctrl.accelCmd = 0.3; // 30% accelerator pedal
        car->ctrl.brakeCmd = 0.0; // no brakes
    }
}
```

Try to understand why this algorithm works sometimes and implement it in bt.cpp. There is also a case where the algorithm fails, we will fix that in chapter 2.6, but try to find out what's wrong with it. The simulation timestep is 0.02 seconds, so the angle needs to be for 2 (0.02 times 100) seconds greater than 30 degrees, till we start getting unstuck. We will stop getting unstuck, when the angle becomes smaller than 30 degrees.

3.1.2 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later. Summary

- You have implemented the above stuff and played with it.
- You have found the bug in isStuck(tCarElt * car), eventually.

3.2 Comments

This section covers some selected topics related to coding for TORCS.

3.2.1 Performance

Because there are several robots in the simulation, we have to take care that our robot doesn't eat up too much performance. Look at the previous listing, I call there 3 times RtTrackSideTgAngleL. That's very bad, because there are very expensive utility functions, so you should keep the number of calls minimal. So whereever you can recycle an expensive computation, do it (that's a bit oversimplified, I know).

There are perhaps also some pieces in the code which doesn't need to be evaluated on every simulation step, so do it just every 10th or whatever is appropriate.

3.2.2 Boolean Expressions

If you need complicated boolean expressions, encapsulate them into a function or a method. This will make the code easier to understand, because the name states what we want to ask (remember isStuck()).

3.2.3 Portability

You have seen that TORCS runs on Windows, Linux/x86 and Linux/ppc. So you have to take care for portability. There are also some issues with Visual C++ 6.0.

3.2.4 For Loops

You can write (according to the standard) this for loops:

```
for (int i=0; i < 100; i++) { ... }
for (int i=7; i > 0; i--) { ... }
```

In fact the variable i should disappear after the for loop, but it doesn't in VC++. So to make VC++ happy, write

```
int i;
for (i=0; i < 100; i++) { ... }
for (i=7; i > 0; i--) { ... }
```

3.2.5 Constants in Classes

Implement constants like I do in the next section, or use #define. If you need integer constants you can also abuse enumerations for them. The following example won't work with VC++:

```
class bla {
static const int someconst = 7;
static const float someotherconst = 6.5;
};
```

3.2.6 Magic Numbers

What's also very bad on my bot till now are the numbers, which are spread all over the place. For now we should define them on a central location as constants. Constants are superior over numbers, because they have meaningful names and you need to change them just once. In a later chapter we will read some of them from an XML file.

3.2.7 Numerical Problems

To implement your robot you will certainly need floating point numbers. Keep in mind that these are not the same like the Real numbers from mathematics, they are discrete and there are gaps between any two numbers. If you are interested in the details search the web for ”IEEE 754 floating point numbers”.

3.2.8 Adding up Numbers

If you add up values of very different magnitude, e. g. the float 20.000001 and 0.000001 the sum will be still 20.000001 (and not 20.000002). The cause is the internal representation of floating point numbers, be aware of such problems. As example you can produce this way an endless while loop, when the guard waits for the sum to grow to a certain limit.

3.2.9 Special Numbers

The IEEE 754 floating point numbers have some special numbers to signal a special state or problems. The most interesting is the NaN (not a number) which results e. g. in the case when you divide 0.0 with 0.0. Once you have a NaN stored in a variable all computed results based on it are NaN as well. Sometimes you can avoid the danger with manipulating the expression.

3.2.10 Summary

- You keep an eye on performance.
- You try to write code which is easy to understand.
- You try to write portable code.
- Avoid meaningless numbers, use constants instead.
- Be aware of floating point numbers.

3.3 Define Multiple Robots

First ask yourself the question, why we want more than one driver in one module? It's because we can share the program code and data between the drivers of our module. If we write our driver general enough, we can load parameters for the different tracks and cars, so no code difference is necessary.

3.3.1 Car Definition

In chapter 1.1 we had chosen a cg-nascar-fwd as our car. When you edited bt.cpp, you might have recognized that in this file is no clue which car we want to use. The information about the cars is stored in the bt.xml file. So let's have a look at it.

The following three lines just define the name and the structure. You'll not need to change them.

```
<params name="bt" type="robotdef">
  <section name="Robots">
    <section name="index">
```

The next line

```
      <section name="1">
```

defines the index of the driver. Because we can have more than one driver per module, each one is uniquely identified by its index. You will receive it e. g. in your

```
static void drive(int index, tCarElt* car, tSituation *s)
```

function from TORCS, so you know for which car you need to drive. It has to match with the index you assign to modInfo[i].index in bt.cpp, but you will see that later. The next two lines contain the name and description of you driver, they have to match with the modInfo[i] entries (don't ask me why it has to be defined here and in the bt.cpp).

```
        <attstr name="name" val="bt 1"></attstr>
        <attstr name="desc" val=""></attstr>
```

The next three entries are your teams name, your name and the car to use.

```
        <attstr name="team" val="berniw"></attstr>
        <attstr name="author" val="Bernhard Wymann"></attstr>
        <attstr name="car name" val="cg-nascar-rwd"></attstr>
```

Now you can choose a racing number and define the color of the drivers entry in the leaders board (if you don't know how to enable the leaders board during the race, hit F1 for help).

```
        <attnum name="race number" val="61"></attnum>
        <attnum name="red" val="1.0"></attnum>
        <attnum name="green" val="0.0"></attnum>
        <attnum name="blue" val="1.0"></attnum>
      </section>
    </section>
  </section>
</params>
```

3.3.2 Adding Cars

We will now add two car entries to the file bt.xml (one module is allowed to handle up to ten robots). This will not yet work in the simulation, because we need to change also bt.cpp. You can simply copy the part between ¡section name="1"¿ and ¡/section¿ two times and change the race numbers, the leaders board color if you like, the car you choose and of course the index. To follow the tutorial choose indices 0, 1 and 2 and the names "bt 1", "bt 2" and "bt 3". If you want other cars do

```
$ ls /usr/local/games/torcs/cars
```

to get a list of the installed cars. The file bt.xml should finally look similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    file                 : bt.xml
    created              : Thu Dec 12 02:34:31 CET 2002
    copyright            : (C) 2002 Bernhard Wymann
-->

<!--    This program is free software; you can redistribute it and/or modify  -->
<!--    it under the terms of the GNU General Public License as published by  -->
<!--    the Free Software Foundation; either version 2 of the License, or     -->
<!--    (at your option) any later version.                                   -->

<!DOCTYPE params SYSTEM "../../libs/tgf/params.dtd">
```

```
<params name="bt" type="robotdef">
  <section name="Robots">
    <section name="index">

      <section name="0">
        <attstr name="name" val="bt 1"></attstr>
        <attstr name="desc" val=""></attstr>
        <attstr name="team" val="berniw"></attstr>
        <attstr name="author" val="Bernhard Wymann"></attstr>
        <attstr name="car name" val="cg-nascar-rwd"></attstr>
        <attnum name="race number" val="61"></attnum>
        <attnum name="red" val="1.0"></attnum>
        <attnum name="green" val="0.0"></attnum>
        <attnum name="blue" val="1.0"></attnum>
      </section>

      <section name="1">
        <attstr name="name" val="bt 2"></attstr>
        <attstr name="desc" val=""></attstr>
        <attstr name="team" val="berniw"></attstr>
        <attstr name="author" val="Bernhard Wymann"></attstr>
        <attstr name="car name" val="xj-220"></attstr>
        <attnum name="race number" val="62"></attnum>
        <attnum name="red" val="1.0"></attnum>
        <attnum name="green" val="0.0"></attnum>
        <attnum name="blue" val="1.0"></attnum>
      </section>

      <section name="2">
        <attstr name="name" val="bt 3"></attstr>
        <attstr name="desc" val=""></attstr>
        <attstr name="team" val="berniw"></attstr>
        <attstr name="author" val="Bernhard Wymann"></attstr>
        <attstr name="car name" val="155-DTM"></attstr>
        <attnum name="race number" val="63"></attnum>
        <attnum name="red" val="1.0"></attnum>
        <attnum name="green" val="0.0"></attnum>
        <attnum name="blue" val="1.0"></attnum>
      </section>

    </section>
  </section>
</params>
```

Summary

- You know how to define cars in bt.xml.
- You know you can define up to ten cars.
- To follow the tutorial you have chosen the indices 0, 1 and 2.

3.4 Basic Implementation

This chapter will just cover the very basic initialization and shutdown of the module for multiple robots. More topics will follow in the next section. All discussed changes apply to bt.cpp.

3.4.1 Initialization

First we define how many robots we want to handle (sorry about mixing up the terms ″cars″, ″drivers″ and ″robots″) and the size of the buffer to generate the names.

```
#define BUFSIZE 20
#define NBBOTS 3
```

We also need an array of char pointers to remember the memory locations of the names to free them on shutdown.

```
static char * botname[NBBOTS];
```

We modify the module entry point:

```
/*
 * Module entry point
*/
extern "C" int
bt(tModInfo *modInfo)
{
    char buffer[BUFSIZE];
    int i;

    /* clear all structures */
    memset(modInfo, 0, 10*sizeof(tModInfo));

    for (i = 0; i < NBBOTS; i++) {
        sprintf(buffer, "bt %d", i+1);
        botname[i] = strdup(buffer);      /* store pointer */
        modInfo[i].name    = botname[i];  /* name of the module (short) */
        modInfo[i].desc    = "";          /* description of the module */
        modInfo[i].fctInit = InitFuncPt;  /* init function */
        modInfo[i].gfId    = ROB_IDENT;   /* supported framework version */
        modInfo[i].index   = i;           /* indices from 0 to 9 */
    }
    return 0;
}
```

Now we loop NBBOTS times to initialize all robots. You can see how the name is assembled in the buffer. We have to store the pointers to the names because strdup(buffer) will allocate memory on the heap, which we have to release on shutdown (there is no garbage collector in C or C++).

In case you don't want such systematic names, like "bt 1, bt 2, bt 3", you can also implement them as static character arrays:

```
static char * botname[NBBOTS] = { "tiger", "penguin", "rat" };
```

The loop would look like that then:

```
...
for (i = 0; i < NBBOTS; i++) {
    modInfo[i].name = botname[i];  /* name of the module (short) */
...
```

This has also the advantage that you don't need to free memory on shutdown. The advantage of the first method is, if you want to add a driver, you just need to increment NBBOTS. Remember that the names should match with the names defined in the bt.xml file.

3.4.2 Shutdown

We have also to change the shutdown of the module to free the memory on the heap. Because shutdown is called for every robot, we need just to release the memory of botname with index i.

```
/* Called before the module is unloaded */
static void
shutdown(int index)
{
    free(botname[index]);
}
```

You should now implement the above stuff. In case you try it out, keep in mind, that if you run more than one robot of this module, they will share the stuck variable. So there could be weird effects. Anyway, try a quickrace with your three robots to check if everything is ok so far.

3.4.3 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

3.4.4 Summary

- You understand how initialization and shutdown works.
- You know how to add more cars.
- You have implemented it.

3.5 Finishing Implementation

3.5.1 Overview

In this chapter we will change the structure of the robot. Of course there are other possibilities to structure the robots than the one presented here. We will also clean the code up according to the comments made in chapter 2.2. Don't be afraid about the length of this page, most stuff is familiar for you and is just moved into another file.

3.5.2 The Driver Class Definition driver.h

Now I will show you the basic class definition for our driver. Implement it in a new file called driver.h. But why should we take a class? Perhaps you will implement later a very sophisticated robot, which will use a lot of memory. Assume you have 10 cars available in your module. If you declare it static you allocate always the full memory for 10 drivers, even if you just select one out of 10. Instead of putting huge static arrays in bt.cpp, you can get the memory in the constructor of your driver. This way you just allocate the memory you really need. There are also other advantages like you don't mind anymore of the index.

```
/***************************************************************************

    file                 : driver.h
    created              : Thu Dec 20 01:20:19 CET 2002
    copyright            : (C) 2002 Bernhard Wymann

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify *
 *   it under the terms of the GNU General Public License as published by *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef _DRIVER_H_
#define _DRIVER_H_
```

With this define we make sure that the file content is just included once. There is also a matching #endif on the end of the file. If you want to learn more about such # stuff, search for information about your C preprocessor.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <tgf.h>
#include <track.h>
#include <car.h>
#include <raceman.h>
#include <robottools.h>
#include <robot.h>

class Driver {
    public:
        Driver(int index);

        /* callback functions called from TORCS */
        void initTrack(tTrack* t, void *carHandle,
                       void **carParmHandle, tSituation *s);
        void newRace(tCarElt* car, tSituation *s);
        void drive(tCarElt* car, tSituation *s);
        int pitCommand(tCarElt* car, tSituation *s);
        void endRace(tCarElt *car, tSituation *s);
```

First we include all the stuff we need. Then we define the constructor, which has the robot index as argument. It will store this index, so the following functions initTrack, newRace, drive and pitCommand don't need it anymore. These are public, because they are called then from bt.cpp and should finally do the work. Shutdown is not implemented here, because we need to call the destructor on shutdown to free the memory. In case you allocate memory with new, don't forget to add a destructor with code to release it.

```
    private:
        /* utility functions */
        bool isStuck(tCarElt* car);
        void update(tCarElt* car, tSituation *s);

        /* per robot global data */
        int stuck;
        float trackangle;
        float angle;

        /* data that should stay constant after first initialization */
        int MAX_UNSTUCK_COUNT;
        int INDEX;

        /* class constants */
        static const float MAX_UNSTUCK_ANGLE;
        static const float UNSTUCK_TIME_LIMIT;

        /* track variables */
        tTrack* track;
};
```

You know already isStuck, we move it also into our class. Update is supposed to prepare data we need in more than one place in our robot like trackangle and angle. You can also see the nice names for the constants here. MAX_UNSTUCK_ANGLE tells you more than 30.0, doesn't it?

```
#endif // _DRIVER_H_
```

This closes the leading #ifndef and first implementation of driver.h. The Driver Class Implementation driver.cpp

Now we will put concrete implementations of the above defined stuff in driver.cpp.

```
/***************************************************************************

    file                 : driver.cpp
    created              : Thu Dec 20 01:21:49 CET 2002
    copyright            : (C) 2002 Bernhard Wymann

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#include "driver.h"
```

This includes the previously defined file driver.h.

```
const float Driver::MAX_UNSTUCK_ANGLE = 30.0/180.0*PI;  /* [radians] */
const float Driver::UNSTUCK_TIME_LIMIT = 2.0;           /* [s] */
```

Here are the constants implemented which are already defined in driver.h. We do it this way for compatibility with VC++ 6.0. You should put at least the unit into the comment.

```
Driver::Driver(int index)
{
    INDEX = index;
}
```

In the constructor we save the index for later use. Probably we can remove the INDEX later, because we don't need it.

```
/* Called for every track change or new race. */
void Driver::initTrack(tTrack* t, void *carHandle,
                       void **carParmHandle, tSituation *s)
{
    track = t;
    *carParmHandle = NULL;
}

/* Start a new race. */
void Driver::newRace(tCarElt* car, tSituation *s)
{
    MAX_UNSTUCK_COUNT = int(UNSTUCK_TIME_LIMIT/RCM_MAX_DT_ROBOTS);
    stuck = 0;
}
```

We don't dig into initTrack yet, this will follow in the chapter about how to load custom car setups. In newRace we init our stuck counter variable with 0. Did you recognize that now every driver instance has its own stuck variable? We also replace the former 100 with MAX_UNSTUCK_COUNT, which is computed based on the desired delay in UNSTUCK_TIME_LIMIT and the simulation timestep.

```
/* Drive during race. */
void Driver::drive(tCarElt* car, tSituation *s)
{
    update(car, s);

    memset(&car->ctrl, 0, sizeof(tCarCtrl));

    if (isStuck(car)) {
        car->ctrl.steer = -angle / car->_steerLock;
        car->ctrl.gear = -1; // reverse gear
        car->ctrl.accelCmd = 0.3; // 30% accelerator pedal
        car->ctrl.brakeCmd = 0.0; // no brakes
    } else {
        float steerangle = angle - car->_trkPos.toMiddle/car->_trkPos.seg->width;

        car->ctrl.steer = steerangle / car->_steerLock;
        car->ctrl.gear = 1; // first gear
        car->ctrl.accelCmd = 0.3; // 30% accelerator pedal
        car->ctrl.brakeCmd = 0.0; // no brakes
    }
}
```

You are already familiar with drive from bt.cpp. New is the update of the data and the use of precomputed variables instead of expensive RtTrackSideTgAngleL() calls.

```
/* Set pitstop commands. */
int Driver::pitCommand(tCarElt* car, tSituation *s)
{
    return ROB_PIT_IM; /* return immediately */
}

/* End of the current race */
void Driver::endRace(tCarElt *car, tSituation *s)
{
}

/* Update my private data every timestep */
void Driver::update(tCarElt* car, tSituation *s)
{
    trackangle = RtTrackSideTgAngleL(&(car->_trkPos));
    angle = trackangle - car->_yaw;
    NORM_PI_PI(angle);
}
\begin{lstlisting}

  We will discuss pitCommand in the pit stop chapter. In endRace you can put
  code which should run after the race. Update does assign the current values
  to angle and trackangle. Put in here updates to driver variables which you
  would like to perform on every simulation timestep.

\begin{lstlisting}
/* Check if I'm stuck */
bool Driver::isStuck(tCarElt* car)
{
    if (fabs(angle) < MAX_UNSTUCK_ANGLE) {
        stuck = 0;
        return false;
    }
    if (stuck < MAX_UNSTUCK_COUNT) {
        stuck++;
        return false;
    } else {
        return true;
    }
}
```

This is also already well known. Changes are the use of constants and the precomputed angle. That finishes the driver.cpp.

3.5.3 Changing bt.cpp

Here I will present the fitting bt.cpp file. Major changes include the calls to the driver methods and the construction and destruction of the driver instances. Have a look at it.

```
/***************************************************************************

    file                 : bt.cpp
    created              : Thu Dec 12 02:34:31 CET 2002
    copyright            : (C) 2002 Bernhard Wymann

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifdef _WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <tgf.h>
#include <track.h>
#include <car.h>
#include <raceman.h>
#include <robottools.h>
#include <robot.h>

#include "driver.h"

#define BUFSIZE 20
#define NBBOTS 3

static char *botname[NBBOTS];
static Driver *driver[NBBOTS];
```

Here we allocate an array which holds pointers to our drivers.

```
static void initTrack(int index, tTrack* track,
                      void *carHandle, void **carParmHandle, tSituation *s);
static void newRace(int index, tCarElt* car, tSituation *s);
static void drive(int index, tCarElt* car, tSituation *s);
static int  pitcmd(int index, tCarElt* car, tSituation *s);
static void shutdown(int index);
static int InitFuncPt(int index, void *pt);
static void endRace(int index, tCarElt *car, tSituation *s);

/* Module entry point */
extern "C" int bt(tModInfo *modInfo)
{
    char buffer[BUFSIZE];
    int i;

    /* clear all structures */
    memset(modInfo, 0, 10*sizeof(tModInfo));
    for (i = 0; i < NBBOTS; i++) {
        sprintf(buffer, "bt %d", i+1);
        botname[i] = strdup(buffer);       /* store pointer to string */
        modInfo[i].name    = botname[i];   /* name of the module (short) */
        modInfo[i].desc    = "";           /* description of the module (can be long) */
        modInfo[i].fctInit = InitFuncPt;   /* init function */
        modInfo[i].gfId    = ROB_IDENT;    /* supported framework version */
        modInfo[i].index   = i;            /* indices from 0 to 9 */
    }
    return 0;
}

/* Module interface initialization. */
static int InitFuncPt(int index, void *pt)
{
    tRobotItf *itf = (tRobotItf *)pt;

    /* create robot instance for index */
    driver[index] = new Driver(index);
```

Create an instance for driver index.

