Monograph:
Gallimore, S. (1993) A Road Description Language for the Leeds Driving Simulator Guide (V1.0). Working Paper. Institute of Transport Studies, University of Leeds , Leeds, UK.

Working Paper 389

Published paper

Gallimore, S. (1993) *A Road Description Language for the Leeds Driving Simulator Guide (V1.0).* Institute of Transport Studies, University of Leeds. Working Paper 389

UNIVERSITY OF LEEDS
Institute for Transport Studies

ITS Working Paper 389 *ISSN 0142-8942*

January 1993

A ROAD DESCRIPTION LANGUAGE FOR THE LEEDS DRIVING SIMULATOR USER GUIDE (V1.0)

S Gallimore

CONTENTS

Page

ABSTRACT

GALLIMORE, S (1993). A road description language for the Leeds driving simulator user guide (V1.0). *ITS Working Paper **389***. Institute for Transport Studies, University of Leeds.

A driving simulator has recently been developed at the University of Leeds. Part of this work has been to provide a method of creating a wide variety of road networks to meet the demands of different experiments. This paper describes a simple language that specifies road networks and their appearance, including the definition of road markings, sign posts and roadside objects. It is intended for use by prospective users of the simulator facility in order that they could either build networks themselves or know what information is required for simulator staff to build a network for them.

KEY-WORDS: Driving simulation; scene databases; road networks.

Contact: Stephen Gallimore, Institute for Transport Studies (tel: 0532-335730)
E-Mail: stephen@psyc.leeds.ac.uk

A ROAD DESCRIPTION LANGUAGE FOR THE LEEDS DRIVING SIMULATOR USER GUIDE (V1.0).

1.INTRODUCTION

This document describes the language and compiler used to generate scene databases for the Leeds Driving Simulator. Trying to build a database, by hand coding a program to generate the required database objects, is impractical. While not impossible it would prove to be very time consuming and the resulting program would be very difficult to modify and reuse. The ideal tool for building road systems would be one that allowed the user to interactively draw the road and white lines using some graphical interface, signs and other objects being placed with a single click of the mouse. However it was considered that this would take a large amount of time to create, so while this remains a long term goal it was decided that a text based system would be created first. Eventually it is expected that this will be merged with the Scene Database Preview program and gradually a graphical interface will be substituted for the text system.

2.USING THE COMPILER

The database compiler is called *cr* - compile road, its use is very simple:

$ cr *source_file database_file* [cpp options]

The *source_file* contains a text description of the database to be built. The database will be created with the filename *database_file*, overwriting any previous file of that name. The source file is first passed to the C preprocessor generating a temporary file which is then passed to the actual *build* program. This file is created in /tmp unless the environment variable TMPDIR is set, in which case its contents are used as the temporary directory path name. Any command line options after the database filename are passed to /usr/lib/cpp, allowing for instance additional include search paths to be specified (see the cpp manual page for further details). The default road library directory is automatically passed to cpp, this contains include files with standard definitions and pre-defined pieces of road which may be included into your databases. The file *standard.rd* should be included at the top of your files as it includes some basic definitions which are used throughout this text.

An example compilation command might look like this:

$ cr motorway.rd motorway.db -Imyroadlib

3.SIMPLE ROADS

3.1THE MAIN ROAD

A source file contains a number of definitions which consist of pieces of road. Each definition has a name which must start with a letter and can have letters and digits after this. Both upper and lower case letters can be used and case is significant. No two definitions can have the same name and one of the definitions must have the name *main*. This will be the first definition to be translated into the

database although it does not have to be the first definition in the file. Here is an example:

```
#include <standard.rd>

/* A first simple road map */
main {
 straight 500.0 {}
  road1
  road1
}

road1 {
  curve left 700.0 300.0 {}
  straight 250.0 {}
}
```

This file defines a road network containing a straight road five hundred metres long. The straight is followed by two copies of *road1*, which consists of a left hand curve with a radius of seven hundred metres and length three hundred metres and another straight, two hundred and fifty metres long. Definitions may be called as many times as you like, but these must not lead to direct or indirect recursion, that is calls must not form a loop. This example also shows the use of facilities provided by the C pre-processor, include files and C style comments are two such facilities. Finally this example shows that all distances are measured in metres.

