PopRacer: Final Project Report

Project Developers:

Michael Brook
Damien Clark
Anushka Gunawardana
Mark Stephen Rowan
H. Francis Tedom Noumbi
Peter Zeidman

Introduction

In recent years we have seen a massive interest from not only scientists but also people of all denominations in the development of Artificial Intelligence (AI).
In its short existence, AI has increased understanding of the nature of intelligence and provided an impressive array of applications in a wide range of areas. It has sharpened understanding of human reasoning and of the nature of intelligence in general. At the same time, it has revealed the complexity of modelling human reasoning, providing new areas and rich challenges for the future.
David B. Leake from the University of Indiana (USA) defines AI as the "*science that studies the computational requirements for tasks such as perception, reasoning, and learning, and develops systems to perform those tasks*". It addresses a wide range of problems, uses a variety of methods, and pursues a spectrum of scientific goals.
AI research in the area of *cognitive science* for example has developed models that have helped to understand human cognition. Applied AI research has provided high-impact applications systems that are in daily use throughout the world. AI technology has had broad impact. In fact AI components are embedded in numerous devices, such as copy machines that combine case-based reasoning and fuzzy reasoning to automatically adjust the copier to maintain copy quality. AI systems are also in everyday use for tasks such as identifying credit card fraud, configuring products, aiding complex planning tasks, and advising physicians.
AI technology is being used in autonomous agents that independently monitor their surroundings, make decisions and act to achieve their goals without human intervention. For example, in a 1996 experiment called ``No Hands Across America," the RALPH system [Pomerleau and Jochem1996], a vision-based adaptive system to learn road features, was used to drive a vehicle for 98 percent of a trip from Washington, D.C., to San Diego, maintaining an average speed of 63 mph in daytime, dusk and night driving conditions. Such systems could be used not only for autonomous vehicles, but also for safety systems to warn drivers if their vehicles deviate from a safe path.
Considering all these applications of AI and this last example in particular that we were intrigued and curious to know if it was possible to reproduce such experience in our own ways and in a very exciting and challenging environment so that it can be in use in the real world.
The car racing industry is claiming more and more popularity; it is followed by all kinds of people all over the world therefore generating massive income and providing lots of excitement to the fans of speed. Unfortunately racing remains a very dangerous

© Team popCorn

activity which has proven to be deadly (the Brazilian Ayrton Senna in 1994 at the San Marino, a Formula 1 Grand Prix and many others).
Could it be possible to have those races and the excitement they provide but without having to worry about the injuries and deaths i.e. by having real cars in a real environment dealing with all the parameters of a real race but with the only difference that the human drivers are replaced by intelligent agents?
This project (**popRacer**) is an attempt to provide an answer or, to be modest, a beginning of answer to the question here above. Our aims to produce agents that can drive around a racing track in an optimal way. The project development is articulated around three main points:

- The representation of the environment (racing track)
- The development of the algorithms that will represent the brain of the agent
- The gathering, analysis and processing of the environmental parameters that exist between an agent and its environment (friction, wind force, velocity, acceleration, etc…)

The report is divided into sections, each addressing a point made earlier and explaining the techniques we intend to use as well as the problems encountered and the alternative solutions applied:

- The Bezier Curve and the pixel colour recognition for the graphical representation of the environment (track) and is further developed in the Graphics section (by Mark Rowan)
- The Genetic algorithm and multi-layer perceptron Neural Networks are intended for selecting and evolving the cars and controlling the decision making process. See the Brain section for further details (by Peter Zeidman)
- The Architecture section of the report deals with the interaction between the agents, the physics model involved and the environment (by Michael Brook)
- The Physics section processes all the parameters that intervene in the motion of the agents in order to constraint them to it environment (Anushka & Damien)
- Adapted Version of the K-Neighbours learning algorithm for the collision detection (by Francis Tedom Noumbi)

We will terminate the report with:

- An Evaluation section assessing the progress made and identifying the possible areas of improvement
- A User Guide
- A final Assessment (conclusion) section that summarizes the project
- A Bibliography and Acknowledgments section
- A set of three appendices for the program code, the self assessment forms and other paperwork.

To develop the project the team had weekly meetings (see appendices for minutes) where we program together, report and discuss the individual tasks assigned to individuals when not possible to work altogether on specific parts of the projects.

Pop-11 is the programming language that we chose for the implementation.

© Team popCorn

I. Graphical Representation (Graphics)

Track representation solutions

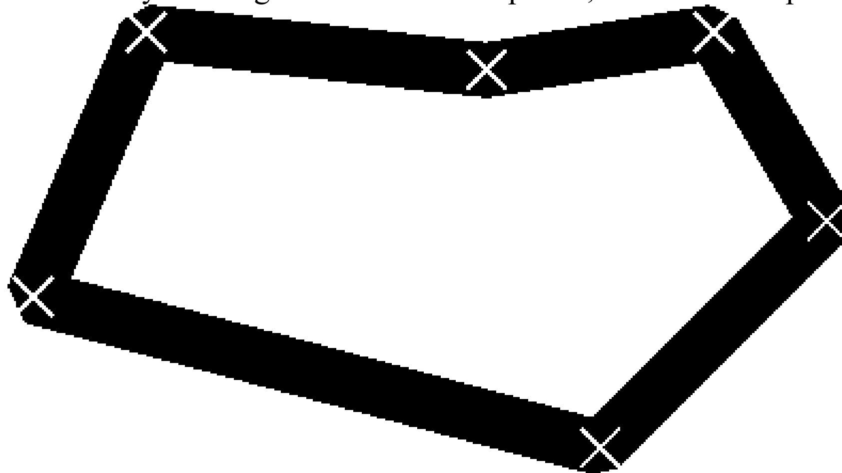
When attempting to represent the track we had several options available to us. We needed something that could be easily and quickly created by the user but which would also be mathematically computable so the simulation engine can determine whether a car is on the track or not.

The options considered were:

- User-drawn track, eg. with points selected by the user and then connected using straight lines or curves.
- Pre-defined mathematical functions eg. x^2, x^3, $1/x$, etc. joined together.
- Constructing a track using other objects eg. lines laid out to close off an area of the screen and form it into a track shape.

Here we go into greater detail for each option:

User-Drawn track
The idea would be to have some sort of a track editor where the user inputs points either textually or graphically (by clicking on the screen). The track itself would then be created by drawing lines between the points, as in the example below.



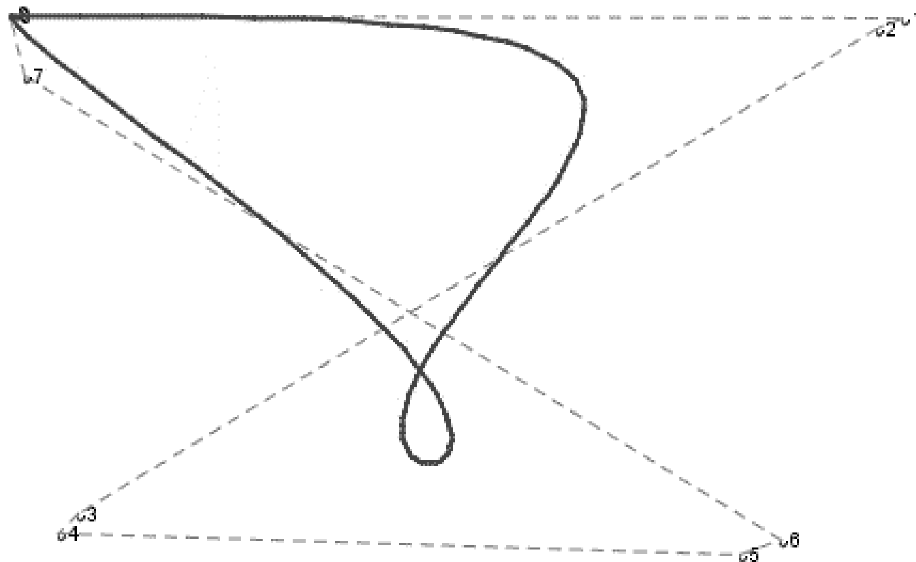
This would be very simple to set up as drawing straight lines between points is a common feature in practically all graphical programming languages (including Pop-11).
This has two main drawbacks though:
1). The track has only straight lengths and ugly square corners. Cars would still be able to take curved paths inside the track due to the thickness of the track but it is not the default encouraged action.
2). Points along the path are hard to compute as they do not belong to any function - they are all completely arbitrary and user-defined and only the actual control points laid down by the user are known. This makes it potentially computationally more expensive to find out if a car is still on the track or not.

With these limitations in mind we then considered replacing the straight lines with curves. Curves are not straightforward to draw in most languages (including

Pop-11) and usually involve interpolating points and plotting very short straight lines to give the impression of a curve (a raster display cannot display true curves, but even a circle can be approximated by lots of short lines).

Curves can be calculated for a set of control points using an algorithm such as the Bézier curve-generation algorithm. One advantage of using curves created in this way is that, as the line is plotted by a function, a point describing a car's position can be passed in and checked if it lies on the line generated by the function. This provides an easy way of telling if the car is on the track or not.

The curves produced by the Bézier algorithm are usually very smooth although a corner can be made sharper by adding extra control points near it to the list of control points. Here is an example of a Bézier curve track, drawn by the applet at http://www.cs.bham.ac.uk/resources/courses/graphics/unit8.php



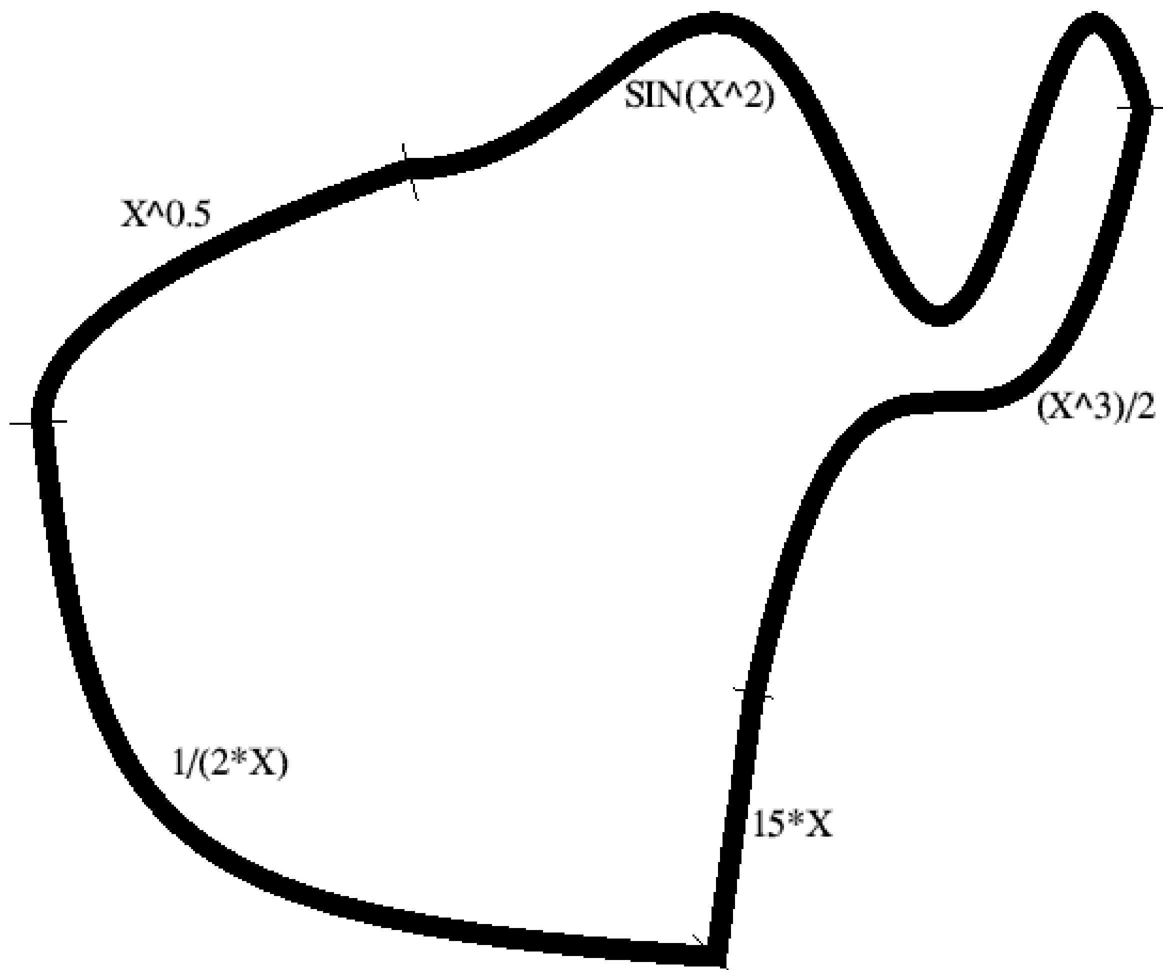
Note how the line goes through *none* of the points except for points 0 and 8, the start and end points. This is typical of a Bézier curve as the only time all the control points can intercept the line is if the line is completely straight. This can be confusing for people the first time they try to draw a Bézier curve as its construction can appear unpredictable.