```
    itf->rbNewTrack = initTrack; /* Give the robot the track view called */
    itf->rbNewRace  = newRace;   /* Start a new race */
    itf->rbDrive    = drive;     /* Drive during race */
    itf->rbPitCmd   = pitcmd;    /* Pit commands */
    itf->rbEndRace  = endRace;   /* End of the current race */
    itf->rbShutdown = shutdown;  /* Called before the module is unloaded */
    itf->index      = index;     /* Index used if multiple interfaces */
    return 0;
}

/* Called for every track change or new race. */
static void initTrack(int index, tTrack* track, void *carHandle,
                      void **carParmHandle, tSituation *s)
{
    driver[index]->initTrack(track, carHandle, carParmHandle, s);
}

/* Start a new race. */
static void newRace(int index, tCarElt* car, tSituation *s)
{
    driver[index]->newRace(car, s);
}


/* Drive during race. */
static void drive(int index, tCarElt* car, tSituation *s)
{
    driver[index]->drive(car, s);
}


/* Pitstop callback */
static int pitcmd(int index, tCarElt* car, tSituation *s)
{
    return driver[index]->pitCommand(car, s);
}
```

```
/* End of the current race */
static void endRace(int index, tCarElt *car, tSituation *s)
{
    driver[index]->endRace(car, s);
}


/* Called before the module is unloaded */
static void shutdown(int index)
{
    free(botname[index]);
    delete driver[index];
}
```

Like you can see most functions route now the call to the corresponding driver method. Shutdown does now also delete the instance of driver index.

3.5.4 Changing the Makefile

Because we need now to compile the additional file driver.cpp, we have to update the Makefile. Change the line

```
SOURCES     = ${ROBOT}.cpp
```

to

```
SOURCES     = ${ROBOT}.cpp driver.cpp
```

Now you should be able to build the whole bunch.

3.5.5 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

3.5.6 Summary

- You know how the robot works so far.
- You know how to add source files.
- You have implemented it.

3.6 Improving Getting Unstuck

3.6.1 Current Version

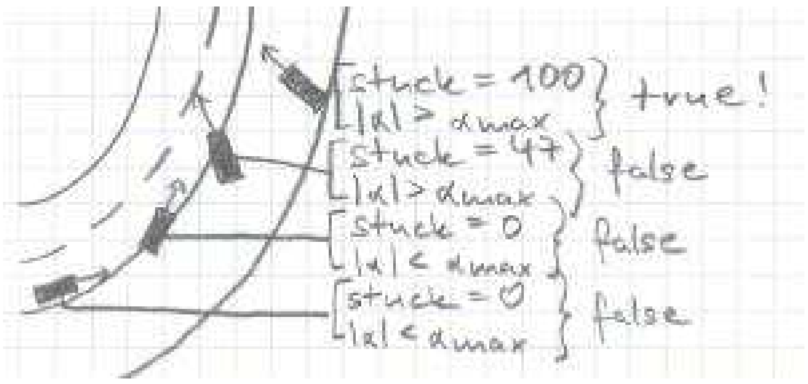


Figure 3.1: sketch of skidding

First I will show you what the problem is with the current isStuck() function. Let's have a look at the sketch on the left. Assume the car comes from bottom left and wants to drive a left turn. It is too fast and starts oversteering, so the angle between the track tangent and the car becomes bigger and bigger. At a certain point the angle is big enough to start incrementing ″stuck″. When the car hits the wall with its back ″stuck″ is 100 and we try to get unstuck. So we are with the back at the wall and try to back up. So we are stuck again... Look careful at the sketch, it shows us also the solution for the problem. The front of the car looks ″inside″ toward the track middle. But like you can see, there is never such a situation where we need to get unstuck. You simply can drive forward to come back on the track.

3.6.2 Look Inside Criteria

If you think about our current code, you can perhaps remember that we already compute the variable ″angle″, which is the track tangent angle minus the car angle. So ″angle″ is positive clockwise (I called it alpha on the sketch). You should also remember car-¿_trkPos.toMiddle, which is positive on the left trackside and negative on the right side. Now you can conclude if ″angle″ times ″tomiddle″ is greater than zero we are looking

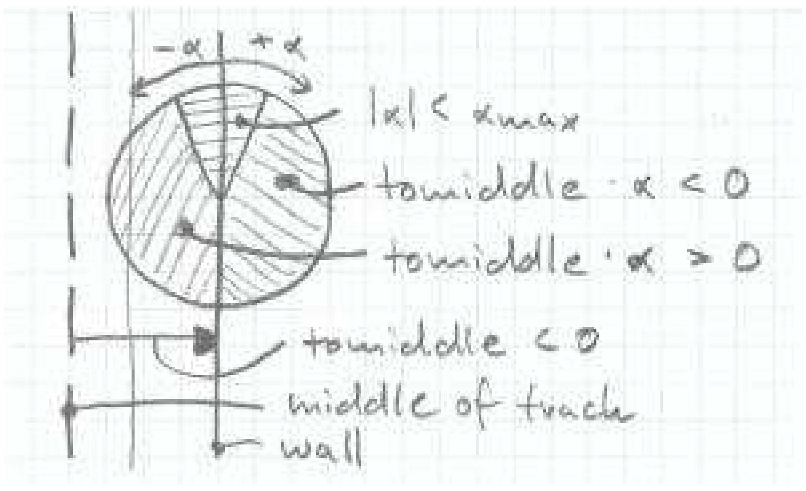


Figure 3.2: sketch of inside criteria

toward the middle. This criteria holds always. You can paint the areas for the different parts of our expressions into the circle to understand it better. So if ”fabs(angle) ¡ MAX_UNSTUCK_ANGLE” is true we try to drive forward, else we check if we look outside and the ”stuck” is already big enough. If that all is fullfilled, we return true to get unstuck. If not we just increment ”stuck”.

3.6.3 Additional Conditions

As you can see, MAX_UNSTUCK_ANGLE is 30 degrees. This is quite much, and can still cause problems if we try to drive forward along the wall. So we will reduce that angle. We will also add the condition, that the speed has to be below a certain threshold, before we try to get unstuck. An additional problem can occur if we try to get unstuck on the middle of the track. The following implementation will solve most of this problems. It will not solve the case, where the wall itself is not parallel to the track.

3.6.4 The Implementation

So finally here is the new isStuck(). Change it in driver.cpp. You can also further improve that.

```
/* Check if I'm stuck */
```

```
bool Driver::isStuck(tCarElt* car)
{
    if (fabs(angle) > MAX_UNSTUCK_ANGLE &&
        car->_speed_x < MAX_UNSTUCK_SPEED &&
        fabs(car->_trkPos.toMiddle) > MIN_UNSTUCK_DIST) {
        if (stuck > MAX_UNSTUCK_COUNT && car->_trkPos.toMiddle*angle < 0.0) {
            return true;
        } else {
            stuck++;
            return false;
        }
    } else {
        stuck = 0;
        return false;
    }
}
```

Change also in driver.cpp this part of the drive function to apply more throttle to back up.

```
if (isStuck(car)) {
    car->ctrl.steer = -angle / car->_steerLock;
    car->ctrl.gear = -1; // reverse gear
    car->ctrl.accelCmd = 0.5; // 50% accelerator pedal
    car->ctrl.brakeCmd = 0.0; // no brakes
} else {
```

Put the constants at the start of driver.cpp. Reduce also MAX_UNSTUCK_ANGLE.

```
const float Driver::MAX_UNSTUCK_SPEED = 5.0;    /* [m/s] */
const float Driver::MIN_UNSTUCK_DIST = 3.0;     /* [m] */
```

You need also to define the new constants in driver.h.

```
static const float MAX_UNSTUCK_SPEED;
static const float MIN_UNSTUCK_DIST;
```

3.6.5 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

3.6.6 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

3.6.7 Summary

- You understand how getting unstuck works.
- You know about every expression why it is necessary.
- You have implemented it.

Chapter 4

Braking and Gear Changing

4.1 Basic Utility Functions

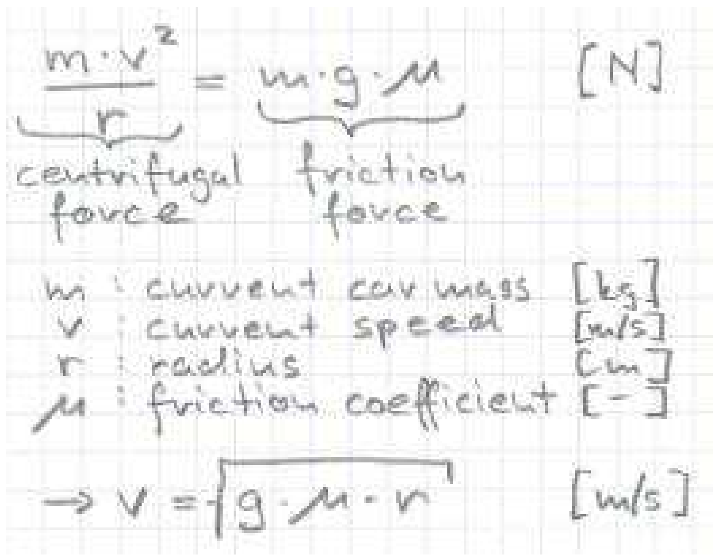
Ah, it's you. Welcome back. The boring "administrative" stuff is over, now we start to bang out the most out of our car. This chapter will introduce you in braking and gear changing. We will still drive on the middle of the track, so don't expect exciting lap times. But what braking concerns, you will implement in this chapter all "secrets" from my berniw robot. But before we start let's record some lap times of our current implementation with the cg-nascar-rwd car:

- mixed-2: 1:24:24, 13314 damage.
- e-track-2: 2:18:76, 201 damage.
- e-track-4: 4:51:75, 4 damage.

In the "Basic" sections of this chapter I will introduce you into driving without considering aerodynamics, anti wheel locking and traction control. In the ”Advanced” sections we will add this features. First we start with developing some useful utility functions.

4.1.1 About Speed Limits

$\frac{m.v^2}{2} = m.g.\mu.$



$\underbrace{\frac{m \cdot v^2}{r}}_{\text{centrifugal force}} = \underbrace{m \cdot g \cdot \mu}_{\text{friction force}}$ [N]
m : current car mass [kg]
v : current speed [m/s]
r : radius [m]
μ : friction coefficient [-]
$\rightarrow v = \sqrt{g \cdot \mu \cdot r}$ [m/s]

Figure 4.1: speed formula Of course

m: current car mass

v: current speed

r: radius

μ: friction coefficient

$\implies v = \sqrt{g.\mu.r}$

All speed junkies get now sick because of the section title. But keep in mind, if you want to race, use TORCS or go to the race track, please don't use public roads. But that's not the subject, I want to discuss the physical speed limits...

Like you know from experience, you can't pass a turn with any speed, there are limits which can be described with physical laws. Of course we need to know how fast we can pass a turn. The force that pushes the car outside the turn is called centrifugal force. That's the left part of the top equation on the sketch. To hold the car on the track, we need to compensate it. Our tires will do that job with friction on the track. The maximum possible speed in a turn results if you solve the equation. You should notice that we assume that all wheels have the same mu and the load is equally distributed over all wheels. We further simplify with the friction coefficient mu, in reality it's a complex function which depends on the relative speed vector between the surfaces, the current load, the shape of the patch, the temperature, ... This leads to the following function:

```
/* Compute the allowed speed on a segment */
float Driver::getAllowedSpeed(tTrackSeg *segment)
{
    if (segment->type == TR_STR) {
        return FLT_MAX;
    } else {
        float mu = segment->surface->kFriction;
        return sqrt(mu*G*segment->radius);
    }
}
```

The function gets a pointer to the segment in question. If it's a straight segment there is no limit, else we compute the limit according to the above formula. You can improve the function e. g. with taking into account the center of gravity of the car.

4.1.2 Distance

For braking we need also to know distances. Here is a function that computes the distance of the car to the end of the current segment (remember the track segments or look up chapter 1.3). I put this into a function, because car-¿_trkPos.toStart contains depending on the segment type the arc or the length, so we need to convert it sometimes to the length (arc times radius = length).

```
/* Compute the length to the end of the segment */
float Driver::getDistToSegEnd(tCarElt* car)
{
    if (car->_trkPos.seg->type == TR_STR) {
        return car->_trkPos.seg->length - car->_trkPos.toStart;
    } else {
        return (car->_trkPos.seg->arc - car->_trkPos.toStart)*car->_trkPos.seg->radius;
    }
}
```

4.1.3 Accelerator

We need also to know for a given speed how much accelerator pedal we have to apply. If we tell TORCS to accelerate 100% (1.0), it tries to run the engine at the maximum allowed rpm (car-¿_enginerpmRedLine). We can compute the angular speed of the wheel with the speed and the wheel radius. This has to be equal like the desired rpm divided by the gear ratio. So we finally end up with an equation for the desired rpm, the pedal value is then rpm/enginerpmRedLine. If the given speed is much higher than the current speed, we will accelerate full.

```
/* Compute fitting acceleration */
float Driver::getAccel(tCarElt* car)
{
    float allowedspeed = getAllowedSpeed(car->_trkPos.seg);
    float gr = car->_gearRatio[car->_gear + car->_gearOffset];
    float rm = car->_enginerpmRedLine;
    if (allowedspeed > car->_speed_x + FULL_ACCEL_MARGIN) {
        return 1.0;
    } else {
        return allowedspeed/car->_wheelRadius(REAR_RGT)*gr /rm;
    }
}
```

4.1.4 Finishing Implementation

We have to define the constants G and FULL_ACCEL_MARGIN in driver.cpp

```
const float Driver::G = 9.81;                    /* [m/(s*s)] */
const float Driver::FULL_ACCEL_MARGIN = 1.0;     /* [m/s] */
```

and in driver.h.

```
static const float G;
static const float FULL_ACCEL_MARGIN;
```

Put also the above method interfaces into driver.h.

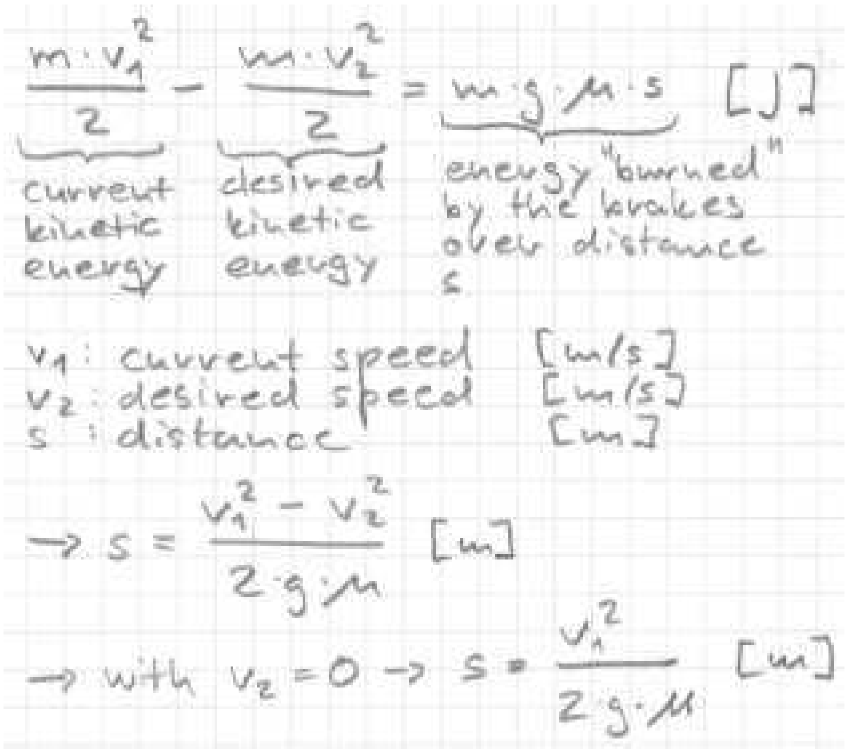
```
float getAllowedSpeed(tTrackSeg *segment);
float getAccel(tCarElt* car);
float getDistToSegEnd(tCarElt* car);
```

4.1.5 Summary

- You respect speed limits on public roads.
- You know the physics of driving through turns.
- You have implemented it.

4.2 Basic Braking

4.2.1 Braking Distance



$$\underbrace{\frac{m\cdot v_1^2}{2}}_{\text{current kinetic energy}} - \underbrace{\frac{m\cdot v_2^2}{2}}_{\text{desired kinetic energy}} = \underbrace{m\cdot g\cdot \mu\cdot s}_{\text{energy "burned" by the brakes over distance } s} \quad [J]$$

v_1	: current speed	[m/s]
v_2	: desired speed	[m/s]
s	: distance	[m]

$$\rightarrow s = \frac{v_1^2 - v_2^2}{2\cdot g\cdot \mu} \quad [m]$$

$$\rightarrow \text{with } v_2 = 0 \rightarrow s = \frac{v_1^2}{2\cdot g\cdot \mu} \quad [m]$$

Figure 4.2: brake distance formulas

$\frac{m.v_1^2}{2} - \frac{m.v_2^2}{2} = m.g.\mu.$

v_1: current speed

v_2: desired speed

s: distance

$\implies s = \frac{v_1^2 - v_2^2}{2.g.\mu}$

with $v_2 = 0$

$\implies s = \frac{v_1^2}{2.g.\mu}$

Like you have seen in the previous section, there is a physical speed limit in the turns. Imagine the following situation: we drive on a straight with speed v1, and in distance d is a turn. We can compute the possible speed (we call it v2) in the turn with the getAllowedSpeed() method, which we have developed in the previous chapter. If we make some assumptions, we can compute the required minimal braking distance s. Now if s is greater or equal than d we have to start braking (you see we brake as late as possible, and even a bit later).

But how do we compute this distance? Look a the top formula on the sketch. This time we work with energy equations. Our car has a certain amount of kinetic energy, and we want to reach a lower energy state. We have to "burn" the energy difference in our brakes. You can solve the equation for s, which is our braking distance. We assume in this equation, that mu is constant over s, what is actually wrong e. g. at mixed tracks. But don't worry, it will work quite well anyway, but here is room for improvement.

There is a special case of the formula, where v2 equals zero. We need that to compute the maximum braking distance. The idea is that in the worst case we have to stop, so checking this distance is sufficient.

4.2.2 Implementation

Here is the code that computes if we need to brake.

```
float Driver::getBrake(tCarElt* car)
{
    tTrackSeg *segptr = car->_trkPos.seg;
    float currentspeedsqr = car->_speed_x*car->_speed_x;
    float mu = segptr->surface->kFriction;
    float maxlookaheaddist = currentspeedsqr/(2.0*mu*G);
```

maxlookaheddist is the distance we have to check (formula with special case v2 = 0).

```
    float lookaheaddist = getDistToSegEnd(car);
```

lookaheaddist holds the distance we have already checked. First we check if we need to brake for a speed limit on the end of the current segment.

```
    float allowedspeed = getAllowedSpeed(segptr);
    if (allowedspeed < car->_speed_x) return 1.0;
```

Compute the allowed speed on the current segment. We check our speed, and if we are too fast we brake, else we continue with the algorithm. Here you can improve the return value, it's a bit tough to brake full (e. g. make it dependent on the speed difference).