3.2THE CURRENT POSITION

After every piece of road has been built, the current position and direction of travel is updated to match the end of the new road. The compiler maintains a current environment which contains the:

- width of the road;
- direction of travel, in degrees;
- position in the world;
- precision of curves.

Initially the width is set to 9.3m, a single carriageway road with 1m hard strips. The direction is set to zero (due North), the position set to the (x,z) origin and the curve precision set to one degree. For reasons associated with the graphics system we work with the X and Z axis, where Z takes the role of Y in a graph. In fact the Y axis is the up/down axis, that is height, in the graphical model. Furthermore if we look down on the world the -Z axis goes *up* the page, hence if we are going due North (the default) we are going down the -Z axis. Most of the time all of this is hidden to the user, everything is defined relative to the current position and direction. However it may be necessary to directly set the direction and position, in this case you must be aware of how the axis work. The curve precision determines how curved roads and white lines are turned into polygons. If the arc of the curve is greater than the precision it is split up into polygons every *precision* degrees. The default setting should cope with any curve but this may generate more polygons than really necessary for most, this may become significant in reducing the size of large databases and in keeping the number of polygons down to a manageable level. The following commands directly affect the environment without building a road:

- width <*width in metres*>
- position *X Z*

- direction <*angle in degrees*>
- turn [left|right] <*angle in degrees*>
- precision <*angle in degrees*> or prec <*angle*>

In all the commands that need a left or right qualifier, the first letter, that is *l* and *r*, can be used as an abbreviation. In addition to the above there are three control flags contained in the environment and three key words to set them, these are:

- build [on|off]
- step [on|off]
-verbose [on|off]

The build flag allows the building of graphical objects to be turned on or off. When turned off, no objects are added to the scene database but the environment is updated as if they were. When step is turned on only one command is interpreted at a time, currently this is of little use but will be used as a debugging tool when the compiler is integrated with the graphical system. The build flag works in a nested fashion, if turned off more than once it must be turned back on again the same number of times before the building of objects resumes. When verbose mode is turned on information about each generated road segment is printed on the console.

3.3STRAIGHT

The simplest piece of road is a straight, this has constant width and does not change the direction of travel. The syntax for a straight is:

straight <*length in metres*> { }
or
str <*length*> { }

Normally there would be commands between the curly braces describing road furniture such as white lines and road signs, these will be described later. This command will build a road using the current width and road direction, the centre line of the road starts at the current position and ends length metres away in the current road direction. An example of a straight is:

```
straight 250.0 { }          /* A 250m straight road */
```

3.4CURVES

The next most basic piece of road is a constant radius curve, that is an arc of a circle. The syntax for a curve is:

curve [left|right] <*radius in metres*> <*length in metres*> { }
or
cu [left|right] <*radius*> <*length*> { }

The *length* refers to the length of the centre line which is an arc of a circle of radius *radius,* starting at the current position with its initial tangent being the current direction. The centre of the circle and hence the direction the road turns is determined by the left or right qualifier. The current environment is updated with the position and tangent at the endpoint of the centre line, see figure 1.

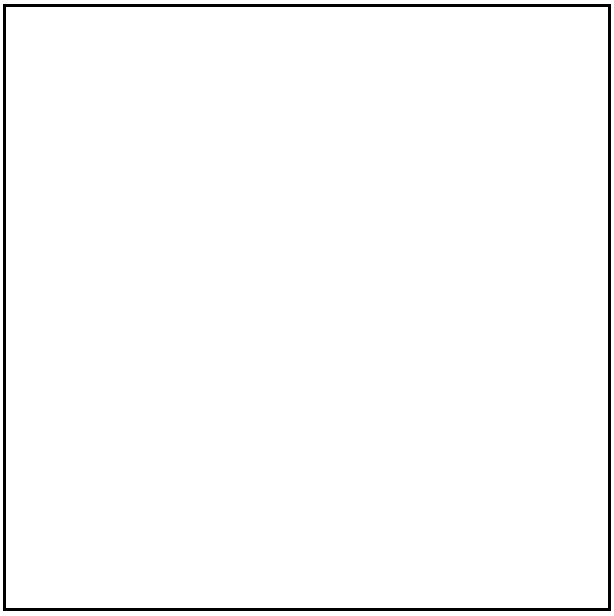

Figure 1

3.5BRANCHES

Branches allow the quick construction of simple junctions by temporarily allowing construction to take place along a path at ninety degrees to the current road direction. The syntax is as follows:

branch [left|right] { *<road statements>* }
or
br [left|right] { *<road statements>* }