The advantages outweigh these disadvantages. It is relatively simple to create a track with curved and straight sections, and the Bézier function itself is reasonably easy to implement in code.

Pre-defined functions

This is, in a way, similar to the Bézier idea discussed previously as a Bézier curve can be described by a series of mathematical functions *splined* together. The idea would be to use knowledge of the properties of certain graphs eg. x^3 to spline them together in such a way as to produce a track.

Here is an example:

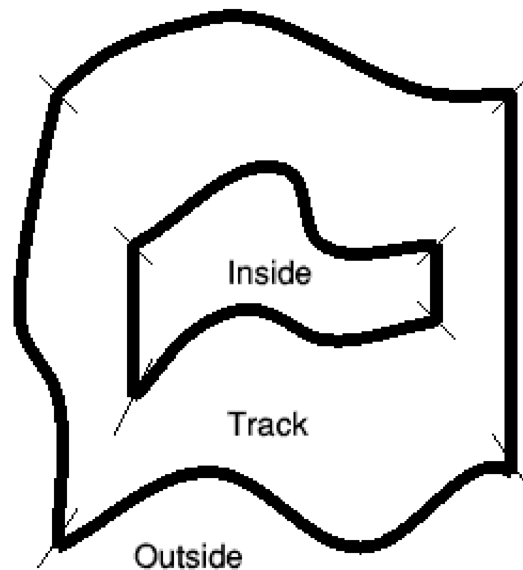


Clearly this is capable of producing a very nice track outline with hairpins, chicanes, smooth curves, and straight sections. Also because the functions are all relatively simple to compute it is elementary to pass in a car's current x-value and obtain a y-value from the graphs to determine if the car is on the track or not.

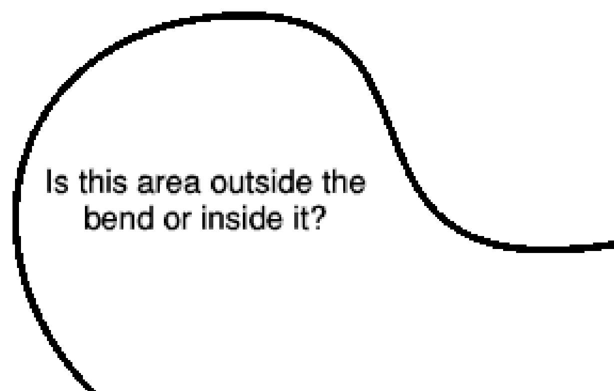
However actually implementing this idea is not so simple - x and y offsets for each graph have to be calculated (otherwise the graphs would all have the same origin!) and scaling has to be added correctly or the graphs will not match up. Also a reasonable mathematical knowledge is required on the user's part to be able to produce anything track-like, and even then it is down to a lot of trial and error.

Track objects

This idea was very different in that, instead of drawing lines to represent a track and bounding the cars to within *n* pixels of the line, we would use objects (including lines such as Bézier curves) to represent the edges of a track and not allow cars to cross over these objects.



This would allow variation in the widths of the track as can be seen in the example above but it is not an easy task to decide whether a car has gone off the track as areas have to be calculated between the objects and this presents many problems on its own. How do you determine which 'side' of an object the car is on? How do you even determine where the 'sides' of each object are? (Consider an arc – it doesn't enclose any area and could represent either the upper or lower wall of the track – you can't just assume for example that all points with an x value lower than the line are outside the track).



<u>Conclusion</u>

Having discussed all these options we had considered, we decided that the best representation for the track would be a user-drawn Bézier curve as it has a good trade-off between ease of programming, ease of construction of tracks for the user, and the ability to simply discover if a given point falls on the track or not.

Implementation of the Bézier curve

A set of control points has to be supplied (either manually or by a user clicking on the screen) along with the number of interpolations of the curve to be performed. A larger number of interpolations will result in a smoother curve. So a curve with this value set to 6 will have just 6 straight sections in the shape defined by the control points, and will look rather poor compared to a curve made up of 25 sections

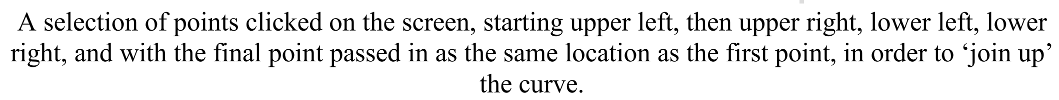
It is implemented using the following pseudocode. The list of control points and the output list of co-ordinates are both structured as a list containing two lists, the first for x values and the second for corresponding y values, eg. [[x_1 x_2 x_3 x_4][y_1 y_2 y_3 y_4]].

```
findBezierPoints(list of control points,
interpolations)
   stepsize = 1/(interpolations-1)
   n = number of control points

   for step from 0 to 1 by stepsize
         for k from 0 to n
               x = x + x(k)*blendingFunction(k,n,step)
               y = y + y(k)*blendingFunction(k,n,step)
         next k
         return x(step),y(step) ;;; append these to a
list
   next step
endproc

blendingFunction(k,n,step)
   return (n! / (k! * (n-k)!)) * (step^k * (1-u)^(n-k))
endproc
```

(Mathematical functions from Sorge, V. and Styles, I. "*Raster conversion algorithms for curves: 2D splines*").

An example to show the finished code (as seen in *bezier.p* in the appendices) working is as follows:
Input list of control points.
** [[0 7 29 64 111 166 227 290 349 400] [0 13 46 89 132 165 178 161 105 0]]
Output list of actual co-ordinates (20 interpolations).
** [[0 1 10 31 66 117 181 258 341 427 509 581 635 665 665 627 546 417 236 0]
[0 25 89 177 276 376 467 541 594 622 622 594 541 467 376 276 177 89 25 0]]

It was then a simple task to hook this code up to a graphical Pop-11 window which uses the rc_button library to listen for mouse clicks and adds the co-ordinates of these clicks to the list of control points, before passing them to the Bézier generation code. The returned list of actual co-ordinates can then be connected on-screen using simple straight lines to give the resulting curve.



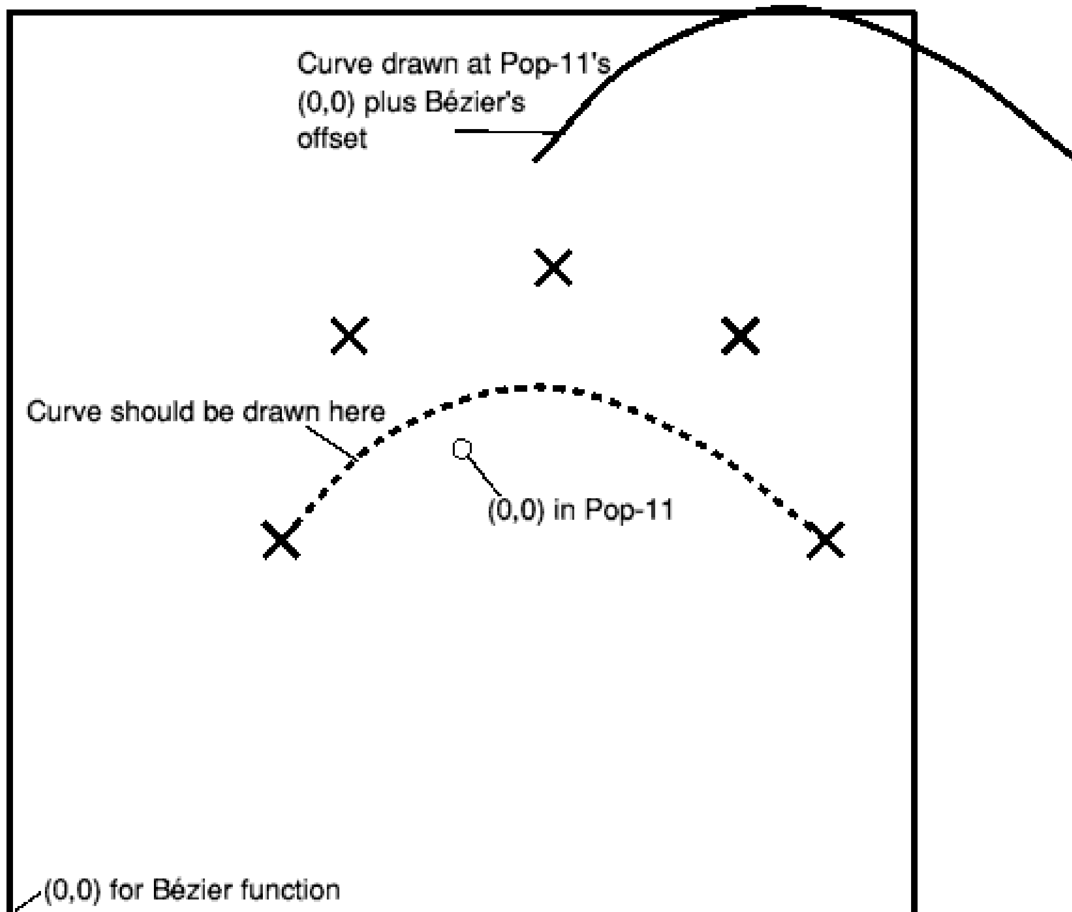
A selection of points clicked on the screen, starting upper left, then upper right, lower left, lower right, and with the final point passed in as the same location as the first point, in order to 'join up' the curve.

The resulting Bézier curve, once the points have been obtained and passed to the Bézier generation algorithm, and then returned as co-ordinates which have been joined with thick straight lines.

Dealing with Pop-11 coordinate system

Unfortunately Pop-11 uses a different coordinate system to what we were used to – rather than having (0,0) at the bottom-left corner, it puts it in the centre of the window.

However the Bézier generation algorithm only generates relative co-ordinates, ie. starting at (0,0) which would mean that any track drawn by clicking points on the screen would appear offset by (width_of_screen/2, height_of_screen/2).



Our solution was to note the Pop-11 co-ordinates of the first point clicked on (in the above example, approximately (-50, -25)) and then subtract this offset from all points before passing them to the Bézier function (as the Bézier function can only work on relative co-ordinates originating at (0,0)).

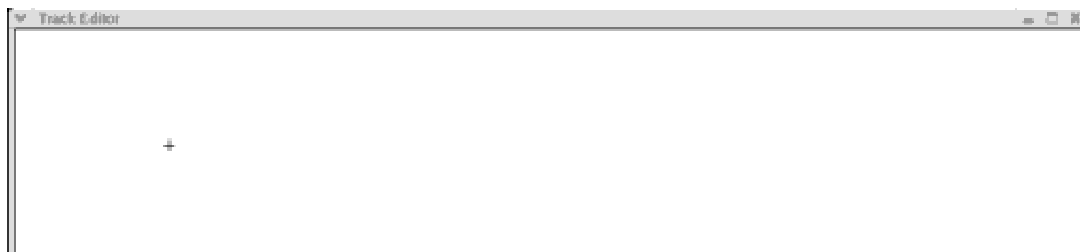
We then added the offset back onto the returned list of actual points so they could be plotted in the correct location on the screen.

Track editor

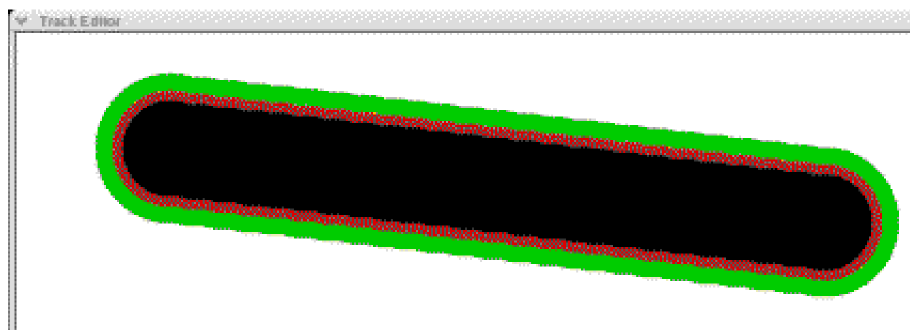
Once Bézier curves were shown to be reliable and easy for the user to generate we implemented a track editor integrated into PopRacer. This was based on the demonstration explained above but had the extra feature of actually drawing the track based on the current points the user has clicked on, updating itself with each new click.