```
    segptr = segptr->next;
    while (lookaheaddist < maxlookaheaddist) {
```

The first line moves segptr to the next segment. The guard of the loop checks if we have already checked far enough.

```
        allowedspeed = getAllowedSpeed(segptr);
        if (allowedspeed < car->_speed_x) {
```

Compute the allowed speed on the *segptr segment. If the allowed speed is smaller than the current speed, we need to investigate further.

```
            float allowedspeedsqr = allowedspeed*allowedspeed;
            float brakedist = (currentspeedsqr - allowedspeedsqr) / (2.0*mu*G);
```

Here we compute the braking distance according to the formula above.

```
            if (brakedist > lookaheaddist) {
```

Here the magic check is done. If the required distance to brake is greater than the current distance we need to brake. This works because the simulation timestep is small, so we fail the point in the worst case with 2.0 meters. So to fix that you can add always 2 meters to the brakedist, or better a speed dependent value.

```
                return 1.0;
            }
        }
        lookaheaddist += segptr->length;
        segptr = segptr->next;
    }
    return 0.0;
}
```

The remaining code is straightforward. If we decided to brake we return 1. If we loop further we update the lookaheaddist and switch to the next segment. A comment to the return value: 1 means apply full brakes, so we have later to adjust the pressure in the brake system, that the wheels don't lock up immediately or we have enough pressure at all. You will see this on the xj-220, the wheels will smoke very nice... A hint: it should also be possible to compute the right pressure and do an automatic set up of that value.

Now we need also to change the drive method to call getAccel and getBrake.

```
        car->ctrl.gear = 4;
        car->ctrl.brakeCmd = getBrake(car);
        if (car->ctrl.brakeCmd == 0.0) {
            car->ctrl.accelCmd = getAccel(car);
        } else {
            car->ctrl.accelCmd = 0.0;
        }
```

We have to switch into a higher gear to check the stuff. We just accelerate if we don't brake. And finally you need to update driver.h with the interface of getBrake.

```
        float getBrake(tCarElt* car);
```

4.2.3 Testdrive

Do some test runs with the different cars. Enable the debug vector (press "g") to watch the return values of the robot. With cg-nascar-rwd you should recognize that the braking is very "flashy" on high speeds, try to explain why. The lap times are now:

- mixed-2: 1:36:66, 4 damage.
- e-track-2: 1:53:85, 0 damage.
- e-track-4: 2:19:08, 4 damage.

4.2.4 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

4.2.5 Summary

- You have understood the physics of braking.
- You have implemented it.

Chapter 5

Steering and Trajectory

5.1 The Vector Class

In this chapter we will first develop a class for 2D-vectors. It will make our life easier when we try to compute the steer angle or distances and angles to other cars in the later chapters. After that we improve the steering of the car and reduce the lap times. We will do it this time with a more heuristic approach, because the other methods are not that easy to implement. At the end of this chapter we will discuss some other methods for finding a path around the track.

5.1.1 Implementation

I will simply show you the vector class ″v2d″ here. Look up the implementation for details. We implement operators for vector addition, subtraction, negation, multiplication with a scalar and the dot product. Further we want also be able to normalize the vector, to rotate it around a center and to get it's length. Put the following code in a new file called linalg.h.

```
/***************************************************************************

    file                 : linalg.h
    created              : Wed Feb 18 01:20:19 CET 2003
    copyright            : (C) 2003 Bernhard Wymann

 ***************************************************************************/
```

```
/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef _LINALG_H_
#define _LINALG_H_

class v2d {
    public:
        /* constructors */
        v2d() {}
        v2d(const v2d &src) { this->x = src.x; this->y = src.y; }
        v2d(float x, float y) { this->x = x; this->y = y; }

        /* operators */
        v2d& operator=(const v2d &src);          /* assignment */
        v2d operator+(const v2d &src) const;     /* addition */
        v2d operator-(void) const;               /* negation */
        v2d operator-(const v2d &src) const;     /* subtraction */
        v2d operator*(const float s) const;      /* multiply with scalar */
        float operator*(const v2d &src) const;   /* dot product */
        friend v2d operator*(const float s, const v2d & src);

        /* methods */
        float len(void) const;
        void normalize(void);
        float dist(const v2d &p) const;
        float cosalpha(const v2d &p2, const v2d &center) const;
        v2d rotate(const v2d &c, float arc) const;

        /* data */
        float x;
        float y;
};

/* assignment */
inline v2d& v2d::operator=(const v2d &src)
{
    x = src.x; y = src.y; return *this;
}

/* add *this + src (vector addition) */
inline v2d v2d::operator+(const v2d &src) const
{
    return v2d(x + src.x, y + src.y);
}

/* negation of *this */
inline v2d v2d::operator-(void) const
{
    return v2d(-x, -y);
}

/* compute *this - src (vector subtraction) */
inline v2d v2d::operator-(const v2d &src) const
{
    return v2d(x - src.x, y - src.y);
}

/* scalar product */
inline float v2d::operator*(const v2d &src) const
{
    return src.x*x + src.y*y;
}

/* multiply vector with scalar (v2d*float) */
inline v2d v2d::operator*(const float s) const
{
    return v2d(s*x, s*y);
}

/* multiply scalar with vector (float*v2d) */
inline v2d operator*(const float s, const v2d & src)
{
    return v2d(s*src.x, s*src.y);
}

/* compute cosine of the angle between vectors *this-c and p2-c */
inline float v2d::cosalpha(const v2d &p2, const v2d &c) const
{
    v2d l1 = *this-c;
    v2d l2 = p2 - c;
    return (l1*l2)/(l1.len()*l2.len());
}

/* rotate vector arc radians around center c */
inline v2d v2d::rotate(const v2d &c, float arc) const
{
    v2d d = *this-c;
    float sina = sin(arc), cosa = cos(arc);
    return c + v2d(d.x*cosa-d.y*sina, d.x*sina+d.y*cosa);
}

/* compute the length of the vector */
inline float v2d::len(void) const
{
    return sqrt(x*x+y*y);
}

/* distance between *this and p */
inline float v2d::dist(const v2d &p) const
{
    return sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
}

/* normalize the vector */
inline void v2d::normalize(void)
{
    float l = this->len();
    x /= l; y /= l;
}
```

5.1.2 Summary

- You know how to deal with vectors.
- You implemented the above stuff in linalg.h.

5.2 The Straight Class

Like you will see later it is also nice to have a straight class. It's purpose is to compute intersection points of straights and distances between points and straights.

5.2.1 Implementation

Put the following code also into linalg.h. Like you can see the straights are described with two 2D-vectors. One holds a point on the straight and the other the direction. Because we normalize the direction in the constructor we can simplify the computation of the distance to a point.

```
class Straight {
    public:
        /* constructors */
        Straight() {}
        Straight(float x, float y, float dx, float dy)
            {  p.x = x; p.y = y; d.x = dx; d.y = dy; d.normalize(); }
        Straight(const v2d &anchor, const v2d &dir)
            { p = anchor; d = dir; d.normalize(); }

        /* methods */
        v2d intersect(const Straight &s) const;
        float dist(const v2d &p) const;

        /* data */
        v2d p;          /* point on the straight */
        v2d d;          /* direction of the straight */
};

/* intersection point of *this and s */
inline v2d Straight::intersect(const Straight &s) const
{
    float t = -(d.x*(s.p.y-p.y)+d.y*(p.x-s.p.x))/(d.x*s.d.y-d.y*s.d.x);
    return s.p + s.d*t;
}

/* distance of point s from straight *this */
inline float Straight::dist(const v2d &s) const
{
    v2d d1 = s - p;
    v2d d3 = d1 - d*d1*d;
    return d3.len();
}

#endif // _LINALG_H_
```

5.2.2 Summary

- You know how to deal with straights.
- You implemented the above stuff in linalg.h.

5.3 Steering

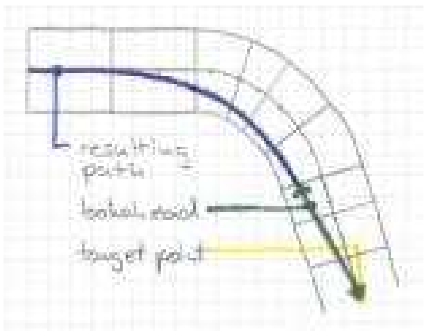


Figure 5.1: steering with lookahead to middle

In this section we will improve the steering with a heuristic approach. In a nutshell, the presented method still drives on the middle of the track, but now we will steer toward a point ahead of the car. The disadvantage of this simple method is that we can't drive with the biggest possible radius through turns like you can see easily on the sketch. This drawing doesn't match exactly with the method presented below, because the lookahead distance will be measured on the middle of the track.

5.3.1 The Target Point

First we discuss how to get the target point. We know the position of our car and the geometry of the track. So we follow the track middle lookahead meters, there is our target point. Put the following code into driver.cpp.

```
v2d Driver::getTargetPoint() {
    tTrackSeg *seg = car->_trkPos.seg;
    float lookahead = LOOKAHEAD_CONST + car->_speed_x*LOOKAHEAD_FACTOR;
```

We compute the lookahead distance with a heuristic. Like you can see lookahead grows with the speed.

```
    float length = getDistToSegEnd();

    while (length < lookahead) {
        seg = seg->next;
        length += seg->length;
    }
```

This loop finds the segment which contains the target point.

```
    length = lookahead - length + seg->length;
    v2d s;
    s.x = (seg->vertex[TR_SL].x + seg->vertex[TR_SR].x)/2.0;
    s.y = (seg->vertex[TR_SL].y + seg->vertex[TR_SR].y)/2.0;
```

Now we compute the distance of the target point to the start of the segment found and the starting point itself. From that we are able to compute the target point in the global coordinate system. We have to distinguish between straights and turns.

```
    if ( seg->type == TR_STR) {
        v2d d;
        d.x = (seg->vertex[TR_EL].x - seg->vertex[TR_SL].x)/seg->length;
        d.y = (seg->vertex[TR_EL].y - seg->vertex[TR_SL].y)/seg->length;
        return s + d*length;
```

For the straight we compute the starting point on the middle of the track and the direction vector. From that we can get the final point and return it.

```
    } else {
        v2d c;
        c.x = seg->center.x;
        c.y = seg->center.y;
        float arc = length/seg->radius;
        float arcsign = (seg->type == TR_RGT) ? -1 : 1;
        arc = arc*arcsign;
        return s.rotate(c, arc);
    }
}
```

For the turns it's a bit different. First we get the rotation center of the segment. Next we compute from the length and the radius the angle of the target point relative to the segments starting point. Finally we rotate the starting point around the center with the computed angle. At the start of driver.cpp we need to define the constants.

```
const float Driver::LOOKAHEAD_CONST = 17.0;     /* [m] */
const float Driver::LOOKAHEAD_FACTOR = 0.33;    /* [1/s] */
```

5.3.2 Heading Toward the Target Point

Now we finally implement the getSteer() method, which computes the steerangle towards the target point. We need atan2() because tan() doesn't work well from -PI..PI ($atan(-y/-x) == atan(y/x)$).

```
float Driver::getSteer()
{
    float targetAngle;
    v2d target = getTargetPoint();

    targetAngle = atan2(target.y - car->_pos_Y, target.x - car->_pos_X);
    targetAngle -= car->_yaw;
    NORM_PI_PI(targetAngle);
    return targetAngle / car->_steerLock;
}
```

Now you need to change Driver::drive() so that getSteer is called: replace

```
        float steerangle = angle - car->_trkPos.toMiddle/car->_trkPos.seg->width;
        car->ctrl.steer = steerangle / car->_steerLock;
```

with

```
        car->ctrl.steer = getSteer();
```

5.3.3 Finishing Implementation

Finally we need some changes in driver.h. The new methods and constants needs to be defined, and we also need to include linalg.h.

```
#include "linalg.h"

        float getSteer();
        v2d getTargetPoint();

        static const float LOOKAHEAD_CONST;
        static const float LOOKAHEAD_FACTOR;
```

5.3.4 Testdrive

Do some test runs with the different cars. The car drives now very nice on most of the tracks. Try also e-track-3 and have a look on the first left turn after the finish line. In the next chapter we will implement a heuristic to pass turns of that type faster.

5.3.5 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

5.3.6 Summary

- You have implemented the above stuff.
- You know how to steer with a heuristic approach.

5.4 Reduce the Lap Time

In this section we modify Driver::getAllowedSpeed, so that we can pass short turns faster. A ”short” turn in this context means that the angle of the turn is small. You have seen that on the new path the radius we drive is larger than the radius in the middle of the track. As heuristic approximation we take first the turns outside radius. Further we divide the radius by the square root of the remaining angle of the turn (remember, this is a heuristic!). This will lead to a problem, but we will fix that later. An interesting feature of this approach is the growing acceleration toward the end of a turn.

5.4.1 Implementation

Replace getAllowedSpeed in driver.cpp with the following version:

```
/* Compute the allowed speed on a segment */
float Driver::getAllowedSpeed(tTrackSeg *segment)
{
    if (segment->type == TR_STR) {
        return FLT_MAX;
    } else {
```

No changes so far. In the next part we compute the remaining angle (arc) of the turn. We need a loop because large turns can be composed from a lot of small turns.

```
        float arc = 0.0;
        tTrackSeg *s = segment;

        while (s->type == segment->type && arc < PI/2.0) {
            arc += s->arc;
            s = s->next;
        }
```

Because we divide the radius by this angle it makes no sense to have values greater than 1.0, so we normalize it.

```
        arc /= PI/2.0;
        float mu = segment->surface->kFriction;
```

Compute the magic radius.

```
        float r = (segment->radius + segment->width/2.0)/sqrt(arc);
        return sqrt((mu*G*r)/(1.0 - MIN(1.0, r*CA*mu/mass)));
    }
}
```

5.4.2 Testdrive

Now we come close to some opponents with the lap times. There are just a few seconds on some tracks left. In the next section we will fix the newly introduced stability problem.

5.4.3 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

5.4.4 Summary

- You are really hot to beat some opponents.

5.5 Stay on the Track

Now we make the car stay again on the track (at least on most of the tracks). We will introduce a margin around the middle of the track. When we are farther away from the middle we set the accelerator to zero. We will refine this by distinguish between inside and outside in the turns. When we are inside we can accelerate further, because the centrifugal force pushes the car outside.

5.5.1 Implementation

Here follows the implementation, put it in driver.cpp.

```
/* Hold car on the track */
float Driver::filterTrk(float accel)
{
    tTrackSeg* seg = car->_trkPos.seg;

    if (car->_speed_x < MAX_UNSTUCK_SPEED) return accel;
```

If we are very slow do nothing. Then check if we are on a straight. If that's the case, check both sides.

```
    if (seg->type == TR_STR) {
        float tm = fabs(car->_trkPos.toMiddle);
        float w = seg->width/WIDTHDIV;
        if (tm > w) return 0.0; else return accel;
```

If we are in a turn check if we are inside or outside. If we are inside do nothing.

```
    } else {
        float sign = (seg->type == TR_RGT) ? -1 : 1;
        if (car->_trkPos.toMiddle*sign > 0.0) {
            return accel;
```

If we are outside and more than "w" from the middle away set accelerator to zero.

```
        } else {
            float tm = fabs(car->_trkPos.toMiddle);
            float w = seg->width/WIDTHDIV;
            if (tm > w) return 0.0; else return accel;
        }
    }
}
```

We need to define WIDTHDIV in driver.cpp.

```
const float Driver::WIDTHDIV = 4.0;     /* [-] */
```

Add the call in Driver::drive(). Change

```
            car->ctrl.accelCmd = filterTCL(getAccel());
```

to

```
            car->ctrl.accelCmd = filterTCL(filterTrk(getAccel()));
```

Finally we declare the new method and constant in driver.h.

```
        float filterTrk(float accel);

        static const float WIDTHDIV;
```

5.5.2 Testdrive

What you got till now is not too bad. You've written a robot that is able to master most of the tracks. We also reduced the lap times very much:

* mixed-2: 1:14:70, 0 damage. * e-track-2: 1:24:45, 0 damage. * e-track-4: 1:49:70, 0 damage.

5.5.3 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

5.5.4 Summary

- You have implemented and understood an almost competitive robot.
- Congratulations:-)

5.6 Trajectories

5.6.1 The Optimal Trajectory?

In case you know how to compute the optimal trajectory very fast (optimal means minimal lap time) please let me know... I skimmed quite a lot of papers about that subject (the robotics community deals with similiar problems) but from there I didn't get much useful information. If I remember right there has been proved that finding the optimal trajectory in three dimensions is in NP. But because our tracks are quite flat a two dimensional approximation would be sufficient. There have been written some papers about that subject and if I remember right there are also solutions available, but just for a mass point robot (no aerodynamics involved) and quite expensive to compute (or should I say simply "slow"). If you are interested the papers are available in the ACM online library.

5.6.2 Heuristic Trajectories

Like you have seen heuristic means to find a trajectory with some reasonable ideas. So we improve our trajectory with adding this and that, improvement x causes problem y and so forth... The problem with heuristics is that they may work great on some tracks but also horribly fail on some other tracks or dislike certain features. I think you can build quite a good driver with this method, but there will be always problems with some track features.

5.6.3 Geometric Trajectories

I call a trajectory geometric when it just takes the track geometry into account, but doesn't consider the car setup and aerodynamics. A nice implementation is in K1999 from RÃ©mi Coulom. It does linearize the curvature of the path in a quite efficient and clever way. But it doesn't deliver optimal paths especially with high downforce car setups in fast turns.

What else did I try so far? My first approach was to find a few initial "good" points on the track and then to put a cubic spline trough it. Now the spline leaves the track on many sections of the track. So I find the place where the spline leaves the track first and insert an additional point on the track. You repeat that till the whole path is on the track. That works quite well (never hit the public).

The next try was to wrap clothoids trough the turns. There is still an implementation of that available in berniw, but not very well refined.

Another experiment (for that tutorial) was to work with the biggest arcs you can fit into the turns. It is not yet refined enough for public release and too complicated for the tutorial.

5.6.4 Machine Learning Approaches

I think TORCS would also be a nice playground for machine learning techniques. So if you are a professor let your students write some robots for TORCS.

5.6.5 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

5.6.6 Summary

- You know about other approaches.
- You have an almost competitive car.

Chapter 6

Using Custom Car Setups

6.1 What is a Car Setup

Racing cars have a lot of parameters one can change. For example you can change the angle of attack of the spoilers, the gears in the gearbox, the stiffness of the springs and many more. A car setup is simply a set of such parameters. Our task is to find an optimal car setup for a given goal. This is a very hard problem, because the parameters are not independent. In fact it's almost impossible to find the optimal setup. Why that? You can imagine all possible setups as n dimensional function (with n the number of parameters) which computes the lap time. We want to find the minimal lap time, so we have to minimize this function. Because this function has (at least!) around 10 dimensions it's almost impossible to find the global minimum. What you can get is a local minimum, which can be much worse than the optimal (global) one.

6.1.1 Qualifying Setup

In the qualifying the goal is to get the minimal lap time. Because there are no other opponents on the track we can drive at the limit. We also just need fuel for a few laps. The top speed on straights doesn't matter as long we reach the minimal lap time.

6.1.2 Racing Setup

The racing setup is a bit different. Of course you still would like to minimize the lap time, but other things become also important. For example it is much easier to overtake on a straight than in a turn, so you want to optimize the setup also for top speed on a straight, altough it probably will increase the lap time. The car needs also to be more stable, so that it doesn't spin of the track when it tries to overtake or to avoid a collision. Keep in mind if you want to win a race you have to reach at least the finish line...

6.1.3 Skill Level

There are four skill levels available, namely rookie, amateur, semi-pro, and pro. On rookie you have a lot of adherence and no damages, with pro you get a lot of damage and less adherence. I mention that here, because this affects the setups. The default skill level is semi-pro. You can configure it for every car individually in the bt.xml file. Add the skill level line in the section of the car you want to configure, e. g. for the car with index 0:

```
<section name="0">
    <attstr name="skill level" val="semi-pro"/>
</section>
```

6.1.4 Summary

- You know what a car setup is.
- You know that for different goals different setups are required.
- You know how to change the skill level of your cars.

6.2 Loading Custom Car Setups

6.2.1 Introduction

First lets have a look on where the default settings for the cars are coming from. They are loaded automatically from the cars type default settings XML file. These files are located in the cars/* directories, e. g. for the cg-nascar-rwd it is the file cars/cg-nascar-rwd/cg-nascar-rwd.xml (relative to /usr/local/share/games/torcs). This happens always, so if we load no settings ourselves, the default settings are active. If we load a custom setup it just overwrites the default settings, if we don't specify a certain value the default value is active. To create a custom setup we make just a copy of the default setup, remove the stuff we don't want to change and edit the values.

6.2.2 Loading Setups

The following implementation will first try to load a custom car setup for the current track. If there is no such setup available, it falls back to a default custom setup. If this fails also, the TORCS default setup stays active. Our custom setup files will be located e. g. for the driver with index 0 in the subdirectories 0, 0/practice, 0/qualifying and 0/race (relative to the bt directory). Put the following code into driver.cpp, method Driver::initTrack().