The following example will make its use clear.

```
main {
  width 10.0                    /* Major road, no hard strips */
  str 150.0 { }
  br l {                  /* Branch left, using the abbreviated form */
    width 7.3                   /* Minor road, no hard strips */
    str 50.0 { }
    cu r 300.0 200.0 { }
  }                             /* End of Branch */
  str 20.0 { }                  /* Continuation of the original road */
  cu l 700.0 300.0 { }
}
```

Here we have a branch off the main road consisting of a narrower road made up of a short straight and right bend. Any changes to the current environment are local to the branch, hence when the statement:

str 20.0 { }

is translated the road width is again ten metres and the position and direction are returned to those values before the branch, that is back to the end of the initial straight. Inside the branch the direction is initially set at right angles to the original direction, either to the left or the right. The starting position of the new road is at the left or right edge of the previous road, the width of that road is used to calculate this. Branches may be nested at will to produce complex road networks. Only ninety degree turns are created with branches, although the *turn* statement could be used in a branch to alter this to help create, for example a slip road.

3.6WHITE LINES

Adding white lining to a road is done by placing pieces of lines relative to the road they are *painted* on. Lines may be straight or curved, solid or dashed, the main restriction is that curved lines cannot be placed on straight roads. There are four basic variations of lines:

solid <*line type*> <*width*> <*start length*> <*end length*>
straight <*start offset*> <*end offset*>

solid <*line type*> <*width*> <*start length*> <*end length*>
curve <*offset*>

dashed <*line type*> <*width*> <*start length*> <*dash length*> <*gap length*>
<*number of dashes*> straight <*start offset*> <*end offset*>

dashed <*line type*> <*width*> < *start length*> <*dash length*> <*gap length*>
<*number of dashes*> curve <*offset*>

Solid straight lines can be placed on straight and curved roads. Valid line types are FA and HA (defined in standard.rd), giving fully and half anti-aliased lines respectively. If several lines are going to be connected together at their ends then HA should be used, this avoids dark lines appearing at the joins. The white line is placed relative to the centre line of the road, starting and ending at the specified lengths along this line and the start and end points shifted tangentially from the centre line by the start and end offset respectively. A positive offset shifts to the right of the centre and a negative value to the left. Note that when putting a straight line on a curved road the tangent at the end point will not be the same as the start because of the change of direction along the curve. Also note that curved lines can not be currently placed on straight roads.

With dashed lines an end length is not directly specified, it is calculated from the other parameters. In the case of curved dashed lines, the size of the dashes and gaps refers to the sizes they would be if placed along the centre line. They get larger as you shift them towards the outside of the curve and smaller when shifted towards the inside. This may seem strange but it does allow all the dashed carriageway markings on motorways to line up at the ends, using the same size parameters. There are two values of *line type,* STRT and CURVED, defined in standard.rd. Only STRT is valid for straight lines but both are valid for curved lines. In this case the line type determines whether the dashes are straight or curved. For most curves using straight lines is desirable, in order to keep the number of polygons used to a minimum, this is often the case on real roads anyway. However on radical bends, particularly in urban situations the white lines need to curve with the road.