The only extra code required was code to pass the current list of control points to the Bézier function every time a new point is added, and plot the resulting co-ordinates on the fly. One extra change that was made was to replace the straight lines between co-ordinates with the interpolated friction circles talked about in the architecture and physics sections.

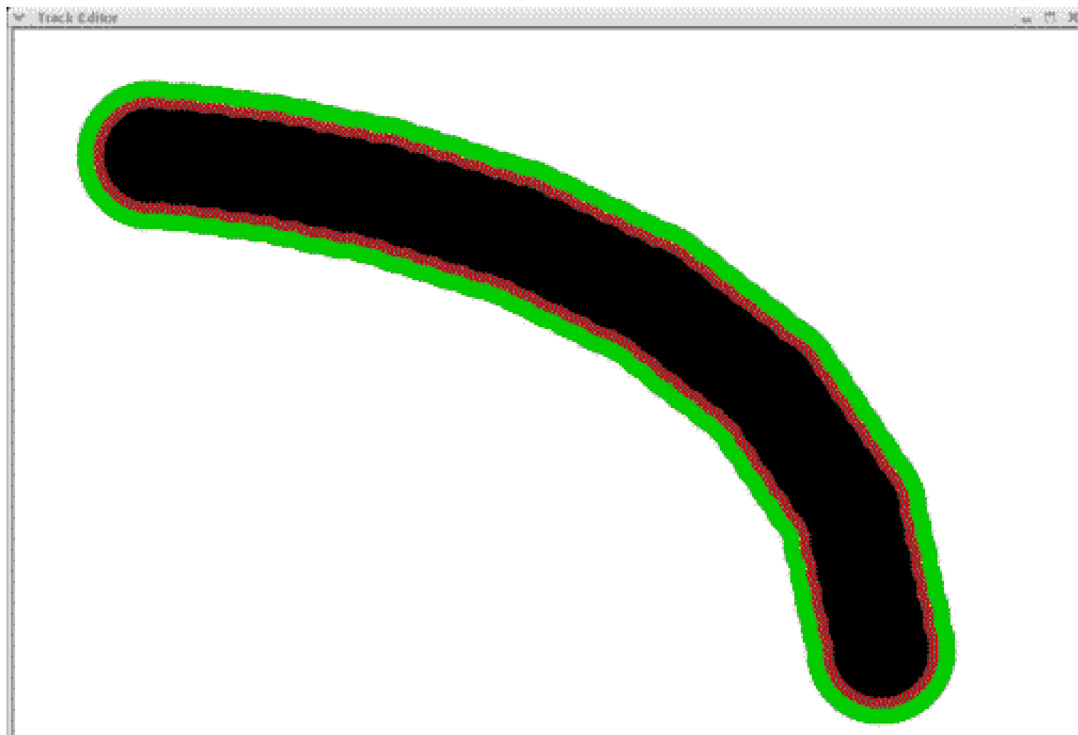
Here is a step-by-step example:



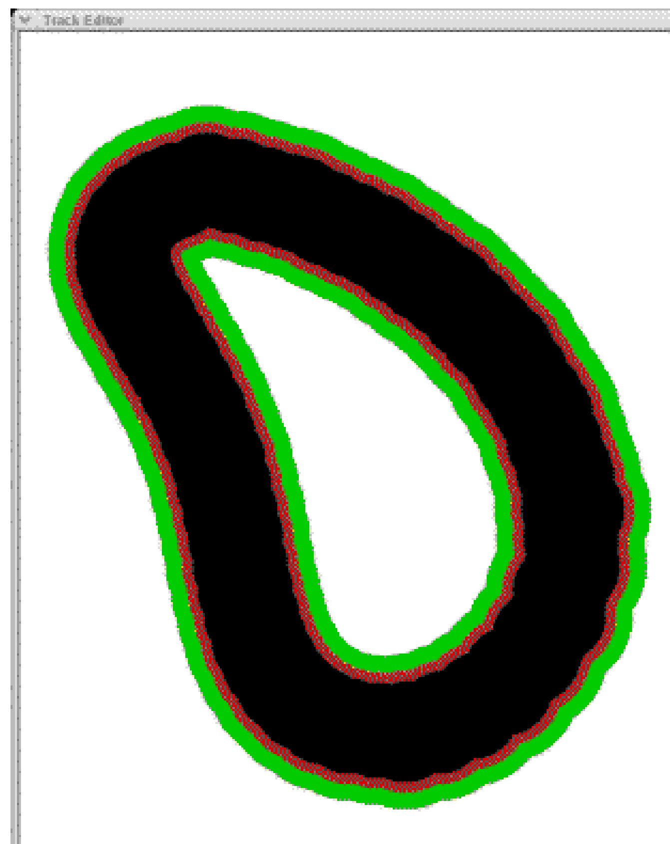
The first point is clicked.



Second point added, and the display refreshes to show the current track design (at this stage, a straight line).

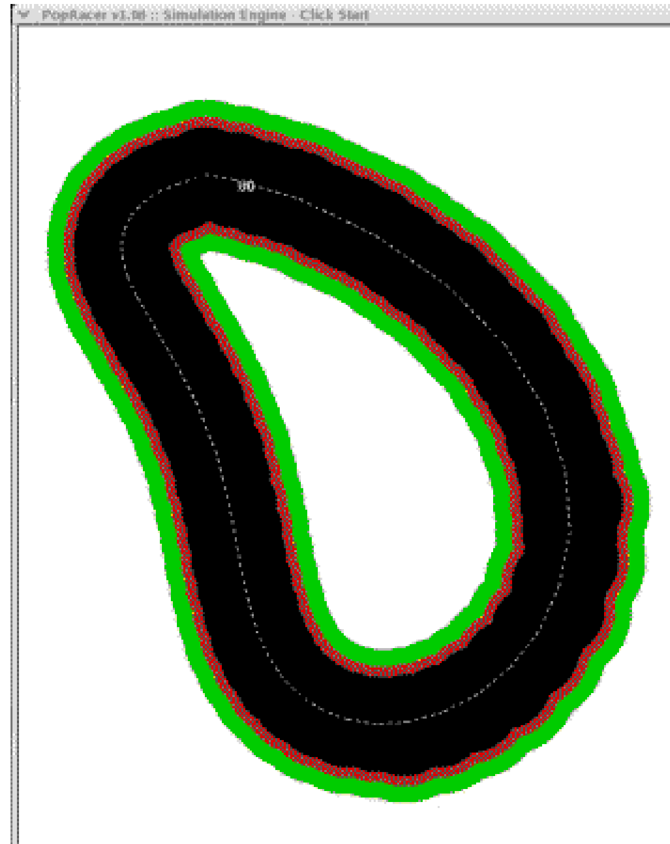


Third point, now more of a curve. The black track is surrounded by brown and green ‘mud’ and ‘grass’.

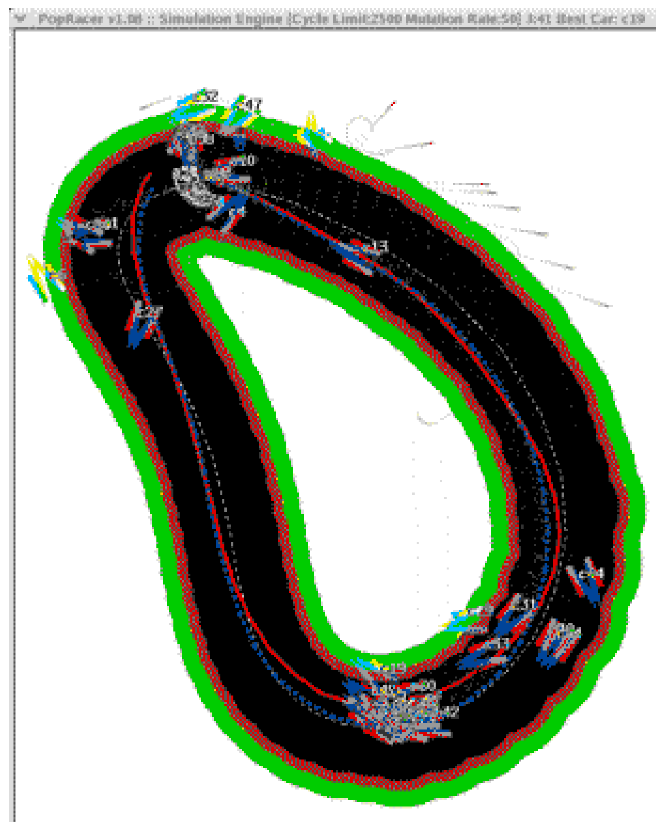


After a few more points the middle button is pressed and the track is closed into a loop.

© Team popCorn



The track has been saved from the editor, and loaded into PopRacer as the race track. Note the (very faint!) white dotted line drawn down the centre of the track.

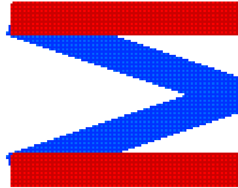


Cars are added to the simulation and evolved, and then the best and previous-best racing lines are drawn.

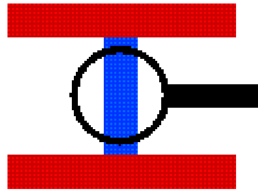
© Team popCorn

The cars

As mentioned in the architecture section, each car is a Pop-11 object which also contains a description of how it is drawn. We settled on a simple design made with four straight lines, as we found that even just by adding an extra line and small circle for one of our designs, the speed of the simulation when multiplied up to 50 cars was reduced by about 5-10%.



The design for our cars. It shows the forward direction of motion, and the wheels.



Our initial design – drawing the extra circle caused a 5-10% slowdown in a simulation with 50 cars!

The cars are plotted using the Pop-11 line library and are moved using the very useful ‘rc_move_by’ command which saves us having to erase and redraw the objects each time.

The way Pop-11 draws these lines is to combine the colours of the object and its background together bitwise using the logical XOR function. This works fine – and is very fast – for our cars on a white background. But most of the time the cars are on the black track, and the XOR’ed car appears inverted, with yellow wheels and a cyan body. This not only looks rather poor, it makes the cars harder to see as the contrast between the yellow and cyan is less than with our preferred red and blue.

So to make the cars look better on the track we inverted their default colours to become yellow wheels and a cyan body, which XOR with the black track to become red and blue respectively. This works much better except for when the cars stray off the track as they then become yellow/cyan, although this has the advantage that they are then less noticeable against the white background, hence being less of a distraction and helping the user to focus on the action happening on the track.

One other unavoidable problem using this method is that if two cars occupy the same position, their colours are XOR’ed and they turn invisible.

II. The Brain

Neural Network / Genetic Algorithm

Background

A controller was needed that would enable the agents to learn how to navigate a given race track, in the best possible time. It would need to fulfil a number of requirements:

- Circumnavigate the track in the shortest possible time
- Overcome obstacles presented by the physics engine, such as friction and limited acceleration
- Operate and learn without 'expert knowledge'; the system should only receive information to which a human driver would have access, such as the distance from the vehicle to the track's edges.

The chosen solution was to use neural networks (NNs). There were a number of reasons for this:

• Neural networks approximate functions; we could provide it with information about the environment, but didn't know what the perfect solution would be. The optimal behaviour isn't just to drive at full speed along the centre of the track; to deal with momentum a vehicle would have to accelerate on straights and slow before bends, just at the right time. We couldn't encode this knowledge in an expert fashion.

• We wanted to create agents that could perform well on any track provided. NNs, if well trained, have the ability to generalize to different situations.

•Directly programmed rules would have created a static level of success, only as good as the knowledge we could encode. Our aim was to produce agents that could perform better than us, and learn to improve on their own performance.

Design

Fundamentally, all implementations of neural networks share several common features. A number of inter-connected processors (*neurons*), capable of performing only simple tasks, are networked by weighted connections in such a way as to attribute varying relevance to each neuron's output. The weights between neurons encode the network's knowledge, updated over a number of iterations via a learning algorithm. A number of learning algorithms and approaches were considered:

- Supervised

Supervised learning requires presenting the network with training data, as well as the expected output. The network's weights are shifted in small steps, until the output matches the desired output, or is deemed close enough. The

network's performance may then be validated against a further set of sample data, called the *validation set*.