```
/* Called for every track change or new race. */
void Driver::initTrack(tTrack* t, void *carHandle,
                       void **carParmHandle, tSituation *s)
{
    track = t;
```

First we compute a pointer to the tracks filename from the whole path. The filename starts after the last "/", so we need to add 1 to the location found with strrchr.

```
    char buffer[256];
    /* get a pointer to the first char of the track filename */
    char* trackname = strrchr(track->filename, '/') + 1;
```

Depending on the race type, we want to load the fitting setup. For that we assemble the required path to our setup file relative to the /usr/local/share/games/torcs directory.

```
    switch (s->_raceType) {
        case RM_TYPE_PRACTICE:
            sprintf(buffer, "drivers/bt/%d/practice/%s", INDEX, trackname);
            break;
        case RM_TYPE_QUALIF:
            sprintf(buffer, "drivers/bt/%d/qualifying/%s", INDEX, trackname);
            break;
        case RM_TYPE_RACE:
            sprintf(buffer, "drivers/bt/%d/race/%s", INDEX, trackname);
            break;
        default:
            break;
    }
```

Now we try to load the setup file. After that we check if the file has been loaded. If not we assemble the path to the default setup file and try to read it. If you are not sure about the setup file name you need for a certain track you can add a printf or cout to print the filename to the terminal.

```
    *carParmHandle = GfParmReadFile(buffer, GFPARM_RMODE_STD);
    if (*carParmHandle == NULL) {
        sprintf(buffer, "drivers/bt/%d/default.xml", INDEX);
        *carParmHandle = GfParmReadFile(buffer, GFPARM_RMODE_STD);
    }
}
```

Because we haven't created any setups yet the loader will fail for now. In the next section we will create the directory structure for the setups.

6.2.3 Summary

- You know where the default setup files are located.
- You know how to create custom setup files.
- You have implemented the above setup loader.

6.3 Creating the Directory Structure

In this section we will create the directory structure and the Makefiles for the cg-nascar-rwd (the car with index 0). For the other cars it works the same way. We need Makefiles just to ease the deployment, we could also copy the setup files manually to the torcs directory.

6.3.1 Creating the Directories

Make sure you are in the bt source directory:

```
$ cd $TORCS_BASE/src/drivers/bt
```

Now we create the subdirectories for the setups:

```
$ mkdir -p 0/practice
$ mkdir 0/qualifying
$ mkdir 0/race
```

For another car replace 0 with its index and do it the same way.

6.3.2 The Makefiles

First we change the main Makefile of bt to process the subdirectory 0. Change the line

```
SHIPSUBDIRS =
```

to

```
SHIPSUBDIRS = 0
```

When you want to add more subdirectories append them to this line, e. g. "0 1". Now create the Makefile in directory 0 with the following content:

```
ROBOT        = bt
SHIPDIR      = drivers/${ROBOT}/0
SHIP         = default.xml
SHIPSUBDIRS = practice qualifying race

PKGSUBDIRS   = ${SHIPSUBDIRS}
src-robots-bt_PKGFILES = $(shell find * -maxdepth 0 -type f -print)
src-robots-bt_PKGDIR    = ${PACKAGE}-${VERSION}/$(subst ${TORCS_BASE},,$(shell pwd))
include ${MAKE_DEFAULT}
```

Like you can see it deploys the file default.xml and the subdirectories practice, qualifying and race. In this subdirectories we need also Makefiles to deploy the setup files. Here is the Makefile for the 0/practice directory.

```
ROBOT        = bt
SHIPDIR      = drivers/${ROBOT}/0/practice
SHIP         = $(shell find *.xml -maxdepth 0 -type f -print)
```

```
src-robots-bt_PKGFILES = $(shell find * -maxdepth 0 -type f -print)
src-robots-bt_PKGDIR   = ${PACKAGE}-${VERSION}/$(subst ${TORCS_BASE},,$(shell pwd))
include ${MAKE_DEFAULT}
```

Here the Makefile for the 0/qualifying directory.

```
ROBOT       = bt
SHIPDIR     = drivers/${ROBOT}/0/qualifying
SHIP        = $(shell find *.xml -maxdepth 0 -type f -print)

src-robots-bt_PKGFILES = $(shell find * -maxdepth 0 -type f -print)
src-robots-bt_PKGDIR   = ${PACKAGE}-${VERSION}/$(subst ${TORCS_BASE},,$(shell pwd))
include ${MAKE_DEFAULT}
```

And finally the Makefile for the 0/race directory.

```
ROBOT       = bt
SHIPDIR     = drivers/${ROBOT}/0/race
SHIP        = $(shell find *.xml -maxdepth 0 -type f -print)

src-robots-bt_PKGFILES = $(shell find * -maxdepth 0 -type f -print)
src-robots-bt_PKGDIR   = ${PACKAGE}-${VERSION}/$(subst ${TORCS_BASE},,$(shell pwd))
include ${MAKE_DEFAULT}
```

Before the Makefiles work there has to be at least one XML file in every directory. Put this stripped down default setup of cg-nascar-rwd into the file 0/default.xml. Copy it also to the following locations:

```
$ cp 0/default.xml 0/practice/g-track-3.xml
$ cp 0/default.xml 0/qualifying/g-track-3.xml
$ cp 0/default.xml 0/race/g-track-3.xml
```

This should be enough for a litte test run. Do a "make install", check if it works without errors and look up in the target directory if everything is in place. In the next sections we will work with these files.

6.3.3 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

6.3.4 Summary

- You know the directory structure.
- You have created all files.
- You have checked if the installation works.

6.4 Car Properties

This section will introduce you in the format of the setup files, the settings constraints and the most important properties (parameters, settings) of the TORCS cars. If you look up the default settings files there are many more settings available than presented here, but most of them you can't change (e. g. the engine). I want to mention here that you can set all these properties up without a setup file, so you are able to compute settings at the startup of the race.

6.4.1 Setup File Format

The setup files are XML files. The structure is very simple, you will see just sections, subsections and attributes. Numerical attributes are usually defined with a name, a unit and the boundaries which define the range of the allowed values.

6.4.2 Settings Constraints

The car belongs to a car category, which defines some constraints like the range of the allowed size, engine power, weight and more. The car model settings defines then the values for its specific car and must not conflict with the constraints from the car category. After that we will load our own setup files which must not violate the constraints from the above definitions.

6.4.3 Properties

You can find all mentioned properties in the file 0/default.xml (relative to the bt directory).

6.4.4 Initial Fuel

```
  <section name="Car">
    <attnum name="initial fuel" unit="l" min="1.0" max="100.0" val="100.0"/>
    </section>
```

The amount of initial fuel for the setup. Perhaps you want to compute that setting at startup instead of putting it into a setup file. Feel free to improve it. For qualifying and short races you want just the minimal amount of fuel you need, for longer races it will depend on your strategy (but we need to implement pit stops first).

6.4.5 Spoilers

```
    <section name="Rear Wing">
        <attnum name="angle" unit="deg" min="0" max="30" val="10"/>
    </section>
```

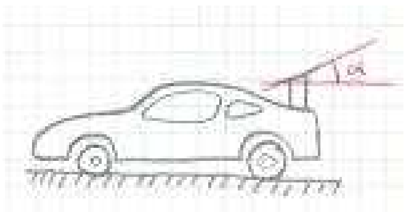


Figure 6.1: spoiler angle of attack

This section defines the angle of attack of the rear spoiler relative to the floor. When you increase the angle of attack you get more downforce and drag. If the race track has very long straights and just a few narrow turns you want a small angle of attack to reduce the drag. If there are many turns you want a big angle of attack to increase the downforce, so you can drive faster through the turns. For some cars there is also a front wing section available. The front spoiler helps to avoid understeering if you set up the rear spoiler with a high angle of attack.

6.4.6 Gearbox

```
    <section name="Gearbox">
        <section name="gears">
            <section name="r">
                <attnum name="ratio" min="-3" max="0" val="-2.0"/>
            </section>

    ...
```

```
        </section>
    </section>
```

The gearbox section defines the properties of the gears. The ″r″ subsection specifies the ratio of the reverse gear. The reverse gear ratio needs to be negative. The wheel velocity is proportional to the engine rpm divided by the ratios, so for the highest gear you need the least value. For quick starts you need a very high value for the first gear.

6.4.7 Differential

```
    <section name="Rear Differential">
        <attstr name="type" in="SPOOL,FREE,LIMITED SLIP" val="LIMITED SLIP"/>
    </section>
```

Here the differential type is defined. The ″Free″ type distributes the power without any control, so it is the classic differential you can find in most of the normal cars. The ″Free″ differential is a good choice for tracks with high friction and not many bumps for the qualifying. When you leave the track with the ″Free″ differential you perhaps get stuck in the grass, sand or dirt.

The ″Limited Slip″ differential allows just a certain amount of speed difference between the left and right wheel. This allows you to come back to the track from grass and dirt. This is most often the best setting for races.

The ″Spool″ setting links together the right and the left wheel, so it's usually a bad setting for all turns. It could make sense on a very bumpy road or for offroad races.

6.4.8 Brakes

```
    <section name="Brake System">
        <attnum name="front-rear brake repartition" min="0.3" max="0.7" val="0.52"/>
        <attnum name="max pressure" unit="kPa" min="100" max="150000" val="11000"/>
    </section>
```

The front-rear brake repartition defines how to distribute the pressure in the brake system. If the value is 0.0 all pressure goes to the rear brakes, if it is 1.0 all pressure hits the front brakes. The max pressure setting defines the pressure for applying full brakes.

6.4.9 Wheels

```
<section name="Front Right Wheel">
    <attnum name="ride height" unit="mm" min="100" max="200" val="100"/>
    <attnum name="toe" unit="deg" min="-5" max="5" val="0"/>
    <attnum name="camber" min="-5" max="0" unit="deg" val="0"/>
</section>
```

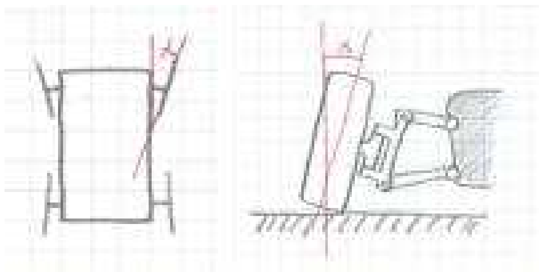


Figure 6.2: toe and camber

The ride height setting defines the initial distance of the cars floor to the track. When you decrease the ride height the downforce increases because of the ground effect, but you cars floor may hit the track. With toe (on the left on the sketch) you can define the initial angle of the wheel to the car. You can improve the stability or the response to steer commands with these settings. Camber (on the right on the sketch) defines the angle of the wheel to the track. At the moment it increases simply the adherence on the track in the simulation when you decrease the angle (toward negative values).

6.4.10 Anti Roll Bar

```
<section name="Rear Anti-Roll Bar">
    <attnum name="spring" unit="lbs/in" min="0" max="5000" val="0"/>
    <attnum name="suspension course" unit="m" min="0" max="0.2" val="0.2"/>
    <attnum name="bellcrank" min="1" max="5" val="1.8"/>
</section>
```

The anti-roll bars prevents the car body from rolling (e. g. in fast turns). When the car body rolls (on the bottom of the sketch) the tire contact patch

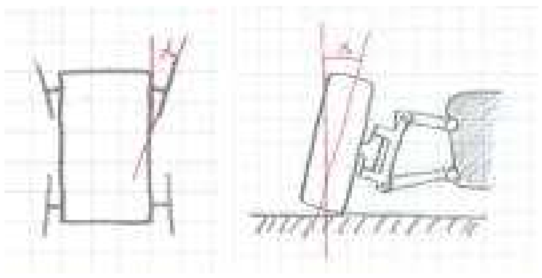


Figure 6.3: rotation of car body

becomes smaller and the grip drops down. The spring setting defines how strong the anti-roll bar is linked. With a high bellcrank value you can increase the strength of the link further.

6.4.11 Suspension

```
<section name="Front Right Suspension">
    <attnum name="spring" unit="lbs/in" min="0" max="10000" val="2500"/>
    <attnum name="suspension course" unit="m" min="0" max="0.2" val="0.2"/>
    <attnum name="bellcrank" min="1" max="5" val="2"/>
    <attnum name="packers" unit="mm" min="0" max="10" val="0"/>
    <attnum name="slow bump" unit="lbs/in/s" min="0" max="1000" val="80"/>
    <attnum name="slow rebound" unit="lbs/in/s" min="0" max="1000" val="80"/>
    <attnum name="fast bump" unit="lbs/in/s" min="0" max="1000" val="100"/>
    <attnum name="fast rebound" unit="lbs/in/s" min="0" max="1000" val="100"/>
</section>
```

The spring setting defines how stiff the spring is. The suspension course defines the distance which the suspension can move. With the bellcrank you can make the whole suspension system more stiff (higher values) or soft (lower values). The packers limit the suspension movement. You need this if you want a soft setup and avoiding the car floor hitting the track on fast track sections. Slow bump and rebound allow you to set up the damping of low frequency oscillations, e. g. caused by braking and steering. Fast bump and rebound are for damping high frequency oscillations, e. g. caused by bumps on the track or if you hit the curbs.

6.4.12 Summary

- You know the most important car parameters and what they are good for.

6.5 Add Custom Properties

6.5.1 Introduction

In this section you will learn how to add your own properties to the setup file and how to load it. As example we store a new property which contains the fuel required for one lap. We load this property and compute from that the whole amount of fuel we need for the race. Then we pass that back to the simulation, so that our car starts with the ideal amount of fuel.

6.5.2 Implementation

First I show you the changes needed in the setup file. For the example we take the driver bt 1 on g-track-3 in practice mode. From that follows that we have to modify the file 0/practice/g-track-3.xml. Add the following to the file above from ¡/params¿.

```
    <section name="bt private">
        <attnum name="fuelperlap" val="2.0"/>
    </section>
```

We define a new section type for all our private settings. Inside the section we define our new property "fuelperlap" as numeric property and assign the value 2.0. You could also put your properties into another file.

Now we add the code to load the property to driver.cpp. First we define new constants to access the section and the attribute.

```
#define BT_SECT_PRIV "bt private"
#define BT_ATT_FUELPERLAP "fuelperlap"
```

The code to load and initialize settings has to be in Driver::initTrack(). Append the following code at the end of the method.

```
    float fuel = GfParmGetNum(*carParmHandle, BT_SECT_PRIV,
        BT_ATT_FUELPERLAP, (char*)NULL, 5.0);
```

Try to load our property. If it fails the value defaults to 5.0. After that compute the required fuel for the race. That's the number of laps times the fuel required per lap. To play safe we add fuel for an additional lap. After that set the initial fuel up.

```
    fuel *= (s->_totLaps + 1.0);
    GfParmSetNum(*carParmHandle, SECT_CAR, PRM_FUEL, (char*)NULL, MIN(fuel, 100.0));
```

If you run a practice session with bt 1 on g-track-3 the initial fuel should be 42.0 if everything works.

6.5.3 Accessing Subsections

You have seen that it is possible to have subsections in the setup files, so that it is possible to structure the parameters. But how do we access parameters in subsections? As example we assume that we add a subsection ″test″ in our private section, which contains the parameter ″cool″.

```
    <section name="bt private">
        <attnum name="fuelperlap" val="2.0"/>
        <section name="test">
            <attnum name="cool" val="2.0"/>
        </section>
    </section>
```

To access the parameter ″cool″ you have to assemble a path for the subsection, which contains simply the section names top down to the subsection separated by ″/″. So the path would be ″bt private/test″. Here is an example how to implement it.