The following is an example of some white line markings:

```
str 150.0 {
  solid HA 0.1 0.0 150.0 str -4.8 -4.8      /* Line along the while left edge */
  solid FA 0.1 100.0 100.0 str -4.8 4.8     /* A line directly across the road */
  dashed STRAIGHT 0.2 20.0 2.5 0.5 30.0 str 0 0       /* Dashed lines along the
                                  centre of the road */
}
```

4.MORE ROAD PRIMITIVES

4.1PATCHES

A patch is a straight road which varies in width along its length, this is useful for the creation of roads that vary in width, laybys and so on. The syntax of the statement is:

patch <*length*> <*left width*> <*right width*> { [*road furniture*] }

The centre line is the same as for a straight road, however the distances between the centre and the edges at the end of the patch are controlled by the arguments. A new centre point and width is then calculated for the environment. White lines are placed in the same way as for straight roads.

4.2CORNERS

There are four corner pieces available to round off junctions. The syntax of these is:

corner [left|right] <*type*> <*radius*>

There are the following definitions in standard.rd for the *type* argument:

- LEFT_FORWARD
- RIGHT_FORWARD
- LEFT_BACK
- RIGHT_BACK

The corners are placed on the specified edge of the road, this does not change the current environment, nor can white lines be placed on corners at the moment.

5.ROAD SIGNS

5.1SIGN POSTS

Road signs may also appear in the road furniture part of road primitives along with white lines. First a pole is defined, then any number of signs may then be attached to the pole. The syntax is as follows:

pole simple <*type*> <*length*> <*offset*> <*height*> <*radius*> { <*signs*> }
or
pole complex <*length*> <*offset*> <*height1*> <*radius1*> <*height2*> <*radius2*>
{ <*signs*> }

In the simple case *type* can be either SQUARE or ROUND, these are defined in standard.rd. A pole, either a flat shaded square tube or a smooth shaded hexagonal tube, which looks round because of the shading, is created at a position relative to the centre line of the road in the same way as white lines. The pole's height and radius is specified as well, in the case of a square pole the radius is half the length of the sides. The square pole is rotated to the direction of the road, taking into account the change in direction of curves, so that the faces of the pole are aligned with the edge of the road.

A complex pole is always hexagonal and consists of two parts, of different height and radius, one placed on top of the other with an angled *sleeve* joining the two parts. It is used to create lit poles with a fat power box at the bottom with a thinner pole on top for the actual signs.

The syntax for the signs is as follows:

<*Sign Name*> <*height*> <*angle*>

The name of the sign is looked up in a database of signs held in the file signs.db in the same directory as the standard header file (a list of valid road sign names is given in appendix B). If an unknown name is given to the compiler an error message is displayed and that sign is ignored, the compiler then continues with the next statement. The sign is placed at the specified height up the pole and is rotated around the outside of the pole so that it would be facing a driver traveling along in the same direction as the road is being built. It is then rotated further by *angle* in a clockwise direction. An example of signposts is given on the next page:

```
str 50 {
  pole simple ROUND 25.0 -6.0 3.0 0.05 {
    NationalSpeedLimit 2.6 0.0
  }
  pole complex 25.0 6.0 1.0 0.06 2.0 0.04 {
    Clearway 2.6 180.0
  }
}
```

5.2ROAD FURNITURE

It is also possible to add other objects to the scene either on the road or by the roadside. The syntax for this is:

object <*object name*> <*length*> <*offset*> <*angle*> <*height*>

The object name is looked up in a database of available objects. If found the object is placed relative to the road in the same way as white lines and sign posts and is oriented in the same way as a road sign. Additionally it is placed at the specified height above the current road surface. An example is:

```
str 30 {
 object RedRover216 15 - 1.5 0 0.005
}
```

It is also possible to print the exact position and orientation that an object would be placed in by using the command:

print <*identifier*> <*length*> <*offset*> <*angle*> <*height*>

6.ROAD PATHS

6.1ENVIRONMENT STACK

So far we have only used the current environment and hence a single path, except in the case of a branch where the environment is saved then restored at the end of the branch. The compiler keeps a+ stack of environments, it is the environment at the top of this stack that is used to build roads. There are a number of statements that allow direct manipulation of this stack:

- drop <*number of items*>
- copy <n^{th} *item*>
- swap
- rotate

Drop removes items from the stack and copy makes a duplicate of the specified item (where 1 is the original top of the stack) on top of the stack. Swap, swaps the top two items on the stack and rotate turns around the top three items on the stack such that the third item becomes the top of the stack, the top the second and the second the third.