- A multi-layer perception with back-propagation is the most common network architecture / training algorithm for supervised neural networks. The back-propagation algorithm is as follows:
-

```
while not CONTINUE
     CONTINUE = FALSE
     for each input data item do
          perform a forward pass of network to
          generate output
          compare output to expected output,
          to find an error value
          if error is too high to finish, set
          CONTINUE = TRUE
          perform a backward pass of error
          value, adjusting weights
     end for
end while
```

After a number of iterations, the network's weights should generate output close to the target output; the car's behaviour should match that demonstrated by a human operator driving a car manually during training.

- Unsupervised

Unsupervised learning provides the network with inputs, but doesn't supply it with the expected output. The system must organize the data as it sees fit. A measure of its performance may be provided by some kind of heuristic.

- A potentially powerful technique combines neural network technology with genetic algorithms (GAs). A GA is an abstraction of the natural process of evolution, the idea being to 'breed' and 'evolve' solutions rather than generate them straight off. New behaviours are created randomly, and successful ones maintained by a process mimicking survival of the fittest. The algorithm is as follows:

```
for each epoch
  create a generation of potential solutions
  for each solution
    apply a performance measure to gauge strength
    choose the best two solutions, call these parents
    destroy old generation
    combine parents by chopping at a selected position
    duplicate solution (offspring) until pop size is reached
    apply random mutation to each offspring
  end for
end for
```

The best weights for the network should be found via random mutation and passed down to the next generation – a good pair of networks will on average create a slightly better set of offspring than themselves; a process repeated until a suitably strong generation is created.

Implementation

The neural network / genetic algorithm (NNGA) module takes advantage of POP-11's supplied object-oriented programming extension. By representing the component parts as objects, management of the code has been made far easier. The diagram below demonstrates how the modules fit together:

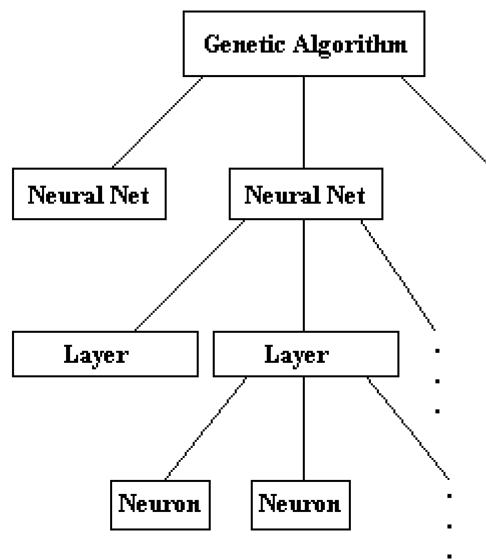


Illustration 1: Modular design of NNGA module

The Genetic Algorithm

The genetic algorithm (GA) class is a container for the whole AI. When initialized, it creates a prescribed number of neural networks (the *population*). These are multi-layer feed-forward networks with random weights. Each network is assigned to a vehicle in the simulation, and acts as its controller. The controller supplies each network with input values, and is returned two floating-point numbers to dictate the throttle level for each wheel.

After two vehicles have completed the track (or 2500 simulation cycles have completed), the GA's learning algorithm begins. The controller provides it with a score for each network, representing its performance on the track (see the *architecture* section). Only the best two networks are preserved, and these are simplified from objects to lists of numbers, each representing a network's weights. These two lists (*genomes*) are now manipulated to create the next generation of networks. Firstly, *crossover* occurs, where the two genomes are spliced at random positions and then combined. This merged genome is then replicated for the desired

size of the population. Next, *mutation* occurs. According to a small probability, a small change is made to the genomes' weights. This ensures that the new population is varied, with the potential of new and beneficial characteristics.

Because only the two best performing networks were chosen as parents, the new generation should perform (on average) better than the first. The mutation stage is essential; whilst it'll normally produce cars that perform badly, on occasion a mutation will prove beneficial, leading to a population with a helpful new character trait. This inheritance process can be witnessed in the simulation – if one parent spirals round the track, and the other drives in a straight line, the offspring will move in larger, more drawn out spirals that are a combination of both parents.

The Neural Network

The neural networks are designed as follows:

Inputs:

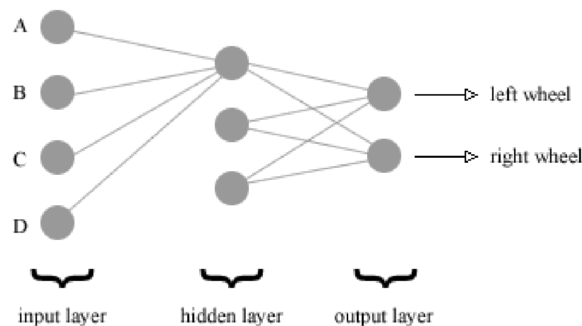


Illustration 2: Neural Network architecture. Some links removed for clarity.

Input Key:

A - Current velocity of left wheel
B - Current velocity of right wheel
C - Bearing / distance to next next waypoint
D - Bearing / distance to next next next waypoint

A fifth input was later added, to allow the agent to decide how close to each waypoint it aims. In addition, the neural network class was configured to make the number of hidden units and hidden layers fully adjustable.

The neural networks have no inherent learning ability; this is handled by the GA, above. Why, then, were neural networks used? For the GA to work, the agents' brains needed to be represented as a list of numbers that could be altered without completely losing the solution. The neural network is ideal for this; it allows the function to be split into many small component parts in a fault-tolerant fashion. This means that the neural network can still function even if the GA 'gets it wrong', and removes an important part of the computation. In addition, whilst smaller groups of neurons in a neural network won't produce the same output as the whole thing, they can still represent properties of the outcome – in this case, the tendency of a vehicle to spiral

or maintain high speeds. This means that crossing over neural nets can combine properties, producing better solutions. Furthermore, the neural network tasks an input space of many dimensions and reduces it to just two; ideal as a 'black box' solution for a large game.

Layer

A layer object stores neuron objects, and a collection of layers forms a neural network. It also performs the calculations for all of its contained neurons.

Neuron

The neuron class's function is mainly to hold the network's weights; integration calculations are performed by parent *layer* class (above). Each neuron object stores an array of weights, as well as its own output. This in turn will form the input to another neuron (unless this one is in the output layer). A diagram of the basic function of a neuron in the neural network paradigm is shown below.

The weights are multiplied by the inputs and summed. The bias is presented as a simulated input to simplify calculations, and saves needing a separate threshold value.

Possible Improvements

The *crossover* section of the genetic algorithm is simple, merely splicing two parent networks at random positions along their list of weights. In some cases this may be through the hidden layers, in others the input layer, etc. By standardizing the crossover procedure to operate more consistently, the rate of network improvement may be increased; especially for networks with multiple hidden layers, which currently don't yield very successful results.

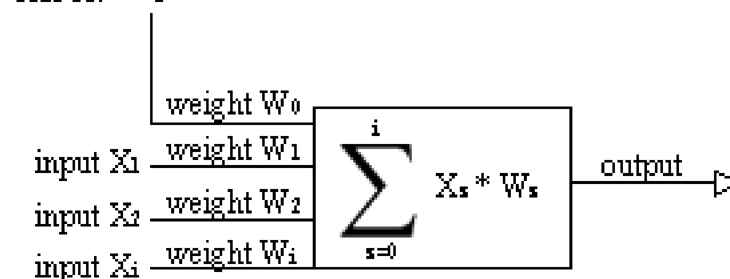


Illustration 3: Neuron diagram

At present two parents are chosen and their characteristics largely preserved. Whilst this reflects nature, there is no reasons why two parents is better than three, and experimenting with additional parents may create a greater variety of solutions at a greater rate.
An emerging field is that of modular neural networks. If networks were separately trained to handle turning left, turning right, acceleration, breaking, etc, it would be interesting to test whether any performance benefit would be gained above that of a single network.

Testing

Hypothesis: The genetic algorithm reliably improves network performance over time
Results: Repeated tests show only an increase in performance from the outset; the problem presented is when to cease training.

The following graph shows the best performance of the networks over a number of generations:

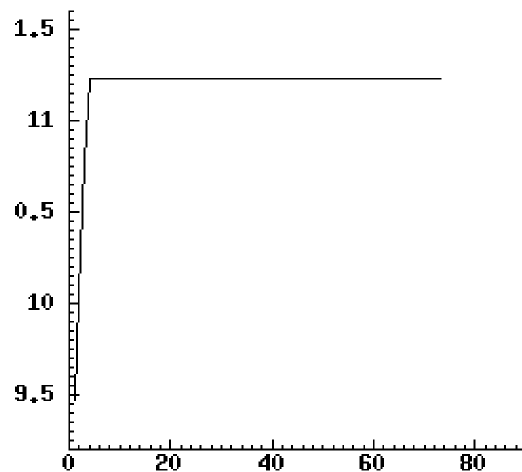


Chart 1: Performance over generations. x-axis: generation, y-axis: score

From this, it is tempting to stop training as soon as the performance level peaks and remains constant for a short number of generations. However, the following test demonstrates the danger in this approach:

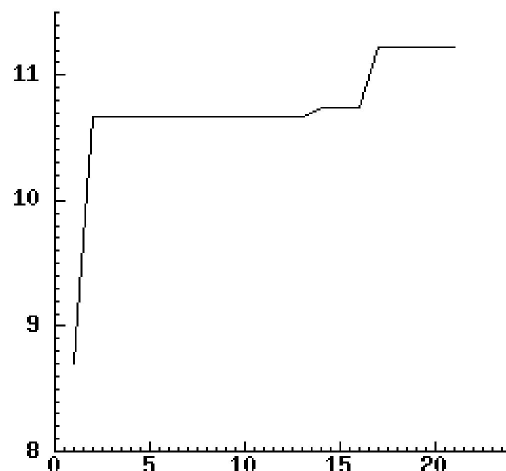


Chart 2: An unexpected rise in performance. x-axis: generation, y-axis: score

Despite over 10 generations of stability, performance still improves in a number steps before a new normal is found. This is mainly because of the random element in the genetic algorithm; there is a small chance of a large change to a network, and as such a small chance of finding an improved solution. When the population is already performing well this probability will be further reduced, accounting for the only

© Team popCorn

occasional jumps in solution quality. Furthermore, whilst new solutions may be generated that could become effective if allowed to evolve, they won't be given this opportunity unless they offer an immediate large improvement in performance; potentially beneficial improvements will regularly be deleted, reducing the rate of climb in the above graphs.

Conclusion: The GA does improve network performance over time, but not consistently. Training should only be stopped when a desired level of performance is reached, or according to a maximum time limit, to avoid choosing a solution caught in a local minimum.

It should also be noted that problem solving methods based on chance will not be suitable for all applications. Were it essential to find a specific solution to this problem, then a GA would be limited as it may not find the exact solution in reasonable time. It is appropriate to the racing simulation as there are a range of possible solutions, of which the GA is required to work towards the most efficient.

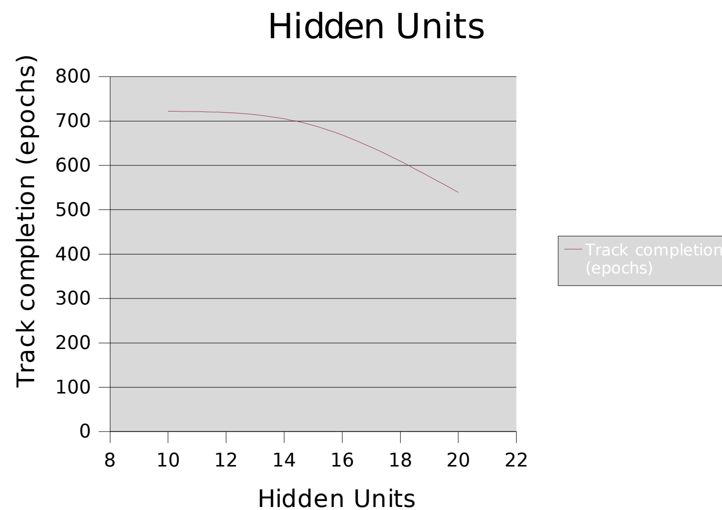
Hypothesis: The number of hidden units per network is inversely proportional to the time required for training

Background: The genetic algorithm module is able to 'breed' neural networks with any given number of hidden layers and neurons per layer. Intuitively, the more hidden neurons there are, the less relevance each neuron has to the network's final outcome. If true, this would suggest that the crossover stage of the genetic algorithm would reduce the quality of networks more easily, by loosing certain 'vital' neurons. To test this hypothesis, it's necessary to experiment with a range of network sizes, and compare the outcome.