```
#define BT_SECT_PRIV_TEST "test"
#define BT_ATT_COOL "cool"

float cool;
char path[256];
sprintf(path, "%s/%s", BT_SECT_PRIV, BT_SECT_PRIV_TEST);
cool = GfParmGetNum(*carParmHandle, path, BT_ATT_COOL, (char*)NULL, 0.0);
```

6.5.4 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

6.5.5 Summary

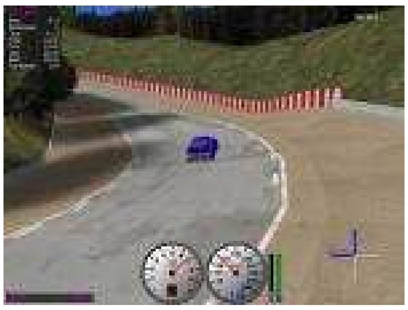
```
    \item You know how to load custom properties.
    \item You know how to access and create subsections.
    \item You know how to set up your car with computed values.
```

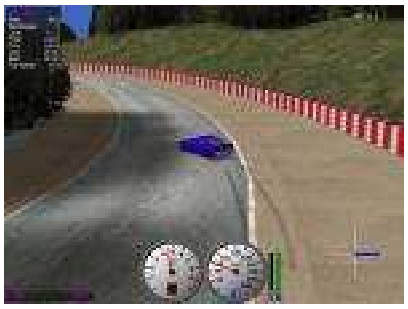
6.6 Setup the Car

In this section we will set up the car bt 1 for the qualifying on g-track-3. The goal is that you get a feeling for the parameters you can change. I don't recommend you to put too much effort in car setups yet, because if you modify the driver or write one from scratch it will not need the same setups. We do the test runs in the practice mode, set laps to 2 and display to normal so that you can analyse the behaviour. You will need to edit the file "0/practice/g-track-3.xml" (relative to the bt directory). To deploy the changed setup you have to run "make install" from the directory "0/practice".

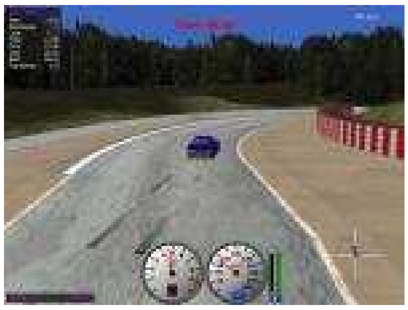
6.6.1 Test Drive 1

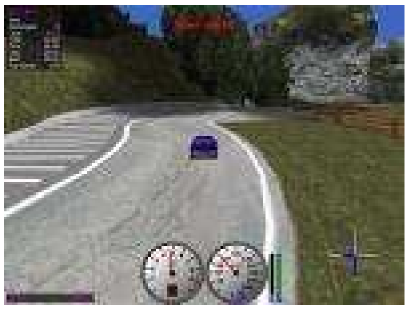
First we run a test without any changes. The lap time is 1:08:86.





You can see on picture 1 the heavy understeering of the car in this narrow turn. It happens in the acceleration part of it, so we can conclude that the "Limited Slip" differential on the driven rear wheels disallows us to pass the turn. So we will set the differential to type "Free". On picture 2





you can see the result of the understeering. The car leaves the track and starts skidding.

On picture 3 you can see the skidmarks form the braking before the turn. You can conclude that the braking balance or pressure is not set up correct. Perhaps we have to move the braking balance towards the rear wheels or to reduce the pressure.

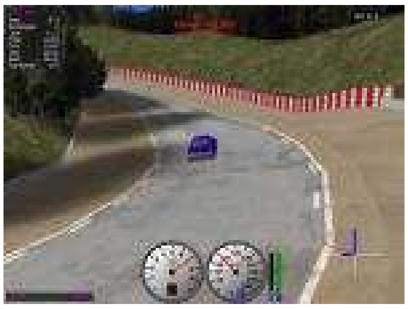
Finally on picture 4 you can see ”classic” understeering. It’s a quite wide turn, so the main cause is probably not enough grip on the front wheels. We can fix that with setting the camber to the minimum value and with proper suspension setup. Perhaps we have to reduce the angle of attack of the rear wing (rebember our speed limit computation). Now make the following changes in the setup file (0/practice/g-track-3.xml) and deploy it (make install).

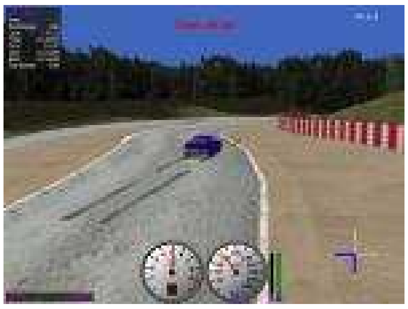
- Set the differential to type ”FREE”.

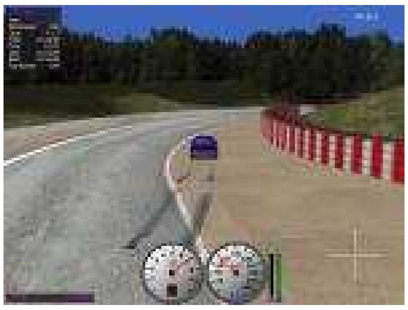
- Set camber of the left and right front wheel to -5.

6.6.2 Test Drive 2

The resulting lap time is 1:07:85, that's 1.01 seconds faster than the previous result.

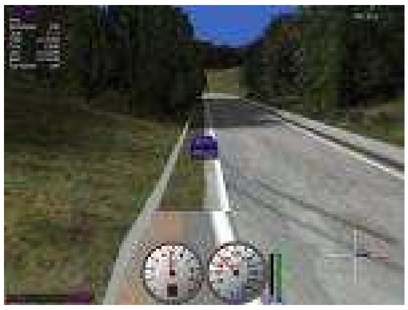






Like you can see on picture 5 we pass this turn now fine without any skidmarks or other trouble. But we face a new problem on the next turn like you can see on the pictures 6, 7 and 8. If you watch the practice session carefully you can observe that the reason for this are locking front wheels. We can conclude that we have to move the braking balance toward the rear wheels and to reduce the pressure.

- Set the brakes "front-rear brake repartition" to 0.47.
- Set the "max pressure" to 9000.



6.6.3 Test Drive 3

The resulting lap time is 1:07:03, that's 1.83 seconds faster than the first result. The car stays now quite good on the track so you can start the refinement of the setup. After I changed the gearbox, suspension, brake and rear wing settings I got finally a 1:05:95, that's 2.88 seconds faster than the default setup. Play now with different values to get a feeling for the settings. An approach to find the optimal values is:

1. Pick a parameter to optimize.
2. Choose a value for the parameter, e. g. the default value.
3. Measure the lap time.
4. Choose a smaller value for the parameter and measure the lap time.
5. Choose a larger value for the parameter and measure the lap time.
6. Explore new values in the direction of the minimal lap time found.
7. If the left and right values produce worse lap times you have found the best local value, choose now another parameter for optimization.
8. Repeat the above till the setup is good enough.
9. Finally copy the setup to the right directory.

You have also to keep an eye on the damage you get per lap. Because you drive just a few laps in the qualifying it doesn't matter if you get some damage. But for long races it can be a problem. Damage causes additional drag and if it reaches a certain limit your car is not able to drive further.

6.6.4 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

6.6.5 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

6.6.6 Summary

- You know how to setup the car.
- You have got a feeling for the different settings.

Chapter 7

Pit Logo and Painting Your Car

7.1 The Look

In this chapter you will learn the basics you need for giving your team an individual look. I'm not an artist, so I can't give you tips how you can make it look really cool. You might have guessed that from the purple look of my team, don't you? But hey, have you ever seen purple racing cars before? There are two things you can change per car, the look of the car and the pit logo. While the general look is a matter of taste you should take care of things like political correctness and legal issues of advertisements and logos. Below you can see some screenshots of cars in front of their pit.

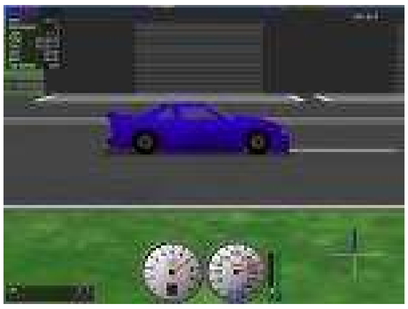


Figure 7.1: The default

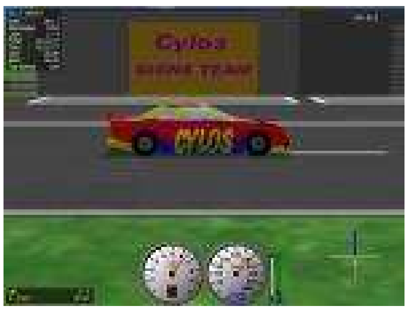


Figure 7.2: Cylos 2 of Gions Team

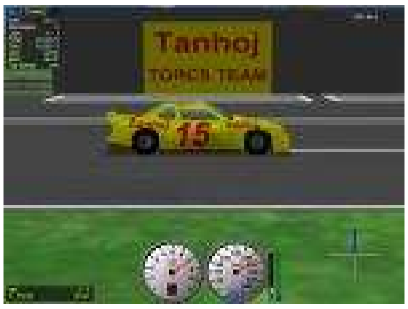


Figure 7.3: Tanhoj of TORCS Team

7.1.1 Textures

The pictures that define the look of your car are called textures. These textures are put onto your car in a process called texture mapping. Car and pit logo textures have to be stored in the SGI rgb format. The size must be a power of two, eg. for the pit logo 128x128 and for the car 256x256. Textures need to be quadratic (width equals height). You can see the default texture of the cg-nascar-rwd car on the left. To create your custom car texture you need to download a default texture and to modify it. To make your work easier there are for some cars textures available as multilayer xcf images in several resolutions. To work with the textures I recommend you to take the highest resolution available, you can reduce it later with your tool.

7.1.2 Tools

To work with the textures I recommend the GIMP (GNU Image Manipulation Program). It is for sure included in your Linux distribution. There

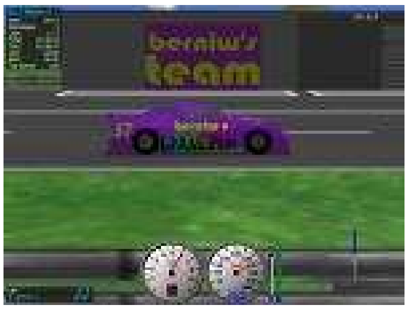


Figure 7.4: Berniw 1 of my Team

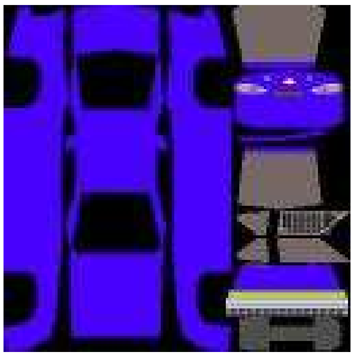


Figure 7.5: car texture

is also a Windows version available.

7.1.3 Summary

- You know what a texture is.
- You know textures need to be quadratic.
- You have a tool to work with xcf and rgb images.

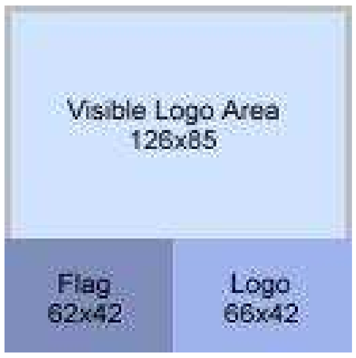


Figure 7.6: logo anatomy

7.2 Creating the Pit Logo

7.2.1 Anatomy of the Pit Logo

The image on the left shows the anatomy of the pit logo for a resolution of 128x128 pixels. In the following discussion I will put the numbers for a 256x256 logo in braces. Lets start with the grey border. This border defines the colour of the pit wall outside the logo area. Its width has to be one pixel, independent of the logo resolution. The visible logo sits inside this border and has a size of 126x85 (254x171) pixels. In the lower left part of the logo you should put the flag of your country (most often images of the flag are available on your country's homepage), the size is 62x42 (124x84) pixels. On the bottom right is space left for another logo of your team with 66x42 (132x84) pixels. I think the lower logo parts will be used in the future for driver selection and results.

7.2.2 Create the Logo

You can create your logo with the GIMP. When you finished your pit logo save it in the SGI format with the name "logo.rgb". Put it into the bt source directory if you want that logo for the whole team or into the bt/0 directory if you want it just for the car bt 1.

7.2.3 Changing the Makefile

Here I assume that we want to deploy one logo for the whole team, so the logo is in the bt directory. Now change in the Makefile in this directory the line

```
SHIP            = ${ROBOT}.xml
```

to

```
SHIP            = ${ROBOT}.xml logo.rgb
```

If you want to put the logo in the car directory you have to put the logo there and to edit the Makefile there. After you have installed the logo with "make install" it should appear in the simulation.

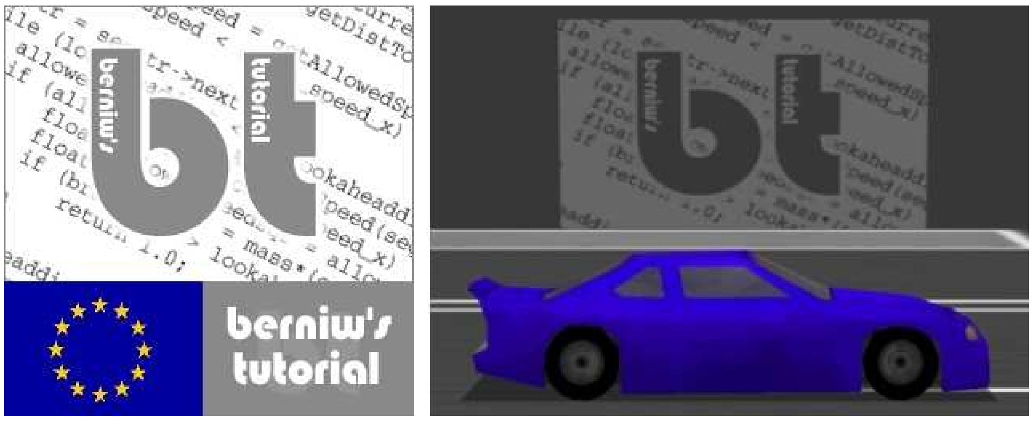


Figure 7.7: bt's logo

7.2.4 Summary

- You know how to create a pit logo.
- You know how to integrate the logo.

7.3 Painting the Car

7.3.1 Overview

To paint your car you have to do the following:

- Download a default texture from CVS.
- Modify the default texture.
- Save the texture in the SGI rgb format.
- Deploy the texture.

7.3.2 Download the Default Texture

For some cars there is a default texture as multilayered xcf file available. If it's the case you should always download this file. You can download the files here. Go into the directory of your cars type, click on the xcf file (if available, else you have to take the rgb) in the first column, then right click the download link and choose "save link target as...".

7.3.3 Modify the Default Texture

If you downloaded an rgb image I recommend you to convert it into an xcf. With xcf you can use layers, which simplifies later changes to the texture (e. g. you want to change the racing number, you have just to edit the racing number layer). Don't throw this xcf away, it is much more comfortable to work with that than with rgb images.

7.3.4 Save the Texture in the SGI rgb Format

The texture is for a specific car, so you have to put it into the cars subdirectory. Say for bt 1 you should save the image as SGI rgb file into the bt/0 source directory.

7.3.5 Deploy the Texture

To deploy the texture you have to modify the Makefile in the cars directory. If we want to deploy the texture of bt 1 you have to modify the Makefile in bt/0. Change

```
SHIP            = default.xml
```

to

```
SHIP            = default.xml cg-nascar-rwd.rgb
```

After you have deployed the texture with "make install" it should appear in the simulation.

7.3.6 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

7.3.7 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

7.3.8 Summary

- You know where and how to get car textures.
- You know about xcf and rgb files.
- You know how to deploy your cars texture.

Chapter 8

Collision Avoidance and Overtaking

8.1 Collecting Data About Opponents

In this chapter we implement the ability to avoid collisions and to overtake opponents. I show you quite simple implementations, therefore there is much room left for improvements. For me it is one of the most interesting parts of the robots, because you can implement it in very different ways and it is not easy to predict which ideas will work well and which will not. The following implementation starts with collecting data about the opponents. After that we implement collision avoidance and finally overtaking.

8.1.1 The Opponent Class

First we define a class for our opponents. It holds data about the opponents relative to our robot, e. g. the distance between opponent X's car and ours. For simplicity I will also put some data into it which is just dependent of the opponent. This is not efficient if we run more than one robot with our module, it would be better to put that data into a separate class whose instances are visible from all robots of our module and are just updated once per timestep (e. g. when we start bt 1 and bt 2 both will compute the speed of opponent X, that is neither efficient nor necessary). You can achieve such an update with checking the simulation time and just update the data when the simulation time has changed. The code would then look somelike that:

```
static double currenttime;   /* class variable in Driver */
...
if (currenttime != situation->currentTime) {
    currenttime = situation->currentTime;
    allcars.update();
}
```

8.1.2 The Opponents Class

The Opponents class will just hold an array of opponents.

8.1.3 The opponent.h File

Put the following stuff into opponent.h in the bt directory. I will explain the details in the discussion of opponent.cpp.

```
/***************************************************************************

    file                 : opponent.h
    created              : Thu Apr 22 01:20:19 CET 2003
    copyright            : (C) 2003 Bernhard Wymann

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef _OPPONENT_H_
#define _OPPONENT_H_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <tgf.h>
#include <track.h>
#include <car.h>
#include <raceman.h>
#include <robottools.h>
#include <robot.h>
```

```
#include "linalg.h"
#include "driver.h"
```

The following lines define bitmasks which we will use to classify the opponents. I will explain them later.

```
#define OPP_IGNORE      0
#define OPP_FRONT       (1<<0)
#define OPP_BACK        (1<<1)
#define OPP_SIDE        (1<<2)
#define OPP_COLL        (1<<3)
```

Because of a cross dependency we need to define a Driver class prototype. After that the declaration of the Opponent class starts.

```
class Driver;

/* Opponent maintains the data for one opponent */
class Opponent {
    public:
        Opponent();

        void setCarPtr(tCarElt *car) { this->car = car; }
        static void setTrackPtr(tTrack *track) { Opponent::track = track; }

        static float getSpeed(tCarElt *car);
        tCarElt *getCarPtr() { return car; }
        int getState() { return state; }
        float getCatchDist() { return catchdist; }
        float getDistance() { return distance; }
        float getSideDist() { return sidedist; }
        float getWidth() { return width; }
        float getSpeed() { return speed; }

        void update(tSituation *s, Driver *driver);

    private:
        float getDistToSegStart();

        tCarElt *car;       /* pointer to the opponents car */
        float distance;     /* approximation of the real distance */
        float speed;        /* speed in direction of the track */
        float catchdist;    /* distance needed to catch the opponent */
        float width;        /* the cars needed width on the track */
        float sidedist;     /* distance of center of gravity of the cars */
        int state;          /* state bitmask of the opponent */

        /* class variables */
        static tTrack *track;

        /* constants */
        static float FRONTCOLLDIST;
        static float BACKCOLLDIST;
        static float SIDECOLLDIST;
        static float LENGTH_MARGIN;
        static float SIDE_MARGIN;
};
```

Finally follows the declaration of the Opponents class. Like you can see it is just responsible for the update of the opponents.

```
/* The Opponents class holds an array of all Opponents */
class Opponents {
    public:
        Opponents(tSituation *s, Driver *driver);
        ~Opponents();

        void update(tSituation *s, Driver *driver);
        Opponent *getOpponentPtr() { return opponent; }
        int getNOpponents() { return nopponents; }

    private:
        Opponent *opponent;
        int nopponents;
};


#endif // _OPPONENT_H_
```

8.1.4 Summary

- You have created opponent.h.
- You have an idea of the data we collect.

8.2 Collecting Data About Opponents Continued

8.2.1 The opponents.cpp File

Here we discuss the concrete implementation of the classes defined in opponent.h. Put all this code into opponent.cpp in the bt directory. We start with the header and the definition of some constants.

```
/***************************************************************************

    file                 : opponent.cpp
    created              : Thu Apr 22 01:20:19 CET 2003
    copyright            : (C) 2003 Bernhard Wymann

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#include "opponent.h"

/* class variables and constants */
tTrack* Opponent::track;
float Opponent::FRONTCOLLDIST = 200.0;  /* [m] distance to check for other cars */
float Opponent::BACKCOLLDIST = 50.0;    /* [m] distance to check for other cars */
float Opponent::LENGTH_MARGIN = 2.0;    /* [m] safety margin */
float Opponent::SIDE_MARGIN = 1.0;      /* [m] safety margin */
```

To avoid collisions it is not necessary to take into account cars that are farther away than a certain distance. With FRONTCOLLDIST and BACKCOLLDIST you can set the range you want to check for opponents in front of your car and behind your car. The safety margins define the minimal distance you would like to maintain to opponents. If you would like to be able to change these parameters without recompilation, you could also put them into the setup XML file and load them at startup.

```
Opponent::Opponent()
{
}

/* compute speed component parallel to the track */
float Opponent::getSpeed(tCarElt *car)
{
    v2d speed, dir;
    float trackangle = RtTrackSideTgAngleL(&(car->_trkPos));

    speed.x = car->_speed_X;
    speed.y = car->_speed_Y;
    dir.x = cos(trackangle);
    dir.y = sin(trackangle);
    return speed*dir;
}
```

The Opponent::getSpeed(tCarElt *car) method computes the speed of the car in the direction of the track. First we get the direction of the track at the cars location. After that we compose the speed vector of the car. We use _speed_X and _speed_Y here, the uppercase X and Y means that the speed vector is in global coordinates, the lowercase x and y are the speeds in the "instantaneously coincident frame of reference", a coordinate system that is oriented like the cars body but doesn't move with the car (if it would move with the car the _speed_x and _speed_y would always be zero). From the trackangle we compute a direction vector of the track (its length is one). Finally we compute the dot product of the speed and the direction, which will give us the speed in the direction of the track.