6.2MANIPULATING PATHS

The branch statement is a shortcut for the following code:

```
copy 1
turn l 90.0
build off
str 5.0 /* if a ten metre wide road */
build on
/* Statements in the branch */
.
.
drop 1
```

Directly manipulating the stack is useful for creating multi-exit pieces of road such as cross roads and roundabouts and putting them into definitions on their own so that they can be used again and again. Here is an example of a simple roundabout which leaves three items on the stack, one for each exit.

```
roundabout {
  build off
  str 5.0 { }
  build on
  br l {
    swap
    drop 1
    cu r 20 31.415927 { }
    exit
    cu r 20 31.415927 { }
    exit
    cu r 20 31.415927 { }
    exit
    cu r 20 31.415927 { }
    str 10 { }
  }
}

exit {
  str 5 { }
  br l {
    str 10 { }
    copy 1
  }
  swap
  str 5 { }
```

6.3DEFINING NETWORK TOPOLOGY

As well as the graphical description of the road it is also possible to specify the topology of the road netowrk. This creates a cyclic graph of road paths linking together nodes which are either junctions or a road end point. For each link an ordered list of road segments is automatically generated using information from the graphical description of the road. A path between two points is started by the junction command:

junction <*type*> <*start junction number*> <*end junction number*>

A path is ended either when another junction command, branch instruction or stack operation is encountered. Between these two points all road segments are assumed to belong to this path. Paths can either be created in both directions along the road or only in one of the directions. This is controlled by the type argument which take the values TWOWAY, F_ONLY and B_ONLY. F_ONLY specifies that only the forwards path should be built, that is the one going in the direction the graphical road is being built. B_ONLY creates only the opposite path to the direction the road is being built.

APPENDIX A

ROAD SIGN NAMES

StopCommand
OneWay
NoThroughRoad
NoEntry
NationalSpeedLimit
Clearway
AheadOnly
TurnLeft
KeepLeft
TurnLeftAhead
TurnRightAhead
NoVehicles
NoRightTurn
NoLeftTurn
GivePriority
GivewayCommand
GivewayWarning
CrossroadPriority
Crossroad
JunctionLeftPriority
JunctionRightPriority
TJunction
LFStaggeredXPriority
RFStaggeredXPriority
LFStaggeredX
RFStaggeredX
LeftMinorJunction
RightMinorJunction
StopWarning
BendToLeft
BendToRight
DoubleBendToLeft
DoubleBendToRight
DualCarriageWayEnd
TwoWayTrafficAhead
TwoWayTrafficAcross
TrafficMergesFromLeft
TrafficMergesFromRight
RoadNarrowsBothSides
RoadNarrowsOnLeft
RoadNarrowsOnRight
ChangeCarriageWay
BlueCountdown300
BlueCountdown200
BlueCountdown100
GreenCountdown300
GreenCountdown200
GreenCountdown100
RGreenCountdown300
RGreenCountdown200
RGreenCountdown100

APPENDIX B

SYNTAX SUMMARY

Definition
    name { [Road Primitives] }

Road Primitives
    straight length { [Road Furniture] }
    str length { [Road Furniture] }
    curve [**left** | **right**] radius length { [Road Furniture] }
    cu [**left** | **right**] radius length { [Road Furniture] }
    branch [**left** | **right**] { [Road Primitives] }
    br [**left** | **right**] { [Road Primitives] }
    patch length left_width right_width { [Road Furniture] }
    corner [**left** | **right**] type radius
junctiontype start_junction end_junction

    width width
    position X Y
    direction angle
    turn [**left** | **right**] angle
    precision angle
    prec angle
    build [**on** | **off**]
    step [**on** | **off**]
verbose[**on** | **off**]

    drop item
    copy item
    swap
    rotate

Road Furniture

    solid type width length length **straight** offset offset
    solid type width length length **curve** offset
    dashed type width length length dashes **straight** offset offset
    dashed type width length length dashes **curve** offset

    pole simple type length offset height radius { [Signs] }
    pole complex length offset height radius height radius { [Signs] }
objectname length offset angle height
printidentifier length offset angle height

Signs

name height angle