Method: The track known as 'U-Track' will be used with the population test1. U-Track consists of two straight sections, joined at one end by a curve. The number of hidden units will begin at 20, and be reduced to 1 via steps of 5. For each test, the time taken for the best agent to navigate the track will be recorded after six epochs (generations). Each test will be conducted three times, and an average recorded.

Results:

Chart 3: Race time against hidden units

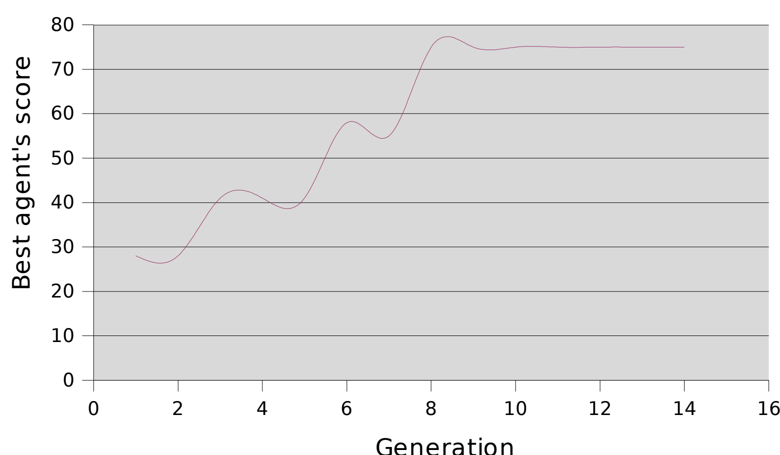


The number of hidden units has a strong correlation with the quality of solution found in a set period of time. Increasing the number of hidden units reduces the time taken proportionally, whereas reducing number leads to fewer vehicles completing the track, and requiring a greater period of time.

The test for 5 hidden units is not displayed above as no vehicles completed the track. Examining this test demonstrates the trade-off between learning rate and hidden units:

The simulation was allowed to run for 15 generations using neural networks with just five hidden units. They improved on their score on a number of discrete generations – 3, 6 and 8. After this the score remained at 75 until the 49th generation, when the simulation was terminated. Clearly we see a much slower rate of learning, without the initial rapid improvement observed in the other hypotheses.

Conclusion:
There are several reasons for the results observed. Firstly, a network with too few hidden neurons cannot represent a complex function, such as the *bézier curve*. It was surprising, therefore, when just five hidden neurons proved enough to navigate the track – albeit after a long period of training. This additional training time is also because a network with fewer neurons is more prone to failure; when the GA crosses over and varies weights, a larger network is less likely to loose trained features due to the increased level of redundancy and reduced proportional significance of each neuron.



Whilst the hypothesis is generally correct, it does have its limits; too few neurons and the application can take too long to find a viable solution. Too many, and the beneficial outcomes become overshadowed by memory and processing limitations.

Hypothesis: The agents' behaviour will differ depending on the shape and surface of the track.

Background: The learning system is designed to drive the simulated vehicles in such a way as to navigate the track in the shortest possible time. If they respond differently to changing environmental situations, then it's a good demonstration that the 'intelligence' is working towards an optimal solution. The physics model subjects the cars to similar driving requirements as found in the real world, meaning the cars should respond in an intuitive fashion: slow down before bends to avoid crashing, and speed up on straights to gain a time advantage.

In addition, the friction level of the road surface can be altered. Measuring the agents' response to these changes will demonstrate the relevance of the physics engine on the outcome, as well as test the networks' ability to generalize to new environments.

Method: The population (test1) has been trained with a friction level of 0.01; an approximate representation of the conditions on a race track. The friction will now be reduced to 0.001, requiring very different behaviour from the agents to stay on the track.

Results: Initially, all but 3 (of 50) vehicles left the track at speed, as they didn't know to decelerate in time for the bend. Only two of the remaining cars progressed towards the end of the track, and they did this in a laborious manner – spinning in large, overlapping circles. The second generation saw the majority of the population adopting this behaviour, and eventually the circles straightened out until a new successful strategy was adopted.

The solution found by the genetic algorithm is illustrated below.

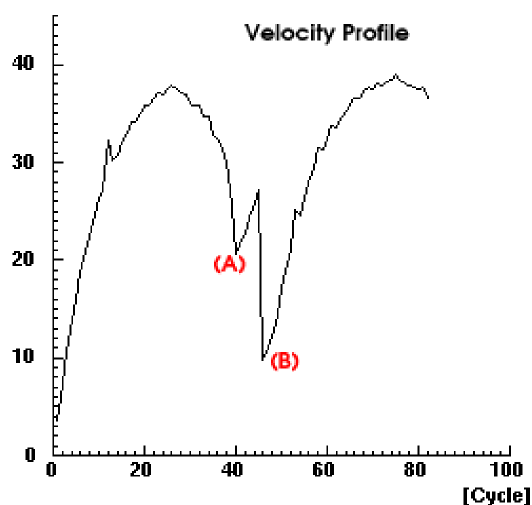


Illustration 4: Velocity Profile showing slingshot. Velocity is marked on the x-axis

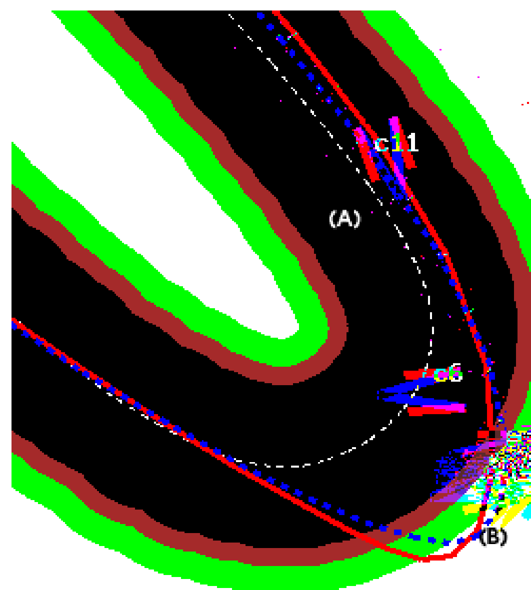


Illustration 5: Corresponding track screenshot

© Team popCorn

As the vehicles approached the region marked (A), they gradually reduced their velocity. A short burst of speed then followed, to power them through the bend. The throttle was very low until B when the velocity rose rapidly, in effect using a 'slingshot' technique to throw the vehicle into the next straight.

When compared to the velocity profile of a car experiencing normal friction levels, the difference can clearly be seen (see below). Here, the velocity drops rapidly when entering the bend, and rises equally rapidly on leaving it. There's no need for the prior increase in throttle employed in the low friction environment, as friction slows the vehicles sufficiently.

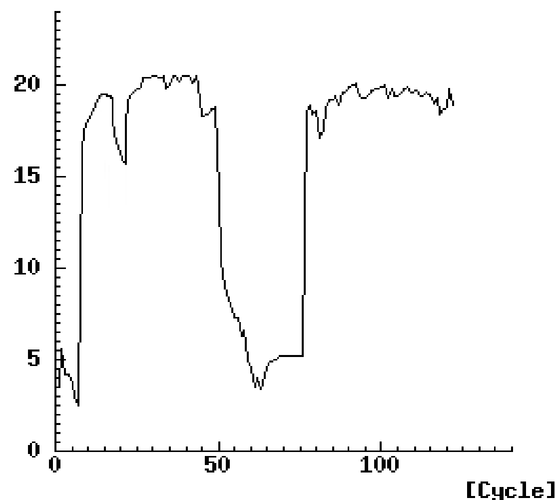


Illustration 6: The velocity profile of U-Track with friction co-efficient 0.01

Conclusion: Not only do the agents alter their behaviour according to the track's shape, there is a marked difference in behaviour as the friction co-efficient is altered. A pre-trained population was able to generalize to new environmental conditions in a short number of epochs, demonstrating the learning abilities of the genetic algorithm system.

III. The Architecture

Simulation Architecture

The PopRacer simulation is made up of several modules, which are powered by a set of procedures that makes up the 'Simulation Engine'.

This modular approach has enabled us to separate the development of application to different team members.

Object Orientated Design

© Team popCorn

The simulation has been implemented by utilising the object orientated design facilities provided by Pop-11's libraries (objectclass & rc_lib), which has aided in constructing the world and the intelligent agents.
The cars in the simulation are stored and represented by 'car_object' objects. Each object holds data and a graphical representation of itself, which are stored in 'slots' within each instance of the 'car_object'.

The car_objects also 'inherit' the ability to be drawn and moved/rotated on the screen by inheriting the property 'is rc_rotatable', which simply specifies that the object can be passed to drawing procedures, moved and also rotated using the inbuilt procedures provided by Pop-11 and its 'rc_lib' set of libraries.

The structure of the 'car_object' is outlined below:

Slot	**Use**
rc_picx	Current Location (X)
rc_picy	Current Location (Y)
rc_axis	Current Orientation (In Degrees)
velocityX	Current Velocity of Car (X)
velocityY	Current Velocity of Car (Y)
alpha	Current Bearing of Car
Mass	The actual mass of the car
waypointQueue	A List of Waypoints in the form [X Y]
reachedPoint	Boolean, true when the car has finished course.
throttleLeft	Left Throttle Value
throttleRight	Right Throttle Value
bonus	Bonus score for the car, the car gets a bonus each time it reaches a waypoint.
iterations	Number of iterations of the simulation it has taken for the car to get to a waypoint (reset to 0 when reaches a point).

The car_objects are then stored inside a list called 'cars', which makes drawing of the all the cars very convenient and easy.

These object-orientated facilities are also exploited in the Neural Network/Genetic Algorithm module of the simulation, where a single object called 'ga' stores several neural network objects, which powers each one of the cars.

It is implemented so that the car_object at location X in the 'cars' list, has its neural network located at X within the 'ga' object.

Goal-Orientated Intelligent Agent

The cars are designed to be primarily goal-orientated agents, in which they decide how best to reach their goals either through the use of a Neural Network/Genetic Algorithm or using a Rule Based Engine.
The car tries to reach each of its goals by modifying its left and right throttles for each of its wheels.

The cars goals are to reach a set of waypoints, which are specified by the track in the form [<X Coordinate> <Y Coordinate>].
Each car has its set of goals stored within a 'waypointQueue' slot, whereby once it reaches the waypoint at the front of the queue it is removed and the car moves onto the next waypoint in the queue.
For the Neural Network/Genetic Algorithm to function we have implemented a method in which we can order the cars according to their performance.
This has been implemented as follows:

- Each car has an 'iterations' slot and 'bonus' slot.
- Iterations stores the number of cycles of the simulation it has taken for a car to reach its current waypoint. This number is reset each time the car reaches a waypoint.
- Bonus stores a value, which is incremented by 1000 each time a car reaches a waypoint.
- When a cars waypointQueue is empty, it starts to accumulate an extra bonus of 2000 per cycle of the simulation until another car finishes the track.

When at least two cars have finished the track, the simulation is stopped and the population of cars are evolved using the Neural Network/Genetic Algorithm module where the cars chosen for breeding by the Genetic Algorithm are the two with the lowest fitness value. The actual fitness value is calculated as follows:

Fitness = Iterations - Bonus

This new population is then run on the track, this then continues for a set number of generations.

Overview of Simulation Engine

The 'Simulation Engine' is a set of procedures that integrates the separate modules of the overall application; the neural network/genetic algorithm, physics model and graphics system.

The 'Simulation Engine' is made up of several procedures:

- Simulation Loop Procedure
- Car Sensor Procedures
- Agent and World Creation/Maintenance Procedures
- Graphing/Statistics Procedures

Simulation Loop Procedure

This procedure is simply several nested loops, which run the simulation by calling the different modules of the application. Each module can be made up of one or more procedures.