```
/* Compute the length to the start of the segment */
```

```
float Opponent::getDistToSegStart()
{
    if (car->_trkPos.seg->type == TR_STR) {
        return car->_trkPos.toStart;
    } else {
        return car->_trkPos.toStart*car->_trkPos.seg->radius;
    }
}
```

This method computes the distance of the opponent to the segments start. The reason is that $car- >_t rkPos$.toStart contains the length of the segment for straight segments and for turns the arc, so we need a conversion to the length.

Now we will have a look at the update function. It is responsible for the update of the values in the Opponent instances.

```
/* Update the values in Opponent this */
void Opponent::update(tSituation *s, Driver *driver)
{
    tCarElt *mycar = driver->getCarPtr();

    /* init state of opponent to ignore */
    state = OPP_IGNORE;

    /* if the car is out of the simulation ignore it */
    if (car->_state & RM_CAR_STATE_NO_SIMU) {
        return;
    }
```

First we get a pointer to the drivers car, set the initial state to ignore the opponent and if the opponent is not longer part of the simulation we return.

```
    /* updating distance along the middle */
    float oppToStart = car->_trkPos.seg->lgfromstart + getDistToSegStart();
    distance = oppToStart - mycar->_distFromStartLine;
    if (distance > track->length/2.0) {
        distance -= track->length;
    } else if (distance < -track->length/2.0) {
        distance += track->length;
    }
```

Now we compute the distance of the center of the drivers car to the center of the opponents car. We achieve that by computing the distances to the start line and taking the difference. If you think about that you will recognize that this is just an approximation of the real distance. If you want a more accurate value you have to compute it with the cars corners (look up car.h). The if part is to ″normalize″ the distance. From our position up to a half track length the opponent is in front of us (distance is positive), else it is behind (distance is negative). A detail is that we can't use _distFromStartLine of the opponent, because it is a private field.

```
/* update speed in track direction */
speed = Opponent::getSpeed(car);
float cosa = speed/sqrt(car->_speed_X*car->_speed_X + car->_speed_Y*car->_speed_Y);
float alpha = acos(cosa);
width = car->_dimension_x*sin(alpha) + car->_dimension_y*cosa;
float SIDECOLLDIST = MIN(car->_dimension_x, mycar->_dimension_x);
```

We update the speed with the previously introduced getSpeed() method. Then we compute the width of the opponents car on the track (think the car is turned 90 degrees on the track, then the needed "width" is its length).

```
/* is opponent in relevant range -50..200 m */
if (distance > -BACKCOLLDIST && distance < FRONTCOLLDIST) {
    /* is opponent in front and slower */
    if (distance > SIDECOLLDIST && speed < driver->getSpeed()) {
        catchdist = driver->getSpeed()*distance/(driver->getSpeed() - speed);
        state |= OPP_FRONT;
        distance -= MAX(car->_dimension_x, mycar->_dimension_x);
        distance -= LENGTH_MARGIN;
        float cardist = car->_trkPos.toMiddle - mycar->_trkPos.toMiddle;
        sidedist = cardist;
        cardist = fabs(cardist) - fabs(width/2.0) - mycar->_dimension_y/2.0;
        if (cardist < SIDE_MARGIN) state |= OPP_COLL;
    } else
```

Here we check if the opponent is in the range we defined as relevant. After that the classification of the opponent starts. In this part we check if it is in front of us and slower. If that is the case we compute the distance we need to drive to catch the opponent (catchdist, we assume the speeds are constant) and set the opponents flag OPP_FRONT. Because the "distance" contains the value of the cars centers we need to subtract a car length and the safety margin. At the end we check if we could collide with to opponent. If yes, we set the flag OPP_COLL.

```
    /* is opponent behind and faster */
    if (distance < -SIDECOLLDIST && speed > driver->getSpeed()) {
        catchdist = driver->getSpeed()*distance/(speed - driver->getSpeed());
        state |= OPP_BACK;
        distance -= MAX(car->_dimension_x, mycar->_dimension_x);
        distance -= LENGTH_MARGIN;
    } else
```

Here we check if the opponent is behind us and faster. We won't use that in the tutorial robot, but you will need it if you want to let overlap faster opponents.

```
    /* is opponent aside */
    if (distance > -SIDECOLLDIST &&
        distance < SIDECOLLDIST) {
            sidedist = car->_trkPos.toMiddle - mycar->_trkPos.toMiddle;
            state |= OPP_SIDE;
        }
    }
}
```

This part is responsible to check if the opponent is aside of us. If yes we compute the distance (sideways) and set the flag OPP_SIDE.

Now the implementation of the Opponents class follows.

```
/* Initialize the list of opponents */
Opponents::Opponents(tSituation *s, Driver *driver)
{
    opponent = new Opponent[s->_ncars - 1];
    int i, j = 0;
    for (i = 0; i < s->_ncars; i++) {
        if (s->cars[i] != driver->getCarPtr()) {
            opponent[j].setCarPtr(s->cars[i]);
            j++;
        }
    }
    Opponent::setTrackPtr(driver->getTrackPtr());
    nopponents = s->_ncars - 1;
}

Opponents::~Opponents()
{
    delete [] opponent;
}
```

The constructor allocates memory and generates the Opponent instances. The destructor deletes the instances and frees the memory.

```
/* Updates all the Opponent instances */
void Opponents::update(tSituation *s, Driver *driver)
{
    int i;
    for (i = 0; i < s->_ncars - 1; i++) {
        opponent[i].update(s, driver);
    }
}
```

The update method simply iterates through the Opponent instances and calls update on them.

8.2.2 Modifying driver.h

To access some data of Driver from Opponent we need to add some methods and variables to driver.h. Put the following methods into the public section.

```
        tCarElt *getCarPtr() { return car; }
        tTrack *getTrackPtr() { return track; }
        float getSpeed() { return speed; }
```

We need also the speed of our car. For that we add a variable to the private section.

```
        float speed;    /* speed in track direction */
```

8.2.3 Compiling opponent.cpp

To check if everything is ok so far you should compile opponent.cpp now. You need to change the Makefile in the bt directory. Change the line

```
SOURCES     = ${ROBOT}.cpp driver.cpp
```

to

```
SOURCES     = ${ROBOT}.cpp driver.cpp opponent.cpp
```

and run "make".

8.2.4 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

8.2.5 Summary

- You have created opponent.cpp.
- You know how it works.

8.3 Front Collision Avoidance

8.3.1 Introduction

In this section we start avoiding collisions. With front collisions I mean the class of collisions where we hit with our front the back of an opponent. We can avoid this collisions with braking, so we will add an additional filter to the brake value. The reason why I handle front and side collisions different is that we need different commands.

8.3.2 Changes in driver.h

We include opponent.h and define prototypes for the classes Opponent and Opponents.

```
#include "opponent.h"
class Opponents;
class Opponent;
```

We add a destructor to the public section, the method prototype for the filter and the variables in the private section.

```
        ~Driver();

        float filterBColl(float brake);

        Opponents *opponents;
        Opponent *opponent;
```

8.3.3 Implementation in driver.cpp

To release the opponents instance we add the destructor.

```
Driver::~Driver()
{
    delete opponents;
}
```

We create the instance of Opponents in Driver::newRace() and get a pointer to the array of Opponent objects.

```
    /* initialize the list of opponents */
    opponents = new Opponents(s, this);
    opponent = opponents->getOpponentPtr();
```

Now call the filter as usual in the Driver::drive() method. Change the line

```
        car->ctrl.brakeCmd = filterABS(getBrake());
```

to

```
        car->ctrl.brakeCmd = filterABS(filterBColl(getBrake()));
```

Here we update the speed of the drivers car in track direction and the opponents data. Put this code at the end of Driver::update().

```
    speed = Opponent::getSpeed(car);
    opponents->update(s, this);
```

Finally we implement the brake filter. In a nutshell it iterates through the opponents and checks for every opponent which has been tagged with OPP_COLL if we need to brake to avoid a collision. If yes we brake full and leave the method.

```
/* Brake filter for collision avoidance */
float Driver::filterBColl(float brake)
{
    float currentspeedsqr = car->_speed_x*car->_speed_x;
    float mu = car->_trkPos.seg->surface->kFriction;
    float cm = mu*G*mass;
    float ca = CA*mu + CW;
    int i;

    for (i = 0; i < opponents->getNOpponents(); i++) {
```

We set up some variables and then we enter the loop to iterate through all opponents. The presented solution is not optimal. You could replace the call to getDistance() with getChatchDist(), but additional code is necessary to make it work. The computation of the brakedistance should be familiar from previous chapters.

```
        if (opponent[i].getState() & OPP_COLL) {
            float allowedspeedsqr = opponent[i].getSpeed();
            allowedspeedsqr *= allowedspeedsqr;
            float brakedist = mass*(currentspeedsqr - allowedspeedsqr) /
                              (2.0*(cm + allowedspeedsqr*ca));
            if (brakedist > opponent[i].getDistance()) {
                return 1.0;
            }
        }
```

```
    }
    return brake;
}
```

8.3.4 Testdrive

To check if the code works run a race with bt 3, bt 1 and bt 2 on wheel 1. The cars should drive behind each other without collisions.

8.3.5 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

8.3.6 Summary

- You can run your robot ”family” without collisions.
- You know how it works.

8.4 Side Collision Avoidance

In this section we add collision avoidance for cars that drive aside of our car. The following implementation will add an additional filter to the steer value. It doesn’t check if we leave the track.

8.4.1 Changes in driver.h

We declare the filter method prototype and the side margin. If the opponent is inside this margin we will try to avoid a collision.

```
        float filterSColl(float steer);

        static const float SIDECOLL_MARGIN;
```

8.4.2 Implementation in driver.cpp

At the beginning of the file we define the new constant.

```
const float Driver::SIDECOLL_MARGIN = 2.0;    /* [m] */
```

In the Driver::drive() method we call the steer filter. Change the line

```
        car->ctrl.steer = getSteer();
```

to

```
        car->ctrl.steer = filterSColl(getSteer());
```

The filterSColl method searches first for the nearest car aside of ours (approximately). If there is another car it checks if it is inside the SIDECOLL_MARGIN. If yes, it computes a new steer value and returns it.

```
/* Steer filter for collision avoidance */
float Driver::filterSColl(float steer)
{
    int i;
    float sidedist = 0.0, fsidedist = 0.0, minsidedist = FLT_MAX;
    Opponent *o = NULL;

    /* get the index of the nearest car (o) */
    for (i = 0; i < opponents->getNOpponents(); i++) {
        if (opponent[i].getState() & OPP_SIDE) {
```

Again we iterate through all cars and check those which have been marked with the OPP_SIDE flag. We init minsidedist with a large value because we search for the minimum.

```
            sidedist = opponent[i].getSideDist();
            fsidedist = fabs(sidedist);
            if (fsidedist < minsidedist) {
                minsidedist = fsidedist;
                o = &opponent[i];
            }
        }
    }
```

Here we check if the current opponent is the nearest one found so far. If that is the case we store a pointer to it and update the minimal distance found (minsidedist).

```
    /* if there is another car handle the situation */
    if (o != NULL) {
        float d = fsidedist - o->getWidth();
        /* near enough */
        if (d < SIDECOLL_MARGIN) {
```

If we found an opponent we check if it is inside the margin. To keep the method simple I make some assumptions, e. g. our car is parallel to the track (that is obiously not true).

```
            /* compute angle between cars */
            tCarElt *ocar = o->getCarPtr();
            float diffangle = ocar->_yaw - car->_yaw;
            NORM_PI_PI(diffangle);
            const float c = SIDECOLL_MARGIN/2.0;
            d = d - c;
            if (d < 0.0) d = 0.0;
            float psteer = diffangle/car->_steerLock;
            return steer*(d/c) + 2.0*psteer*(1.0-d/c);
        }
    }
    return steer;
}
```

Here we compute the angle between the two cars and the weight d. Psteer holds the steer value needed to drive parallel to the opponent (if you draw the scene it is obvious). Finally we weight the original steer and the psteer value and sum them up. This will push away our car from the opponent when we are near. You could improve this function with more accurate distance values and checking if the normal steer value points already more away from the opponent than the new computed value.

8.4.3 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

8.4.4 Summary

- You have implemented side collision avoidance.
- You know how it works.
- You know how to improve it.

8.5 Overtaking

To be able to overtake we need to modify again the steer value. This time we achieve that with the modification of the target point. Depending on the side we want to pass an opponent we slightly add an offset vector to the target point. This offset is perpendicular to the track tangent. The opponent we choose for overtaking is the one with the smallest catchdistance, because we will reach it first.

8.5.1 Changes in driver.h

Add the following method, variable and constant declarations to the private section.

```
        float getOvertakeOffset();

        float myoffset;  /* overtake offset sideways */

        static const float BORDER_OVERTAKE_MARGIN;
        static const float OVERTAKE_OFFSET_INC;
\begin{lstlisting}



\subsection{Implementation in driver.cpp}

  At the beginning of the file we define the new constants. You should also
  change the WIDTHDIV constant to 3.0.

\begin{lstlisting}
const float Driver::BORDER_OVERTAKE_MARGIN = 0.5; /* [m] */
const float Driver::OVERTAKE_OFFSET_INC = 0.1;    /* [m/timestep] */

const float Driver::WIDTHDIV = 3.0;               /* [-] */
```

The BORDER_OVERTAKE_MARGIN is the border relative to the filterTrk width. It should avoid that we leave the range on the track where filterTrk allows acceleration. OVERTAKE_OFFSET_INC is the increment of the offset value. In Driver::newRace() we initialize myoffset to zero.

```
    myoffset = 0.0;
```

Now we have a look on how we compute the offset of the target point.

```
/* Compute an offset to the target point */
float Driver::getOvertakeOffset()
{
    int i;
    float catchdist, mincatchdist = FLT_MAX;
    Opponent *o = NULL;

    for (i = 0; i < opponents->getNOpponents(); i++) {
        if (opponent[i].getState() & OPP_FRONT) {
            catchdist = opponent[i].getCatchDist();
            if (catchdist < mincatchdist) {
                mincatchdist = catchdist;
                o = &opponent[i];
            }
        }
    }
```

First we find the opponent with the smallest catchdist. Remember, that is the car which we will reach first. Of course the candidates need to have OPP_FRONT set.

```
    if (o != NULL) {
        float w = o->getCarPtr()->_trkPos.seg->width/WIDTHDIV-BORDER_OVERTAKE_MARGIN;
        float otm = o->getCarPtr()->_trkPos.toMiddle;
        if (otm > 0.0 && myoffset > -w) myoffset -= OVERTAKE_OFFSET_INC;
        else if (otm < 0.0 && myoffset < w) myoffset += OVERTAKE_OFFSET_INC;
    } else {
```

In case there is an opponent to overtake we let slightly grow the offset toward the other track side to pass it. The borders are checked so that the offset stays within "w".

```
        if (myoffset > OVERTAKE_OFFSET_INC) myoffset -= OVERTAKE_OFFSET_INC;
        else if (myoffset < -OVERTAKE_OFFSET_INC) myoffset += OVERTAKE_OFFSET_INC;
        else myoffset = 0.0;
    }
    return myoffset;
}
```

In case we have not found an opponent the offset goes back slightly toward zero. In the next section we will add the offset to the target point.

8.5.2 Summary

- You have implemented the code above.
- You know how it works.

8.6 Finishing Overtaking

Finally we need to add the offset to the target point. We will integrate that directly into the getTargetPoint() method. For that we need a normalized vector at the target point perpendicular to the track tangent. Before we add this vector we multiply it with the offset.

8.6.1 Implementation

Have a look on the changed Driver::getTargetPoint() method.

```
/* compute target point for steering */
v2d Driver::getTargetPoint()
{
    tTrackSeg *seg = car->_trkPos.seg;
    float lookahead = LOOKAHEAD_CONST + car->_speed_x*LOOKAHEAD_FACTOR;
    float length = getDistToSegEnd();
    float offset = getOvertakeOffset();
```

The first change is the offset variable, initialized with getOvertakeOffset().

```
    while (length < lookahead) {
        seg = seg->next;
        length += seg->length;
    }

    length = lookahead - length + seg->length;
    v2d s;
    s.x = (seg->vertex[TR_SL].x + seg->vertex[TR_SR].x)/2.0;
    s.y = (seg->vertex[TR_SL].y + seg->vertex[TR_SR].y)/2.0;

    if ( seg->type == TR_STR) {
        v2d d, n;
        n.x = (seg->vertex[TR_EL].x - seg->vertex[TR_ER].x)/seg->length;
        n.y = (seg->vertex[TR_EL].y - seg->vertex[TR_ER].y)/seg->length;
        n.normalize();
```

Here the new v2d variable "n" has been added. It is initialized with the normalized vector perpendicular to the track tanget.

```
        d.x = (seg->vertex[TR_EL].x - seg->vertex[TR_SL].x)/seg->length;
        d.y = (seg->vertex[TR_EL].y - seg->vertex[TR_SL].y)/seg->length;
        return s + d*length + offset*n;
    } else {
```

Finally "n" is multiplied with offset and added to the target point. A similar implementation follows for the turns.

```
        v2d c, n;
        c.x = seg->center.x;
        c.y = seg->center.y;
        float arc = length/seg->radius;
        float arcsign = (seg->type == TR_RGT) ? -1.0 : 1.0;
        arc = arc*arcsign;
        s = s.rotate(c, arc);
        n = c - s;
        n.normalize();
        return s + arcsign*offset*n;
    }
}
```

8.6.2 Testdrive

Before you go to the track you should set the camber of the front wheels to -5 degrees. Change this in the file 0/default.xml. My conclusion after some races is that the collision avoidance and overtaking works really well (at least for its simplicity).

8.6.3 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

8.6.4 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

8.6.5 Summary

- You have implemented and understood collision avoidance and overtaking.
- You have a robot which is able to race.
- You know how to improve it.

Chapter 9

Pit Stops

9.1 Pit Stops

In this chapter we enable our driver to perform pit stops. First let us discuss what they are good for and which steps we need to stop in the pit. You already know that your car can get damage during the race and consumes fuel. To reach the finish line in longer races we need to be able to repair and refuel our car (tire wearing and changing is currently not implemented in TORCS). Damage is modelled as damage "points", if you reach 10000 you are not able to drive further. A pit stop consists of the following steps:

- Decide that we want to stop in the pit (strategy).
- Drive into the pit lane with respect to the pit speed limit.
- Drive to our pit and stop here, request the pit stop in TORCS.
- When the car is slow enough and in the pit the simulation "captures" our car and asks us for the damage to repair (in damage "points") and how much fuel we need.
- TORCS repairs and refuels our car.
- The car becomes released, we drive to the pit exit with respect to the speed limit.

- Back to business, race further.

You might have recognized that we do not ask if our pit is free, that is because TORCS provides currently one pit per car (not per team). From the above list we can derive the items we need to implement:

- A path to the pit and back to the track. We will use a spline to compute an offset to the middle of the track (do you remember the overtake offset?).
- A strategy part which decides if we need a pit stop.
- The callback which tells TORCS how much repair and fuel we need.
- An additional filter for the brake value to stop in the pit.
- Some helper methods.

9.2 Summary

- You know what pit stops are good for.
- You know about repairing, damage "points" and refueling.
- You have an idea what we need to implement.

9.3 Splines

9.3.1 Introduction

In the last section you have seen that we need to find somehow a path into our pit. I decided to show you an implementation which uses "cubic" splines to create the path. Cubic splines are interpolating polynomials between given base points, such that the function itself and the first derivative is smooth. We need a smooth path because our car can not follow a path with "cracks". First I will show you the implementation of the spline algorithm, after that we will prepare the base points.

9.3.2 The Spline Implementation

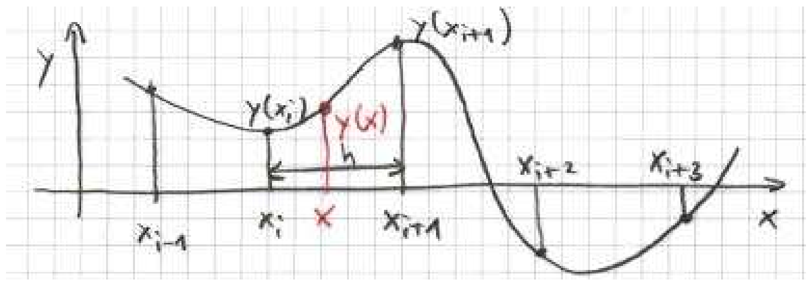


Figure 9.1: spline picture

If you want to know how the following algorithm works, you can look it up here(Appendix A). In a nutshell, you query for a given parameter ("x") the value of the spline function ("y"). The algorithm searches first the interval in which "x" falls, then it applies the interpolation to compute y(x). Now let's have a look on the spline.h file (create it in the bt directory).