The 'Simulation Loop' itself is designed as follows:

Pseudo Code:

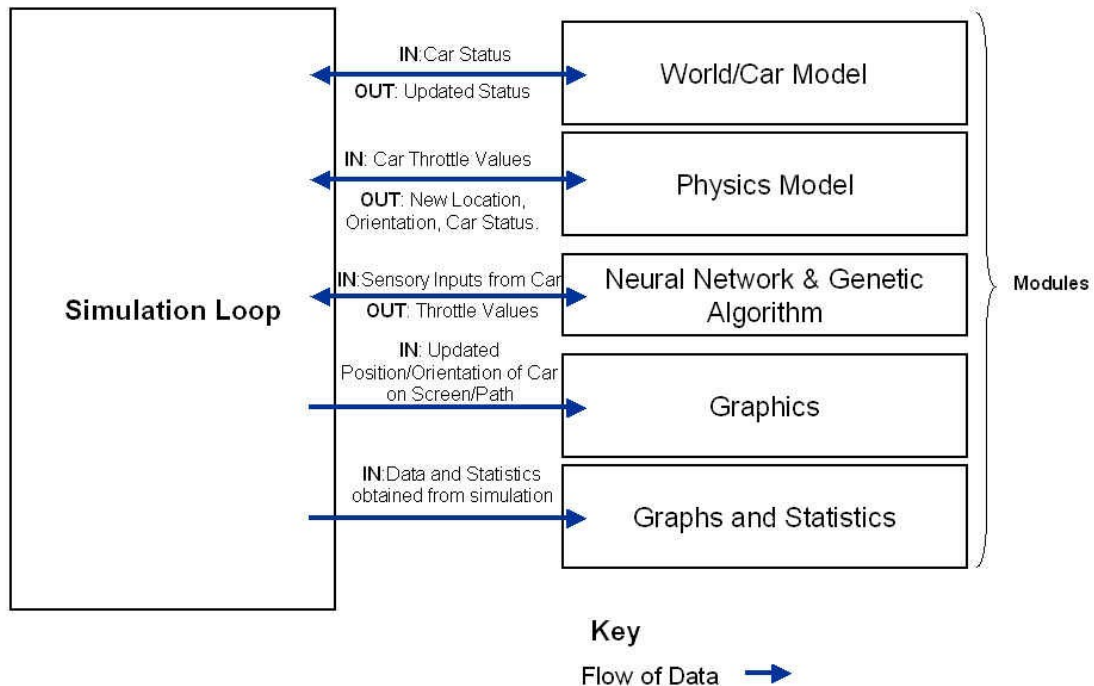
- For g generations
    - Get Neural Networks from ga object.
    - For c cycles

        - For each car in the World
            - Get current status and sensor readings
            - Pass status and sensor readings into N.N & G.A/Rule Based Engine.
            - Pass N.N & G.A generated Throttle Values to Physics Model to determine new location and orientation.
            - Check current waypoint distance, and update bonus and waypointQueue accordingly.
            - Update Car Status
            - Draw Car in new position/orientation.
        - End for car

        - Calculate Statistics
        - Draw/Update Graphs
        - Clear World/Cars
        - Evolve population
        - Create new Cars
    - End for c

End for g

Below is a diagram outlining the flows of data between these modules and the 'Simulation Loop' itself.



Each module shown in the above diagram is outlined below, with an in-depth explanation found in the relevant section of this report.

World/Car Model

This module creates and manages the track and the cars in the world.
It is made up of several procedures, which:

- Create new cars
- Remove Cars
- Setup tracks
- Determine whether a car is on the track or not.

Physics Model

This module is contained within a separate file (physicsEngine.p), which implements a Newtonian physics model.
Throttle values are passed into the model and the new location and orientation of the car is returned.

The physics model has the ability to assign different friction coefficients to the cars depending on whether a car is on the track, on the edge of the track or way off the track.
This feature is made possible by generating what we term 'Friction Circles'.

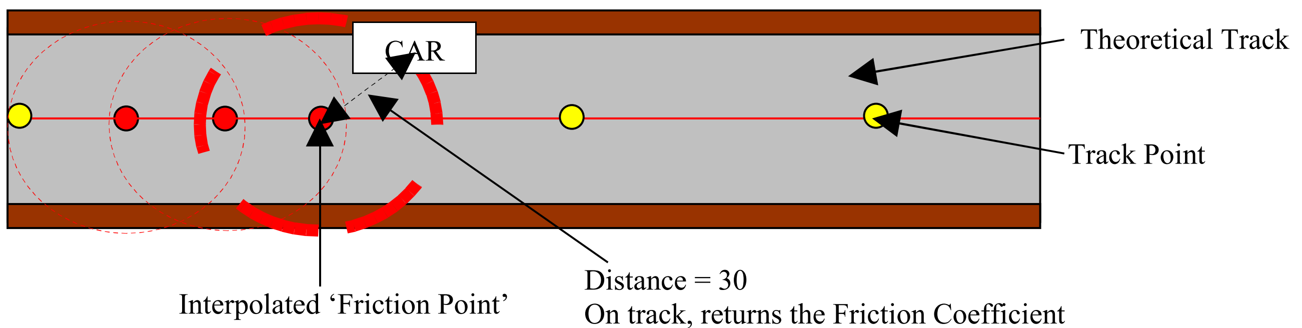
Friction Circles

© Team popCorn

Friction Circles are simply points along a track, however there are many more of these points than there are in the track points list.
'Friction Points' are generated by interpolating along the track points list, only if the distance between one point and another is greater than a certain distance.
It is then a simple process to determine if a car is on track by:

- Finding the closest 'Friction Point' to the car.
- If the distance between the point and car is >80 then
    - Return 50% of the friction coefficient and force the car to crash.
- If the distance between the point and car is >70 and <=80 then
    - Return 250% of the friction coefficient.
- If the distance between the point and car is >45 and <= 70 then
    - Return 200% of the friction coefficient ELSE return friction coefficient

Example Diagram



When these distances are applied it turns the 'Friction Points' into a 'Friction Circles', where the distance is the radius of the track or radius to the edge of the track.
By interpolating it causes the 'overlapping' of these 'circles' which enables us to determine fairly accurately if the car is on the track or not.

Neural Network & Genetic Algorithm

This module is contained within a separate file (nnga.p), which both creates and manages populations of neural networks.
It implements the object-orientated design facilities of Pop-11, which enabled us to give the user the ability to specify the structure of the Neural Networks used in the population easily.

Graphics

This module is implemented using the RCLIB library provided by Pop-11, these libraries have been utilised to draw the track/world and the cars/agents.

A key object orientated feature of Pop-11 that is used for implementing the graphical side of the simulation is called 'rc_linepic', this feature enabled us to specify the graphical representation of the cars within a single 'slot' within each instance of the car_object in the simulation.
This has enabled us to draw any particular 'car_object' by writing 'rc_draw_linepic(car_object)', which draws the object at a location and orientation specified within itself.

The track itself is drawn by overlapping circles of different colours, a white dotted line is then drawn following the track's points.

Graphs and Statistics

This module produces the statistics and graphs for the simulation.
It generates graphs for:

- Average Fitness Value of Population for each Generation.
- Best Fitness Value of Population for each Generation.
- The current velocity of the best car in the current Generation.

It also prints out these values to the console.
This module enables us to determine the performance of the genetic algorithm and the neural networks.
The current velocity graph is of particular use in proving that the cars are ‘intelligently’ slowing down for bends and then accelerating for straights.

Car Sensor Procedures

- Distance
  Calculates the distance from one point to another in the world.

- Bearing
  Calculates the bearing from the car to a waypoint in the world.

Agent and World Creation/Maintenance Procedures

These procedures ensure that the world and the cars/agents are initialised correctly and updated accordingly as the simulation is run.

Load and Save Features

Pop-11 provides some very useful in-built procedures, which has enabled us to save and load populations of cars and tracks.

Genetic Algorithm Populations

The G.A population is encapsulated into a single object, which can be saved and loaded from disk.

Tracks

The simulation has several predefined tracks already hard-coded, as well as having an in-built ‘Track Editor’ so the user can design and test cars out on different tracks.

© Team popCorn

This feature would not be of any practical use without the ability to save and load previously designed tracks, so we added this facility.

The tracks are not objects, but have three components, which are then saved into a single file.

It is structured as follows:

- trackPoints : A list of points representing the track.
- trackOffsetX: The offset of the track in the global X-axis, so the track is drawn in relation to the first point in the trackPoints list.
- TrackOffsetY: The offset of the track in the global Y-axis, so the track is drawn in relation to the first point in the trackPoints list.

Loading tracks is also very straightforward, where the values from the file are copied into variables.

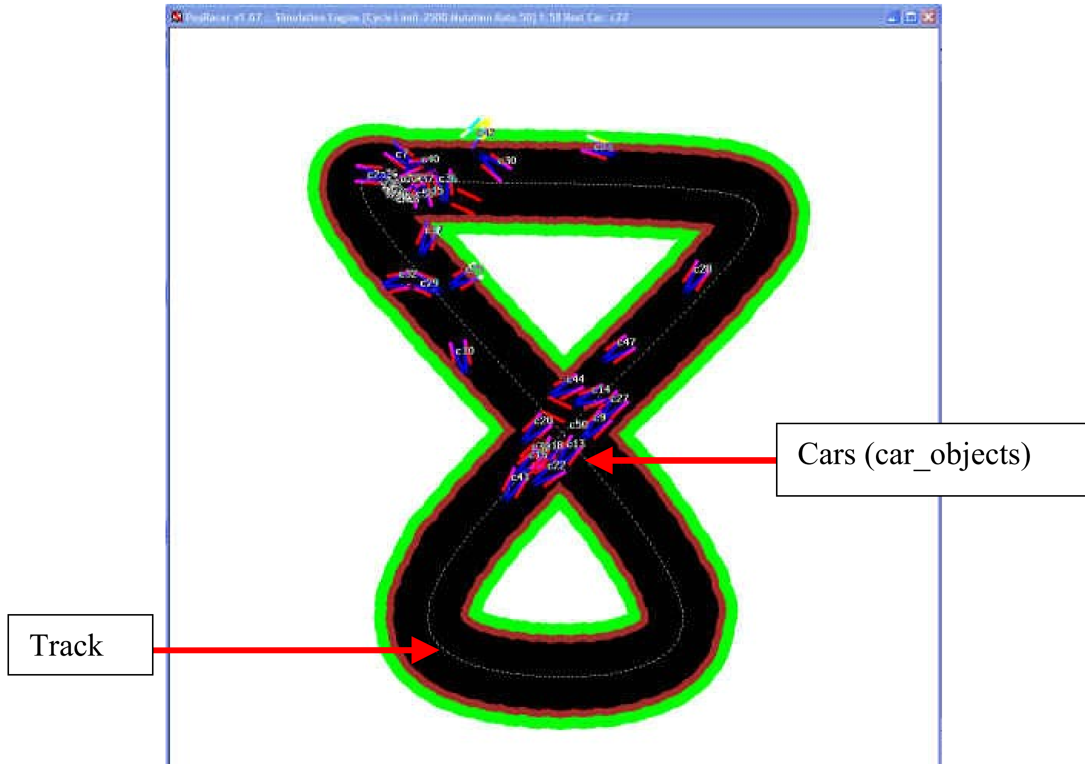
The track data and genetic algorithm populations are also saved along with a 'tag', which ensures that incorrect data cannot be loaded into the simulation.
This tag is simply a list added to the end of the file either '[popracer track]' for a track or '[popracer population]' for a genetic algorithm population.

The files loaded are also checked to ensure they have the correct number of fields. This feature was added near the end of development, and so it was necessary to enable old format files to be loaded in. When the user does this they are informed that the file format is different and advised to resave either the track or genetic algorithm population.

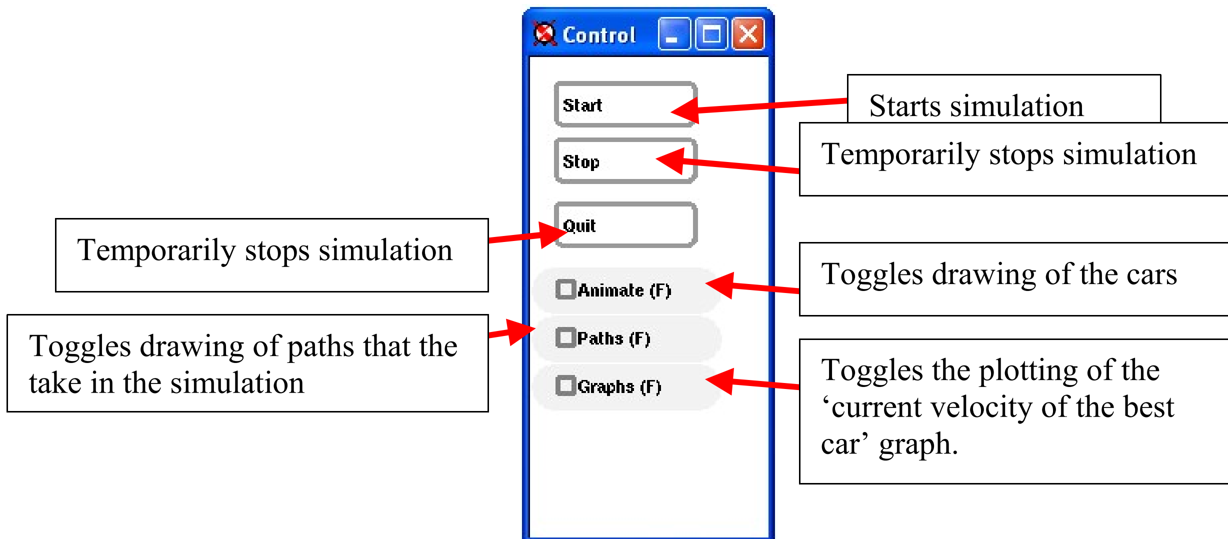
Graphical User Interface (GUI)

The graphical user interface is made possible by using Pop-11's inbuilt libraries.
The interface is very simplistic with it being made up of three graphing windows, a control panel and a window showing the world.

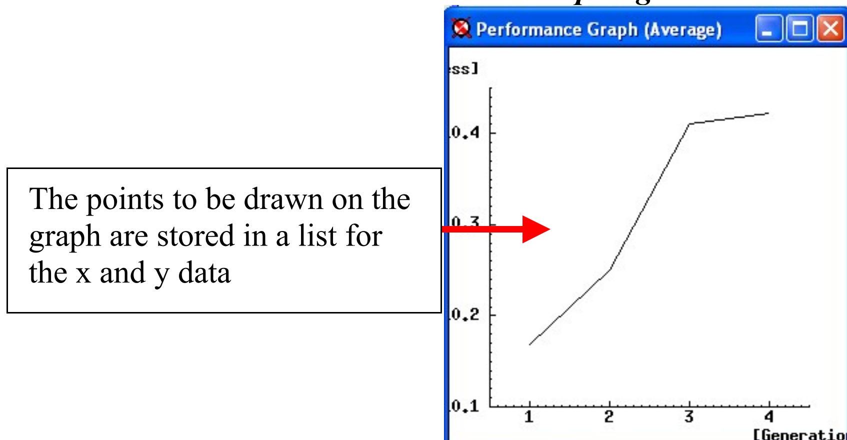
Main Simulation Window



Control Window



Graphing Window



Command Line Interface (CLI)

When the simulation is run the user is presented with a simple command line interface, which enables them to customise the simulation as they see fit. The command line interface is powered by Pop-11's excellent pattern matching procedures to enable commands to take arguments.

The user is given the following commands:

- setcars <number of cars>
  Choose the number of cars to be simulated
- loadcars <filename>
  Load a previously-saved population
- savecars <filename>
  Save a the currently loaded population of cars.
- settrack <name of track>
  Choose a previously-saved track
- setcycles <number of cycles>
  Number of cycles of simulation given to the cars as a target to beat
- setquicktrain <number of cycles>
  Cars are automatically animated once they can complete a circuit in this many cycles
- setmutation <mutation rate>
  Mutation rate of the population in the genetic algorithm
- sethidden <number of hidden units>
  Sets the number of Hidden Units in each layer in the Neural Networks
- setlayers <number of layers>
  Sets the number of layers in the Neural Networks
- waypointai <0/1>
  Switches on (1) or off (0), the ability of the cars to determine when they have intercepted a waypoint
- parameters
  Display details for the current population
- createtrack
  Launch the track editor to create a new track
- savetrack <file name>
  Saves the track in the track editor to the file specified
- loadtrack <file name>
  Loads a track from the file specified

© Team popCorn

- help
  Display help documentation

- quit
  Cleanly exit the simulation

A command line is also made available when the user clicks 'Stop', which enables them to then save the currently running car population into a file.
'Load', 'save' and 'quit' commands are provided through this command line.

IV. The Physics

Literature Review

As a project with so much scope, a 'racing simulator' provided us with a very broad array of problems, and, as a result, possible solutions. As part of the idea behind the project was to make something very extensible, we drew upon ideas from quite a large selection of sources; some complex models – such as the physics, and some being more rudimentary ideas that we arrived at our own unique solutions to.

We decided early on, that we would be able to separate the project into 3 core components. As a racing simulator showcasing an AI, we arrived at the idea that the problem was separable into an 'Engine', responsible for the physics modelling, world and object handling; an 'Intelligence', capable of outputting desired actions to the Engine, and, importantly, an 'Interface', responsible for accepting user input and graphically displaying the state of the physics engine.
There was some discussion at the start of the project, that we might include a natural language processing element to the interface. Initially, this was suggested because of the ease of which it can be accomplished using Pop11's built-in pattern matching functionality, however, we dismissed the idea as adding yet more scope to an already complex problem. We later reviewed this, given difficulties with some of Pop11's 'quirks', and decided it would be prudent to fall back on some console input.

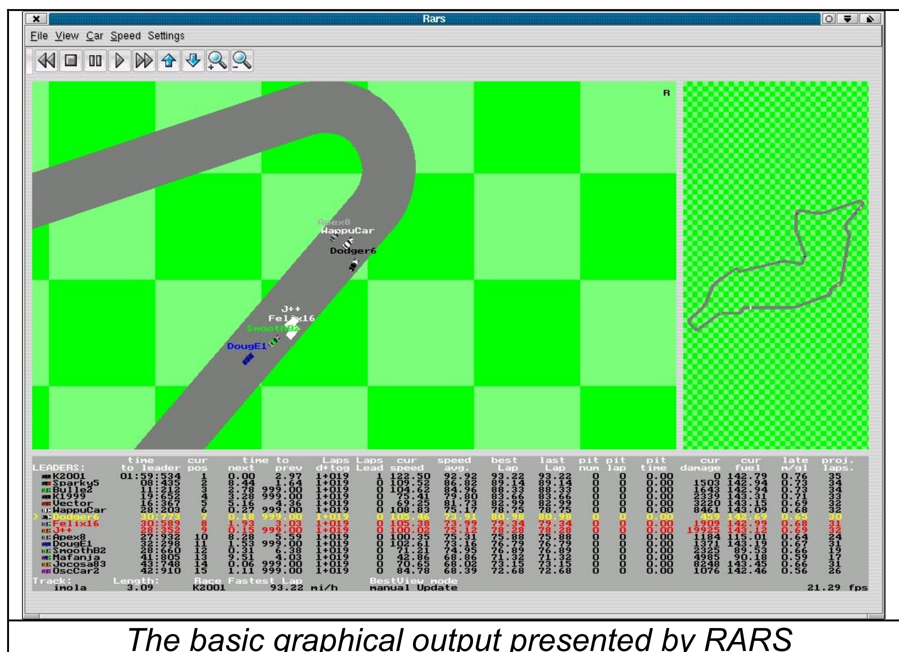
Whether or not it was just our good intuition that caused us to separate the project into 3 modules, we found through research that we were not the only people to decide on this solution. In fact, the most promising example we could find was a project with a very similar theme to our own – a racing simulator with an AI driven component.

RARS

The 'Robot Auto Racing Simulator', or RARS, was the most prominent example of an AI-driven Racing simulator we could find. Having originated in

1997, with continued development to this day, the project has an immense maturity that our own brief project would obviously have been incapable of matching. The breadth of work that had been done on RARS was quite impressive, and much of its functionality would have been difficult, if not impossible, to match using Pop11. As such we examined RARS with an eye for features that we might be able to include in our own project – and solutions to possible problems that we had not yet encountered.

From the brief overview given on the RARS website, we found that the physics of RARS approximates the real world very loosely. The most important consideration of the simulation was apparently to be ‘good enough’ to approximate real driving, but not to model it with absolute accuracy. The system in RARS, as a result, only makes use of 2 dimensions. By neglecting a z axis for depth, the implementation is greatly simplified, and so a lot of phenomena seen in the real world would be impossible to duplicate. The simulation also, we noticed, involved an ‘Alpha’ component – used to describe the offset of each car object’s velocity vector from their pointing vector. While not entirely realistic, it was noteworthy that the simplification did not have a significant impact on the final ‘physics’ product. Of course, a completely accurate simulation would be impossible on modern computing hardware in real time, and so the balance of complexity used in RARS was very appealing to us. Even if the computation power available had been limitless, the task of building a 3D physics model was extremely daunting – and our research into the area provided very little return.



The basic graphical output presented by RARS

As may be seen above, the graphical output is 2d, showing a top-down view of the racing cars, along with a selection of statistics, calculated for each car.

One of the core design decisions with RARS, which we were interested in, was the modularity of the AI agents. From the ground up, RARS had been

intended to showcase various different AI techniques, and compare them with each other through simulation. As such, at each simulation step, every ‘plug-in’ in RARS is capable of querying the physics engine for certain information, and returning a desired target value for its own speed and turn angle, etc, with the physics being responsible for calculating the required amendments to the world in order to make this a possibility. While this seemed a workable system, the concept of a ‘target’ that had to be calculated by the physics engine seemed in many ways unnatural to us, and something that should be the responsibility of the agents themselves.

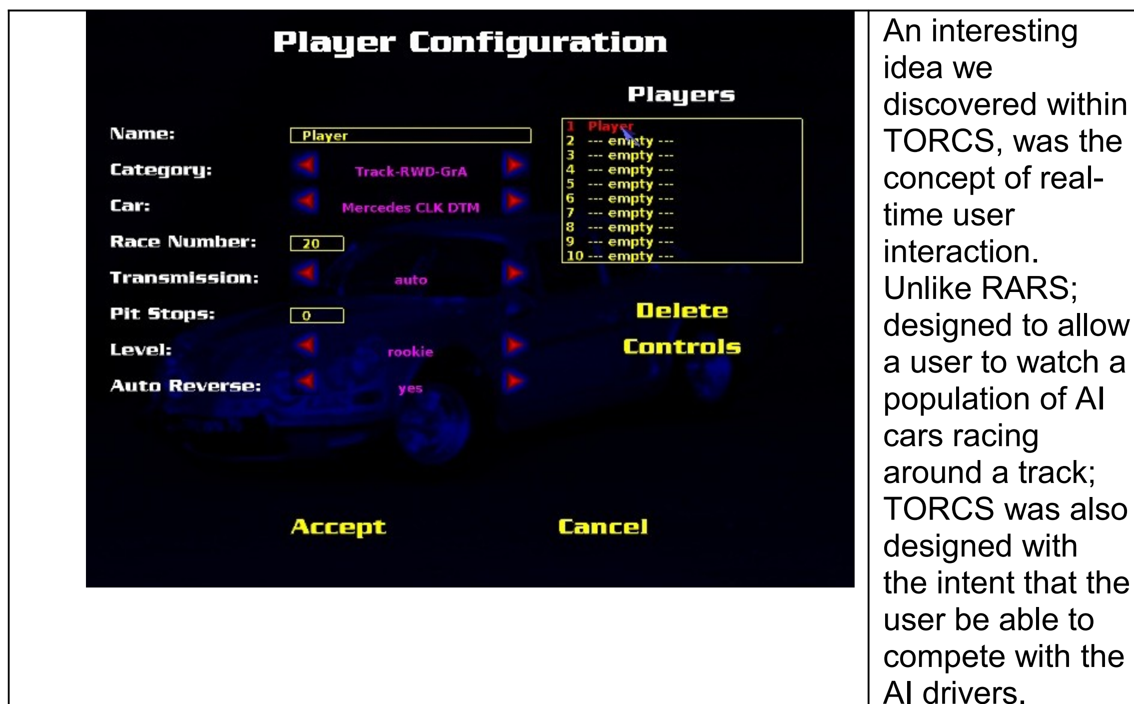
TORCS

When searching for further solutions, it did not take long to find another project very similar to RARS. Upon examination of TORCS, ‘The Open Racing Simulator’, it is obvious that RARS has been its inspiration. As the project FAQ states; “The goal is to have programmed robots drivers racing against each others”, a goal not dissimilar from our own.
Examination of the structure of TORCS was somewhat difficult, as it is not nearly as well documented as RARS. Essentially the physics engine employed is very similar to the one in RARS, though with a few additions to make it slightly more complex. Rigid body collision detection is in place, with car objects defined as models constructed of polygons. We also found that a damage system was in place, along with some modelling of aerodynamics. Whilst these additions made the simulation more realistic in a sense, the programming effort required to replicate them would have been quite formidable. The way in which the track object is expressed in TORCS was also quite unusual; being constructed of a series of ‘straight’ or ‘turn’ segments, such that the track is built up sequentially.