```
/***************************************************************************

    file                 : spline.h
    created              : Wed Mai 14 19:53:00 CET 2003
    copyright            : (C) 2003 by Bernhard Wymann
    email                : berniw@bluewin.ch

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef _SPLINE_H_
#define _SPLINE_H_

class SplinePoint {
    public:
        float x;    /* x coordinate */
        float y;    /* y coordinate */
        float s;    /* slope */
};
```

The SplinePoint class holds the data for each base point of the spline. For the interpolation algorithm we need to define also the slope at (x,y).

```
class Spline {
    public:
        Spline(int dim, SplinePoint *s);

        float evaluate(float z);

    private:
        SplinePoint *s;
        int dim;
};

#endif // _SPLINE_H_
```

The Spline class provides just two methods. The constructor to initialize the number of base points (dim) and a pointer to an array of SplinePoints. The second method finally computes the spline values. Here follows the implementation, put it into spline.cpp.

```
/***************************************************************************

    file                 : spline.cpp
    created              : Wed Mai 14 20:10:00 CET 2003
    copyright            : (C) 2003 by Bernhard Wymann
    email                : berniw@bluewin.ch

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/


#include "spline.h"


Spline::Spline(int dim, SplinePoint *s)
{
    this->s = s;
    this->dim = dim;
}


float Spline::evaluate(float z)
{
    int i, a, b;
    float t, a0, a1, a2, a3, h;

    a = 0; b = dim-1;
```

First the algorithm searches for the interval in which the parameter z falls.

```
    do {
        i = (a + b) / 2;
        if (s[i].x <= z) a = i; else b = i;
    } while ((a + 1) != b);
```

Now we know the interval and do the interpolation. For that we compute first the parameters of the polynomial, then we evaluate it.

```
    i = a; h = s[i+1].x - s[i].x; t = (z-s[i].x) / h;
    a0 = s[i].y; a1 = s[i+1].y - a0; a2 = a1 - h*s[i].s;
    a3 = h * s[i+1].s - a1; a3 -= a2;
    return a0 + (a1 + (a2 + a3*t) * (t-1))*t;
}
```

Finally you need to add spline.cpp to the Makefile. Change

```
SOURCES      = ${ROBOT}.cpp driver.cpp opponent.cpp
```

to

```
SOURCES      = ${ROBOT}.cpp driver.cpp opponent.cpp spline.cpp
```

9.3.3 The Pit Path

The following picture shows the path to and out of the pit. Like you can see we need the base points p0 to p6 to be able to compute the path. Our x-axis is the middle line along the track, so the slope in the base points is zero (parallel to the track).

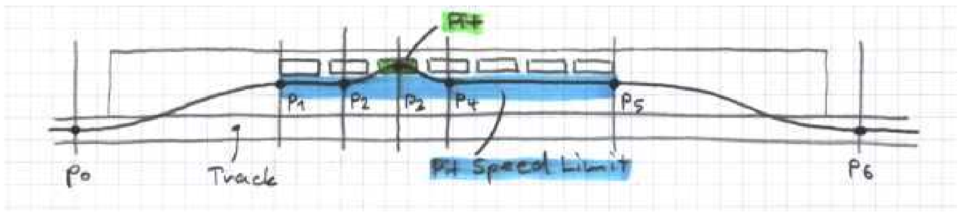


Figure 9.2: path to the pit

We compute the base points of the spline in the constructor of a new class, put it into a new file and save it as pit.cpp. That we don't need to change the constructor all the time I will present you the final result. We will discuss all the new variables later in detail, at the moment we just focus on the base points. The header file will follow later.

```
/***************************************************************************
```

```
    file                 : pit.cpp
    created              : Thu Mai 15 2:43:00 CET 2003
    copyright            : (C) 2003 by Bernhard Wymann
    email                : berniw@bluewin.ch

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/


#include "pit.h"

const float Pit::SPEED_LIMIT_MARGIN = 0.5;      /* [m/s] savety margin */
const int Pit::PIT_DAMMAGE = 5000;              /* [-] */
```

The speed limit margin defines how close our speed limiter should go to the allowed speed. The pit damage defines the threshold when we decide to go into the pit to repair our car.

```
Pit::Pit(tSituation *s, Driver *driver)
{
    track = driver->getTrackPtr();
    car = driver->getCarPtr();
    mypit = driver->getCarPtr()->_pit;
    pitinfo = &track->pits;
    pitstop = inpitlane = false;
    fuelchecked = false;
    fuelperlap = 0.0;
    lastpitfuel = 0.0;
    lastfuel = car->priv.fuel;

    if (mypit != NULL) {
```

Set up several variables for strategy and pit stops. Of interest is the mypit variable, if it is not null we have a pit (there are tracks without pits or more drivers in the race than pits available), and the pitinfo variable. In mypit you can find specific information about your own pit, pitinfo holds data about the pit lane. If we do not own a pit we cannot plan a path.

```
        speedlimit = pitinfo->speedLimit - SPEED_LIMIT_MARGIN;
        speedlimitsqr = speedlimit*speedlimit;
```

First the speed limit and the square of it are computed.

```
        /* compute pit spline points along the track */
        p[3].x = mypit->pos.seg->lgfromstart + mypit->pos.toStart;
```

The position of our pit along the track relative to the start is the position of the pit segment start plus the relative position of our pit in the pit segment.

```
        p[2].x = p[3].x - pitinfo->len;
```

A reasonable point to go from the pit lane into our pit is the middle of the previous pit. If you experiment with this value you have to make sure that you do not collide with other cars in their pit.

```
        p[4].x = p[3].x + pitinfo->len;
```

The same applies for the way back to the pit lane.

```
        p[0].x = pitinfo->pitEntry->lgfromstart;
```

The entry point where we start leaving the track and heading to the pit lane. This value is on much tracks not reasonable, so you want to make it later adjustable in the xml file or compute it with a heuristic.

```
        p[1].x = pitinfo->pitStart->lgfromstart;
```

The start of the pit lane, from here we have to respect the speed limit.

```
        p[5].x = p[3].x + (pitinfo->nMaxPits - car->index)*pitinfo->len;
```

The end of the pit lane. It is computed this way, because on some tracks the end is not reported correct.

```
        p[6].x = pitinfo->pitExit->lgfromstart;
```

And finally the point where we are back on the track. Now we have computed all "x" values of our spline base points.

```
        pitentry = p[0].x;
        pitexit = p[6].x;

        /* normalizing spline segments to >= 0.0 */
        int i;
        for (i = 0; i < NPOINTS; i++) {
            p[i].s = 0.0;
            p[i].x = toSplineCoord(p[i].x);
        }
```

The above loop initializes all the slopes to zero and converts the "x" values into a range from zero to the distance of p[6] and p[0]. The conversion is necessary because the start line can be within the pit lane (the "x" value grows till the start line, then it drops to zero and grows again). To get proper interpolation we need a continuous “x”.

```
        if (p[1].x > p[2].x) p[1].x = p[2].x;
        if (p[4].x > p[5].x) p[5].x = p[4].x;
```

If we have the first or the last pit we need to adjust the ordering of the points.

```
        float sign = (pitinfo->side == TR_LFT) ? 1.0 : -1.0;
        p[0].y = 0.0;
        p[6].y = 0.0;
        for (i = 1; i < NPOINTS - 1; i++) {
            p[i].y = fabs(pitinfo->driversPits->pos.toMiddle) - pitinfo->width;
            p[i].y *= sign;
        }
```

The above loop initializes the “y” values of the spline. You can also experiment with other values.

```
        p[3].y = fabs(pitinfo->driversPits->pos.toMiddle)*sign;
        spline = new Spline(NPOINTS, p);
    }
}
```

Finally we set the “y” value of our pit and create a spline object.

```
Pit::~Pit()
{
    if (mypit != NULL) delete spline;
}
```

If we allocated a spline we have to delete it in our destructor.

9.3.4 Summary

- You know the way into the pit.
- You know how to use the spline class.

9.4 The Pit Class

9.4.1 The Header File

In this section we prepare the header file for the pit class. It contains the definition of the functionality to decide if we want to pit (strategy), to compute the pit path (offset) and some utility functions. Put the following code into a new file named pit.h.

```
/***************************************************************************

    file                 : pit.h
    created              : Thu Mai 15 2:41:00 CET 2003
    copyright            : (C) 2003 by Bernhard Wymann
    email                : berniw@bluewin.ch

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef _PIT_H_
#define _PIT_H_

#include "driver.h"
#include "spline.h"

#define NPOINTS 7
```

First we define the number of points for the pit path spline.

```
class Driver;

class Pit {
    public:
        Pit(tSituation *s, Driver *driver);
        ~Pit();
```

The constructor and destructor. We have already implemented them in the last section.

```
        void setPitstop(bool pitstop);
        bool getPitstop() { return pitstop; }
```

Setter and getter for the pitstop variable.

```
        void setInPit(bool inpitlane) { this->inpitlane = inpitlane; }
        bool getInPit() { return inpitlane; }
```

Setter and getter for the inpitlane variable.

```
        float getPitOffset(float offset, float fromstart);
```

Computes for a given distance from the start line the offset of the path.

```
        bool isBetween(float fromstart);
```

Checks if the given distance from start is between pitentry and pitexit. Such a method is handy because the value wraps-around at the start line.

```
        float getNPitStart() { return p[1].x; }
        float getNPitLoc() { return p[3].x; }
        float getNPitEnd() { return p[5].x; }
```

Methods to get the spline “x” coordinates of the points 1,3 and 5. We need them to control the speed limit in the pit lane and to stop in the pit.

```
        float toSplineCoord(float x);
```

Converts the track (distance from start) “x” coordinates of the spline into spline “x” coordinates, starting from zero with no wrap-around in the “x” value.

```
        float getSpeedlimitSqr() { return speedlimitsqr; }
        float getSpeedlimit() { return speedlimit; }
```

Methods to get the speed limit in the pits. The returned value already includes the security margin.

```
        void update();
        int getRepair();
        float getFuel();
```

The update() method does the housekeeping of the data used for the strategy and decides if we need to pit. The getRepair() and getFuel() methods decides how much damage gets repaired and how much fuel we get at the pit stop.

```
    private:
        tTrack *track;
        tCarElt *car;
        tTrackOwnPit *mypit;     /* pointer to my pit */
        tTrackPitInfo *pitinfo; /* general pit info */
```

Pointers to the track, car, the pit and the pitinfo structure. I use them to avoid too much indirection.

```
        SplinePoint p[NPOINTS]; /* spline points */
        Spline *spline;         /* spline */
```

The spline point data and the spline instance.

```
        bool pitstop;           /* pitstop planned */
```

The pitstop variable indicates if we plan a pit stop. It will become set to false right after the stop, because we do not need to stop anymore.

```
        bool inpitlane;         /* we are still in the pit lane */
```

The inpitlane variable indicates if we are in the pit lane. It will become set to true if pitstop is true and we are in the pit lane (pitentry). It will become set to false if we passed pitexit. The variable is necessary to finish the pit stop correctly.

```
        float pitentry;         /* distance to start line of the pit entry */
        float pitexit;          /* distance to the start line of the pit exit */
        float speedlimitsqr;    /* pit speed limit squared */
        float speedlimit;       /* pit speed limit */
```

Some static data needed by several functions.

```
        bool fuelchecked;       /* fuel statistics updated */
        float lastfuel;         /* the fuel available when we cross the start lane */
        float lastpitfuel;      /* amount refueled, special case when we refuel */
        float fuelperlap;       /* the maximum amount of fuel we needed for a lap */
```

The data necessary to decide if we need to refuel or not. The fuelperlap variable contains the maximum amount of fuel we needed so far for a lap. The amount of fuel we request on the pit stop will be computed based on that value.

```
        static const float SPEED_LIMIT_MARGIN;
        static const int PIT_DAMMAGE;
};

#endif // _PIT_H_
```

Finally the declaration of the constants. You already know them from the last section.

9.4.2 Summary

- You have implemented the header file pit.h.

9.5 Pit Utility Functions

9.5.1 Introduction

In this section I show you the utility functions of the pit class. The functions responsible for the strategy will follow in the next section. Put the following code into pit.cpp.

9.5.2 The toSplineCoord Method

This method converts distances from the start line into distances from the pit entry.

```
/* Transforms track coordinates to spline parameter coordinates */
float Pit::toSplineCoord(float x)
{
    x -= pitentry;
    while (x < 0.0) {
        x += track->length;
    }
    return x;
}
```

First we subtract of the given value the distance of the pit entry from the start. Because the start line can lie between the pit entry and the pit exit, it is possible that the result becomes negative. In such a case we have to add a track length to correct that.

9.5.3 The getPitOffset Method

This method computes the pit offset for the path into the pit which is added to the target point. The arguments given are a current offset and the distance of the target point from the start.

```
/* computes offset to track middle for trajectory */
float Pit::getPitOffset(float offset, float fromstart)
{
    if (mypit != NULL) {
        if (getInPit() || (getPitstop() && isBetween(fromstart))) {
            fromstart = toSplineCoord(fromstart);
            return spline->evaluate(fromstart);
        }
    }
    return offset;
}
```

First we check if there is a pit available. If we have a pit we check if we are already performing a pit stop with getInPit() or if we plan a pit stop and the target point falls onto the pit path. If the expression evaluates to true we want to follow the pit path, therefore we compute a new offset and return it. If one of the above expressions fails, we simply return the given offset.

9.5.4 The setPitstop Method

This is the setter method for the pitstop variable. You should just access the pitstop variable through this method, because it makes sure that the variable is not set while we are on the track parallel to the pit lane. It is not allowed to set the variable to true in this range because we would start immediately to head to the pit path, which would lead to a crack in the path and probably cause an accident. Setting to false is allowed, because the inpit variable will indicate that we have to finish the pit stop.

```
/* Sets the pitstop flag if we are not in the pit range */
void Pit::setPitstop(bool pitstop)
{
    if (mypit == NULL) return;
    float fromstart = car->_distFromStartLine;

    if (!isBetween(fromstart)) {
        this->pitstop = pitstop;
    } else if (!pitstop) {
        this->pitstop = pitstop;
    }
}
```

If we do not own a pit return. If we have a pit do the above explained checks and leave the pitstop variable in the correct state.

9.5.5 The isBetween Method

Checks if the given argument (a distance from the startline) falls between the pit entry and the pit exit.

```
/* Check if the argument fromstart is in the range of the pit */
bool Pit::isBetween(float fromstart)
{
    if (pitentry <= pitexit) {
        if (fromstart >= pitentry && fromstart <= pitexit) {
            return true;
        } else {
            return false;
        }
    } else {
        if ((fromstart >= 0.0 && fromstart <= pitexit) ||
            (fromstart >= pitentry && fromstart <= track->length))
        {
            return true;
        } else {
            return false;
        }
    }
}
```

First we check if pitentry is smaller than pitexit. If that is the case the start line falls not into the pit range. So we can simply check if the given argument is greater than pitentry and smaller than pitexit.

If pitentry is greater than pitexit, then the start line is in the pit range. The given argument falls in this case in the pit range if it is between zero and pitexit or between pitentry and the track end (equals the length).

9.5.6 Summary

- You have understood and implemented the above methods.

9.6 Pit Strategy Functions

9.6.1 Introduction

In this section we will finish the pit class. What is still missing is the strategy functionality. I will show you a very simple implementation, which will work quite well for endurance races, but not for short ones. It works exactly the same way like in the berniw robot.

9.6.2 The update Method

The update method is responsible for the housekeeping of some variables and to decide if we need a pit stop.

```
/* update pit data and strategy */
void Pit::update()
{
    if (mypit != NULL) {
        if (isBetween(car->_distFromStartLine)) {
            if (getPitstop()) {
                setInPit(true);
            }
        } else {
            setInPit(false);
        }
```

If we are between pitentry and pitexit and pitstop is set, set inpit to true. We need that to remember after the pit stop that we have to follow the pit path to leave the pit lane. That is necessary because the pitstop variable is set to false immediately after the pit stop, when we still need to drive out of the pit lane and back to the track. If we are not between pitentry and pitexit we set inpit to false.

```
        /* check for damage */
        if (car->_dammage > PIT_DAMMAGE) {
            setPitstop(true);
        }
```

The first very simple strategy rule. If the damage of the car is greater than PIT_DAMMAGE, we set pitstop to true. Here is very much room for improvement, you could make that dependent on the remaining laps and the situation in the race.

```
        /* fuel update */
        int id = car->_trkPos.seg->id;
        if (id >= 0 && id < 5 && !fuelchecked) {
```

We update the fuel values once per lap when we cross the track segment zero. We check the range of 5 segments that we catch it for sure.

```
            if (car->race.laps > 0) {
                fuelperlap = MAX(fuelperlap, (lastfuel+lastpitfuel-car->priv.fuel));
            }
            lastfuel = car->priv.fuel;
            lastpitfuel = 0.0;
            fuelchecked = true;
        } else if (id > 5) {
            fuelchecked = false;
        }
```

We compute the amount of fuel which the car has consumed till the check one lap before. The amount of the consumed fuel on the last lap is usually the difference between the available amount of fuel one lap ago (lastfuel) and the current amount of fuel available (car-¿priv.fuel). If we have performed a pit stop, this is not true. To make the computation work we have to add to the lastfuel value the amount of new fuel (lastpitfuel).

We take the maximum because if we get some damage or have to brake and accelarate more than usual the car consumes more fuel. If you take the average you may run out of fuel.

```
        int laps = car->_remainingLaps-car->_lapsBehindLeader;
        if (!getPitstop() && laps > 0) {
            if (car->_fuel < 1.5*fuelperlap &&
                car->_fuel < laps*fuelperlap) {
                setPitstop(true);
            }
        }
```

Here comes the second strategy rule, feel also free to improve that. If there are some laps remaining for our robot check if we have fuel for the next one and a half laps and if we need to refuel at all. If that is the case set pitstop to true.

```
        if (getPitstop()) car->_raceCmd = RM_CMD_PIT_ASKED;
    }
}
```

If pitstop is true set the race command accordingly (here we let TORCS know that we want to pit). Our car is captured by the pit if the car is slow enough, near enough to the center of our pit and if the race command is set to RM_CMD_PIT_ASKED.

9.6.3 The getFuel Method

This method computes the amount of fuel we request on the pit stop. It is called from the drivers Driver::pitCommand callback function (I will show you that soon). It is also part of the strategy.

```
/* Computes the amount of fuel */
float Pit::getFuel()
{
    float fuel;
    fuel = MAX(MIN((car->_remainingLaps+1.0)*fuelperlap - car->_fuel,
                   car->_tank - car->_fuel),
               0.0);
    lastpitfuel = fuel;
    return fuel;
}
```

The fuel we need to finish the race is the difference between the remaining laps times the fuel we need per lap and the fuel in the tank. To play safe we request fuel for an additional lap and take the remaining laps of our car (perhaps the leader will have an accident). If the amount of fuel we need is bigger than the tank, we cut the amount with the second expression in the MIN statement. Because we perhaps stopped in the pit to repair damage, the needed amount of fuel can become negative. For that is the surrounding MAX statement.

9.6.4 The getRepair Method

Computes the damage points to repair. A the moment we simply repair the whole damage. This is on short races really braindead... so improve that.

```
/* Computes how much damage to repair */
int Pit::getRepair()
{
    return car->_dammage;
}
```

9.6.5 The Makefile

We have now finished pit.cpp. To check if everything is in place, we add it in the Makefile. Of course it will still do nothing, because we do not instantiate the pit in our driver yet. This will be the last step. Now change in the Makefile the line

```
SOURCES     = ${ROBOT}.cpp driver.cpp opponent.cpp spline.cpp
```

to

```
SOURCES       = ${ROBOT}.cpp driver.cpp opponent.cpp spline.cpp pit.cpp
```

9.6.6 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

9.6.7 Summary

- You have understood and implemented the above methods.
- You know that you can improve the strategy a lot.

9.7 Pit Brake Filter

9.7.1 Introduction

Remember first what we already have implemented to stop in the pit. We have a path into the pit and we have a strategy to decide if we want to stop in the pit. That is fine so far if we just want to drive through the pit, what is missing is a part for stopping in the pit and maintaining the speed limit. In this section you implement the brake filter which is responsible for those tasks.

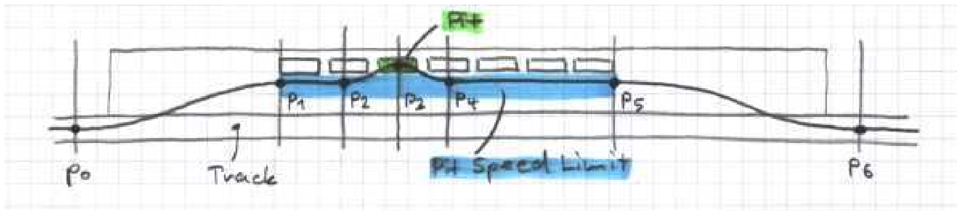


Figure 9.3: path to the pit

9.7.2 Changes in the Header File

Add the following declarations into the header file driver.h.

```
        float brakedist(float allowedspeed, float mu);
```

Because we will use the calculation of the braking distance from the pit brake filter and other places, we encapsulate the formula finally into a method (at the moment we have it hardwired in the getBrake and filterB-Coll methods).

```
        float filterBPit(float brake);
```

The pit brake filter method.

```
        static const float PIT_LOOKAHEAD;
```

We need another (much shorter) lookahead value for the pit, because we have to make a very tough turn from the pit lane into the pit.

```
        static const float PIT_BRAKE_AHEAD;
        static const float PIT_MU;
```

These constants are used as workaround for two problems. The pit has on much tracks another surface than the track itself, so it would be necessary to analyse the friction to brake early enough. Because we do not take care (of course you will implement that, don't you?) we simply multiply on all tracks the friction with PIT_MU for the pit stop.

The PIT_BRAKE_AHEAD is used to workaround the problem of pitentry points which are very close to the first pit. On such tracks we will simply brake too late to get the turn into the pit right, so we have to brake earlier. If you have added better pitentry points (via XML or computed with a heuristic) you can remove that.

9.7.3 The Implementation

Now let's have a look on the implementation. Apply all changes to the file driver.cpp.

```
const float Driver::PIT_LOOKAHEAD = 6.0;        /* [m] */
const float Driver::PIT_BRAKE_AHEAD = 200.0;    /* [m] */
const float Driver::PIT_MU = 0.4;               /* [-] */
```

First define the constants at the beginning of the file.

```
float Driver::brakedist(float allowedspeed, float mu)
{
    float allowedspeedsqr = allowedspeed*allowedspeed;
    float cm = mu*G*mass;
    float ca = CA*mu + CW;
    return mass*(currentspeedsqr - allowedspeedsqr) / (2.0*(cm + allowedspeedsqr*ca));
}
```

Here we encapsulate the brake distance formula into its own method (remember chapter 3). Now everything is in place and we can start with the implementation of the brake filter.

```
float Driver::filterBPit(float brake)
{
    if (pit->getPitstop() && !pit->getInPit()) {
        float dl, dw;
        RtDistToPit(car, track, &dl, &dw);
        if (dl < PIT_BRAKE_AHEAD) {
            float mu = car->_trkPos.seg->surface->kFriction*PIT_MU;
            if (brakedist(0.0, mu) > dl) return 1.0;
        }
    }
```

The above code is responsible for braking early enough to make the turn into the pit lane. Like already mentioned in the discusson of PIT_BRAKE_AHEAD you can remove that code if you have better pitentry values. Now lets have a look on the nice part.

```
    if (pit->getInPit()) {
        float s = pit->toSplineCoord(car->_distFromStartLine);
```

If we are on the pit trajectory, convert the position of our car into spline coordinates.

```
        /* pit entry */
        if (pit->getPitstop()) {
```

If pitstop is true the pit is in front of us.

```
            float mu = car->_trkPos.seg->surface->kFriction*PIT_MU;
            if (s < pit->getNPitStart()) {
```

If we are not yet in the range of the speed limit we have to check if we need to brake to the speed limit.

```
                /* brake to pit speed limit */
                float dist = pit->getNPitStart() - s;
                if (brakedist(pit->getSpeedlimit(), mu) > dist) return 1.0;
```

First compute the distance to the speed limit zone (p1). Then check if we need already to brake.

```
            } else {
                /* hold speed limit */
                if (currentspeedsqr > pit->getSpeedlimitSqr()) return 1.0;
            }
```

If we take the else we are already in the speed limit zone and have to maintain the speed limit. It is (like always in this tutorial) a very simple implementation.

```
            /* brake into pit (speed limit 0.0 to stop ) */
            float dist = pit->getNPitLoc() - s;
            if (brakedist(0.0, mu) > dist) return 1.0;
```

If we did not brake till now we have to check if we need to brake to stop in the pit. We compute the distance to the pit and check the brake distance.

```
            /* hold in the pit */
            if (s > pit->getNPitLoc()) return 1.0;
```

Finally if we passed the pit center and still want to pit we have to brake full, else we will miss the pit.

```
        } else {
            /* pit exit */
            if (s < pit->getNPitEnd()) {
                /* pit speed limit */
                if (currentspeedsqr > pit->getSpeedlimitSqr()) return 1.0;
            }
        }
    }
```

In this code section the pitstop variable is false, so we have just to check if we are in the speed limit zone and to maintain the speed limit in it.

```
    return brake;
}
```

If none of the above expressions returned a brake value, we return the given value.

9.7.4 Conclusion

You have implemented a simple brake filter to stop in the pit. If you analyse the code, you will discover some unpleasant things. Assume for example that an opponent hits us before we stop in the pit and pushes our car over the pit center. Our code will then apply full brakes, but if we got pushed too far away from the pit center TORCS will not capture our car. That means we stay there with full applied brakes forever. I did not fix that because it happens not often, my berniw robot has the same problem. You can solve that with kind of a timeout mechanism or distance measurement to the pit.

I leave it up to you to replace the brake distance code from getBrake and filterBColl through the new brakedist call. In the next section we will put all the stuff together. If you want to celebrate the pit stops with a cold juice it is now time to put it into the fridge.

9.7.5 Summary

- You have implemented and understood the brake filter.

9.8 Put it All Together

9.8.1 Introduction

In this section we put all components we have built together. We have to instantiate the pit class, integrate the pit callback and to compute the target point with taking into account the pit offset.

9.8.2 Changes in driver.h

We include pit.h and add a prototype of the pit class.

```
#include "pit.h"

class Pit;
```

To remember our pit instance we need a variable. We also compute and remember the current speed square, because we need it in several places.

```
        Pit *pit;
        float currentspeedsqr;
```

9.8.3 Creation and Destruction of the Pit Object

Add the release of our pit object into the destructor Driver:: Driver().

```
    delete pit;
```

Add the creation of the pit object into Driver::newRace(tCarElt* car, tSituation *s).

```
    /* create the pit object */
    pit = new Pit(s, this);
```

9.8.4 Changes in Driver::drive(tSituation *s)

Swap the following two statements, because the initialization to zero will overwrite the result of the pit update if it sets the pit race command. Change in Driver::drive(tSituation *s)

```
    update(s);
    memset(&car->ctrl, 0, sizeof(tCarCtrl));
```

to

```
    memset(&car->ctrl, 0, sizeof(tCarCtrl));
    update(s);
```

Change the brake value computation such that it takes into account the pit brake filter. Change

```
        car->ctrl.brakeCmd = filterABS(filterBColl(getBrake()));
```

to

```
        car->ctrl.brakeCmd = filterABS(filterBColl(filterBPit(getBrake())));
```

9.8.5 The Pit Callback

Now we integrate the pit getRepair and getFuel methods into the pit callback. We set pitstop to false. Now it should be finally clear how a pit stop works. Our car stops in the pit and if it is slow and near enough to the pit middle and we request a pit stop, TORCS calls this callback function (remember chapter 2 where you registered it). We tell TORCS our wishes and set pitstop to false. TORCS holds our car captured during the repair and refueling time, so we can't drive away. After TORCS releases our car the drive function is called as usual and we leave the pit. Replace

```
/* Set pitstop commands. */
int Driver::pitCommand(tSituation *s)
{
    return ROB_PIT_IM; /* return immediately */
}
```

with

```
/* Set pitstop commands */
int Driver::pitCommand(tSituation *s)
{
    car->_pitRepair = pit->getRepair();
    car->_pitFuel = pit->getFuel();
    pit->setPitstop(false);
    return ROB_PIT_IM; /* return immediately */
}
```

9.8.6 Change the Target Point Computation

Now we have to modify the getTargetPoint method. That you know where to insert the code the unchanged code is in grey.

```
/* compute target point for steering */
v2d Driver::getTargetPoint()
{
    tTrackSeg *seg = car->_trkPos.seg;
    float lookahead = LOOKAHEAD_CONST + car->_speed_x*LOOKAHEAD_FACTOR;
    float length = getDistToSegEnd();
    float offset = getOvertakeOffset();
```

If we are on the pit trajectory we switch to another computation of the lookahead value. If we are in the speed limit range we even take a shorter lookahead value. You could implement this also with one good formula.

```
    if (pit->getInPit()) {
        if (currentspeedsqr > pit->getSpeedlimitSqr()) {
            lookahead = PIT_LOOKAHEAD + car->_speed_x*LOOKAHEAD_FACTOR;
        } else {
            lookahead = PIT_LOOKAHEAD;
        }
    }

    while (length < lookahead) {
        seg = seg->next;
        length += seg->length;
    }
    length = lookahead - length + seg->length;
```

We have to pass the distance to the start line of the target point to getPitOffset, so we compute that first. After that we call the computation of the pit offset.

```
    float fromstart = seg->lgfromstart;
    fromstart += length;
    offset = pit->getPitOffset(offset, fromstart);

    v2d s;
    s.x = (seg->vertex[TR_SL].x + seg->vertex[TR_SR].x)/2.0;
    s.y = (seg->vertex[TR_SL].y + seg->vertex[TR_SR].y)/2.0;
```

9.8.7 Change the Update

The second last thing left is to update currentspeedsqr and to call the pit update method. Put this code at the end of Driver::update(tSituation *s).

```
    currentspeedsqr = car->_speed_x*car->_speed_x;
    pit->update();
\begin{lstlisting}


\subsection{Change the Track Filter}

  Finally we need to modify Driver::filterTrk(float accel) that it does allow
  accelerator commands in the pits. Change

\begin{lstlisting}
    if (car->_speed_x < MAX_UNSTUCK_SPEED) return accel;
```

to

```
    if (car->_speed_x < MAX_UNSTUCK_SPEED ||
        pit->getInPit()) return accel;
```

9.8.8 Test Drive

Now everything should compile and work fine. To test the pit stops you can insert the following line in the drive function to make the car stop on every lap. If it works you can get now your cold juice out of the fridge... Cheers;-)

```
    pit->setPitstop(true);
```

9.8.9 Final Remarks

That finally was it, you got your TORCS driver license, congratulations. I ask myself if anybody will read till here, so in case you did, please let me know and tell me your impressions. The reason for me to write this tutorial was to make it easier for other people to build a robot and to understand the problem better. I tried to separate the different functionality of the robot into filters and classes, the goal was to keep them as independent as possible. In fact the structure is much better than on my berniw robot, but I'm still not really happy with it. Have fun and contribute to TORCS.

Bye, berniw.

9.8.10 Downloads

In case you got lost, you can download my robot for TORCS 1.2.0 or later.

9.8.11 Feedback

Let me know if you read this chapter and your thoughts about it. Please send me also spelling, grammar, math and code corrections. Thank you for the feedback.

9.8.12 Summary

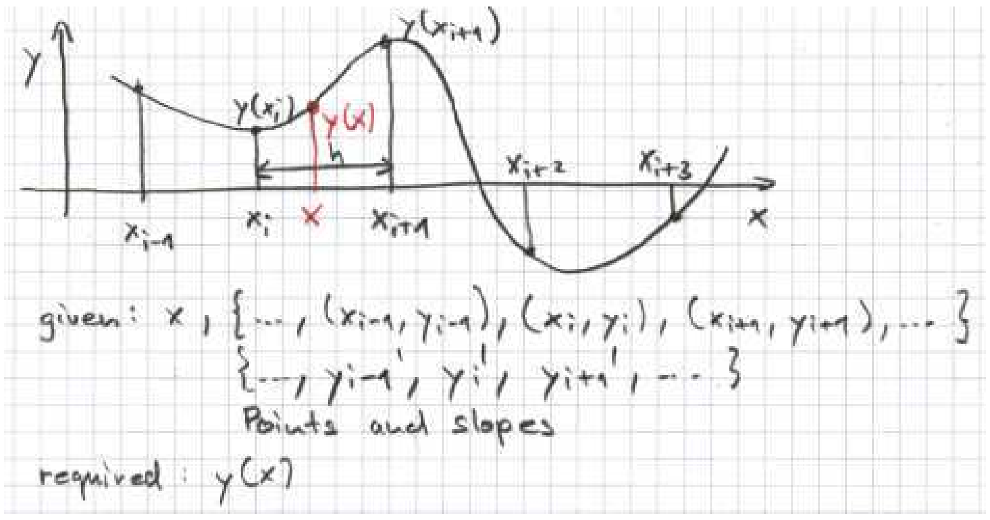
- You have understood a full featured robot and you are ready to improve it or build your own from scratch.

Appendix A

The Math of Cubic Splines

A.1 Problem Statement

Here I show you how to derive the spline algorithm we use. First let us state the problem. Given is a set of points which belong to an unknown function y=f(x). We also know the first derivative of the function in the given points y′=f′(x). What we want to compute is an approximation of the unknown function value y for a given x.



y
$y(x_{i+1})$
$y(x_i)$
$y(x)$
h
x_{i+2}
x_{i+3}
x
x_{i-1}
x_i
x
x_{i+1}
given: x, $\{\ldots, (x_{i-1}, y_{i-1}), (x_i, y_i), (x_{i+1}, y_{i+1}), \ldots\}$
$\{\ldots, y_{i-1}', y_i', y_{i+1}', \ldots\}$
Points and slopes
required: $y(x)$

Figure A.1: spline picture an problem statement

A.2 Algorithm Overview

Like you can see is the x value for which we want to compute y enclosed between two points. In a first step the algorithm finds the surrounding points with binary search. After that we interpolate with the information provided by the two surrounding points.

A.3 Spline Interpolation

The cubic splines are polynomials of the form: $y = a*x^3+b*x^2+c*x+d =: P(x)$

To evaluate the above expression we have to find the parameters a, b, c and d. If we have the parameters we can simply evaluate the expression with the given x and we will get y.

$$\begin{array}{ll} P(x_i) = y_i & P(x_{i+1}) = y_{i+1} \\ P'(x_i) = y_i{}' & P'(x_{i+1}) = y_{i+1}{}' \end{array}$$

We call the above polynomial P, and state four equations for the four unknown parameters a, b, c and d. The first line states the equations of the two surrounding points, which are known, and the second line shows two equations for the first derivatives, which are also known. Now you could already solve this system, but it would lead to quite unpleasant expressions.

$$t = \frac{x - x_i}{h} \qquad h = x_{i+1} - x_i \qquad t = 0..1$$

To avoid that we introduce the new parameter t. Like you can see from the definition and the expressions it is 0 at xi and 1 at xi+1. This will be very pleasant, because evaluating a polynomial for 0 or 1 is much easier than for most of the other numbers.

Now we work with t, so I state a new polynomial $Q(t) := a * t^3 + b * t^2 + c * t + d$

$$
\begin{aligned}
Q(t) &= P(x_i + t \cdot h) \\
Q(0) &= P(x_i + 0 \cdot h) = P(x_i) = y_i \\
Q(1) &= P(x_i + 1 \cdot h) = P(x_i + x_{i+1} - x_i) = P(x_{i+1}) = y_{i+1}
\end{aligned}
$$

We need to convert the given points from P(x) to Q(t), and we get the expected Q(0) and Q(1).

$$
\begin{aligned}
Q'(t) &= (P(x_i + t \cdot h))' = h \cdot P'(x_i + t \cdot h) \\
Q'(0) &= h \cdot P'(x_i + 0 \cdot h) = h \cdot P'(x_i) = h \cdot y_i' \\
Q'(1) &= h \cdot P'(x_i + 1 \cdot h) = h \cdot P'(x_{i+1}) = h \cdot y_{i+1}'
\end{aligned}
$$

The same applies for the derivatives P'(x), we convert it to Q'(t) and get Q'(0) and Q'(1).

$$
\begin{aligned}
Q(t) &= a \cdot t^3 + bt^2 + ct + d \\
Q'(t) &= 3 \cdot at^2 + 2 \cdot b \cdot t + c
\end{aligned}
$$

Now we compute the derivative of the concrete polynomial.

We evaluate the equations for t=0 and get the parameters c and d.

We evaluate the equations with t=1, and get two equations with two unknown parameters. You can solve this system for a and b.

Finally we got all parameters a, b, c and d. We can now evaluate the polynomial. The expression used in the concrete algorithm uses the Horner-Schema to improve efficiency.

A.4 Summary

- You know how the spline works.

$$Q(0) = d = y_i$$
$$Q'(0) = c = h \cdot y_i'$$

$$Q(1) = a + b + c + d = y_{i+1}$$
$$Q'(1) = 3a + 2b + c = h \cdot y_{i+1}'$$

$$b = -3y_i + 3y_{i+1} - 2 \cdot h \cdot y_i' - h \cdot y_{i+1}'$$
$$a = -2 \cdot y_{i+1} + h \cdot y_i' + 2 \cdot y_i + h \cdot y_{i+1}'$$