As with RARS, TORCS continues the idea of an AI ‘plug-in’ to control car objects, and much of the design is focused towards providing a platform to test these ‘robots’ against each other.

A highly polished 3d output in TORCS, using OpenGL

As can be seen in the screenshot above, TORCS is capable of rendering the world state in a complex 3d engine, powered by OpenGL. Whilst this was clearly unfeasible to implement in Pop11, if not impossible; the interface between the graphics engine and world state were still very interesting. The design was such that the graphics engine was a separate module that could 'hook' into the physics, ensuring it could be easily replaced with another alternative display system. Both the 3d graphics and physics systems in the illustration above can make use of the same 3d models for collision detection purposes.



An interesting idea we discovered within TORCS, was the concept of real-time user interaction. Unlike RARS; designed to allow a user to watch a population of AI cars racing around a track; TORCS was also designed with the intent that the user be able to compete with the AI drivers.

As a result of this, the project has an intuitive menu system built-in; similar to one that might be found in a computer game; for configuring various aspects of the simulation.

Upon further examination we found that TORCS had even more user-interaction features. Whilst the core idea of TORCS was based heavily on RARS, it also goes further – in an attempt to make the simulation into a game. Split-screen functionality also exists, so that up to 4 people may compete against each other and the AI agents. Despite the fact that the graphics engine in TORCS was so complex, there was no necessity for this to be the case. The basic idea of creating a game out of the base physics engine and AI, illustrated in TORCS, was especially interesting to us, as it added a new dimension to the project that we thought we might replicate.

'The Physics of Racing Series'

As a core component of the project, the physics were high on the list of priorities. Through our research of the area, we found the 'Physics of Racing Series' by Brian Beckman; a set of 29 articles about the physics of racing

cars. In his own words, “I start with the fundamentals (Newton's Laws, for example) and am slowly but surely building up complexity and covering more advanced topics.”
The incremental approach to the physics was very useful – beginning with the presentation of very basic concepts, and gradually layering on complexity – going into explanations of such things as combination slip and combination grip. There was a limit to the usefulness of all this, however, in that the model Brian Beckman presents is still incomplete. Also notable, the underlying mathematics of many physical aspects is very complex, and with no clear strategy presented of how to link everything together, it would have been a formidable task to attempt to do so; well beyond the scope of our project. Nonetheless, many of the earlier articles covered the basic details necessary for a rudimentary simulation.

Bibliography

Beckman, B. (1991-) ‘The Physics of Racing Series’ (http://www.miata.net/sport/Physics/index.html; email: brianbec@microsoft.com).

Pop-Racer Physics Model

1. Introduction to the problem: An agent without any constraints.
2. A real world solution.
3. What is a physics model?
4. How Newtonian Physics model the physical factors
5. The structure of the model
6. Progress made

The Problem: *An agent without any constraints*

The starting point is to identify the potential problem, namely that any agent requires the imposition of constraints if it is to operate in a non-random fashion (as opposed to an absolutely pre-determined fashion). Essentially, the problem is concerned with preventing the agent performing unrealistically such that it has no relation to the user’s knowledge and perceptions of the physical world. Having some level of constraint also creates the opportunity to establish benchmarks against which the performance of the agent can be tested and monitored.

A real world case

Observing the motion of a real-world agent it becomes apparent that there are real-world constraints that affect the agent’s path. This section will present and discuss some of these real-life constraints.

- Mass

The mass of an object can be defined in simple terms as the amount of matter it contains. When we want to change the velocity of an agent the magnitude of the force we have to exert on the body is proportional to the mass of the body. In other words to move a heavier object we need to exert a bigger force than a force needed to move a lighter object. The following diagram clarifies it further.

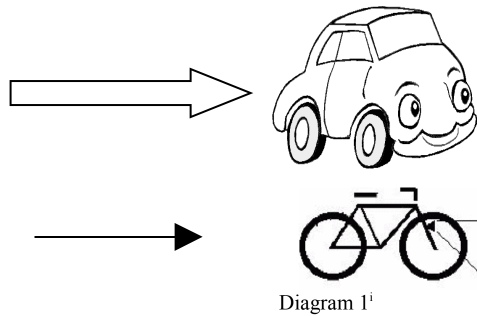
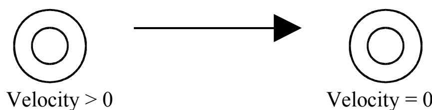


Diagram 1[i]

It is common knowledge that moving a car is harder, i.e. requires a greater force, than it is to move a bicycle. At the same time the car has a bigger mass than a bicycle.

- Friction

"In physics, friction is the resistive force that occurs when two surfaces travel along each other when forced together. It causes physical deformation and heat buildup."[1] It is friction that causes a free rolling ball to slow down and eventually stop. Since it is a resistive force it always acts in the opposite direction to the motion of the agent and so it is harder to move an object on a surface as the friction increases.



- Centripetal and Centrifugal Forces

Consider an agent moving in a circular motion then:

Centripetal force is the term given to the force component that pulls the agent towards the curve.

Centrifugal force is the force that pushes the agent away from the curve; (A later section will explain in more detail how these forces come about).

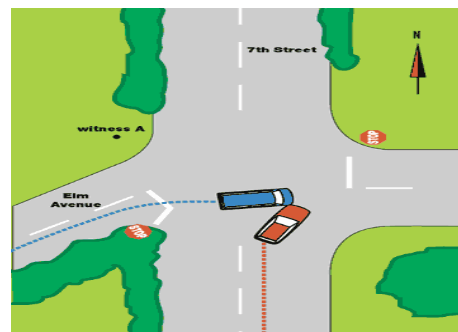
However it is self-evident that in order for an agent to move in a circular path there should be forces acting towards and away from the centre of the curve. In order to implement real-world circular movement in our programme then it is only logical to use Centripetal and Centrifugal forces as a model.

1

- Momentum

It is hard to give a simple, abbreviated explanation of the term momentum relating to a moving object with scientific accuracy. It can be crudely put as a quantity that is directly related to the mass and the velocity of an object. The principle of momentum is a useful tool in explaining the behaviour of objects that collide.

Since the intention of this project is to have several agents moving about in the same track momentum can be used to implement collisions of the agent.



Collision of two cars[ii]

Wikipedia
http://en.wikipedia.org/wiki/Friction

Although the summary above has outlined some physical factors that might affect a real-world agent, in reality these factors do not act independently of each other. On the contrary, what are normally seen as physical constraints on an agent are actually the results of co-dependent factors interacting with each other.

What is a Physics model?

"A model is a description of observed behaviour, simplified by ignoring certain details. Models allow complex systems to be understood and their behaviour predicted within the scope of the model, but may give incorrect descriptions and predictions for situations outside the realm of their intended use. A model may be used as the basis for simulation"[2]

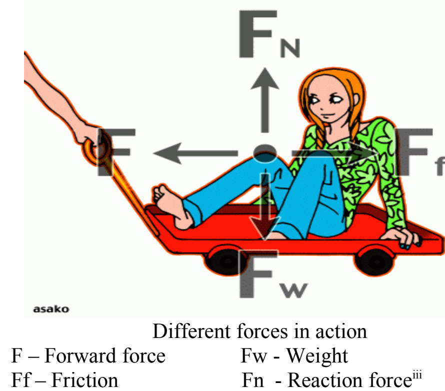
In the previous section some of the physical factors that could affect the motion of an object in the real world were discussed. If the POP-racer project is to be made as realistic as possible it is necessary to produce a model that implements these physical factors into the agent's movement. This is what is meant by a Physics model. The credibility of such a model entirely depends on how well each individual factor can be applied and the possible effects they might have on each other.

Since the project is dealing with small agents that simulate real-world objects it was intuitive to base the physics model on one that is used to explain real-world motion. Hence it seems appropriate to use the simplest but highly effective Newtonian physics in our physics model.

The next section will discuss how Newtonian physics model the previously mentioned physical factors.

[2] Dictionary.com
http://dictionary.reference.com/search?q=model

- **How Newtonian physics could model the physical factors.**



Different forces in action
F – Forward force Fw - Weight
Ff – Friction Fn - Reaction force[iii]

- Relation between the mass of an object and its acceleration.

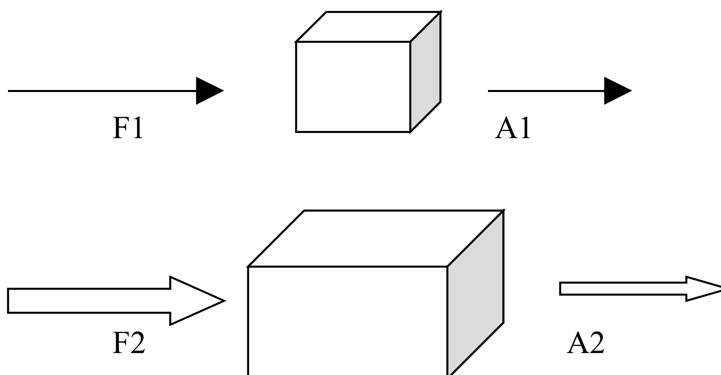
It is the common observation that the mass of an object is directly proportional to the force exerted on the object. Force is also directly proportional to the change of velocity or the acceleration of the agent.

$$F \propto m$$
$$F \propto a$$
$$\Rightarrow \quad F \propto m * a$$

But rather conveniently by Newton's second law of motion we derive the equation

$$F = m * a$$

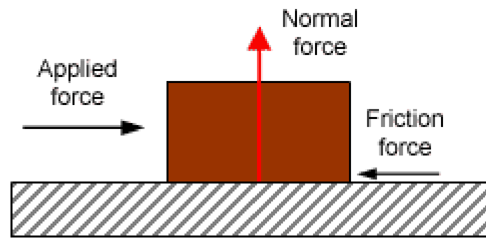
It is worth noting that the direction of the force and the direction of the acceleration are always the same.



Since M2 > M1 and A2 > A1 therefore F1 > F2

- The Friction

The friction force resists the relative motion or tendency for such motion by two surfaces in contact. In Newtonian physics the friction model depends on two values.

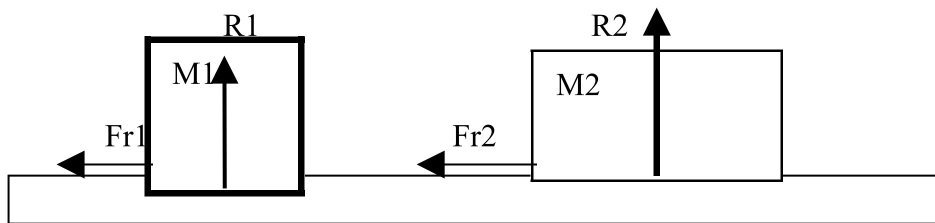


Friction diagram
Reaction force and the normal force are the same[iv]

1. Reaction force exerted by the surface on the object.

According to Newton's 3rd law of motion every action has an equal and opposite reaction. The weight of an object on a surface pushes that surface away. Therefore the surface exerts an equal force on the object upwards. This force is known as the Reaction. It has an equal magnitude to the weight.

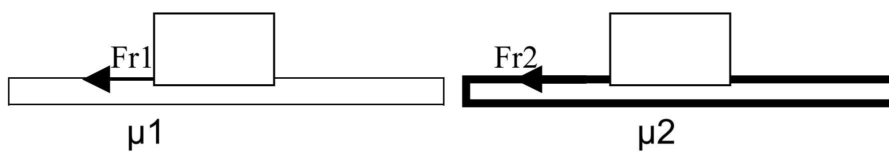
R = W = m * g ;;; g is the gravitational acceleration = 9.8



R2 > R1 Fr2 > Fr1

2. Coefficient of Friction

The second factor that affects the friction of a surface is known as the Coefficient of Friction. This value corresponds to the roughness of the surface. It is a value that lies between 0 and 1 and increases as the roughness of a surface increases.



If μ2 > μ1 then Fr2 > Fr1

Intuitively we know that the resistance against the movement of an object is increased either by increasing the weight of the agent or the roughness of the surface upon which the objects are traveling. These observations lead us to:

$Fr \propto R$
$Fr \propto m * g$
$Fr \propto \mu$

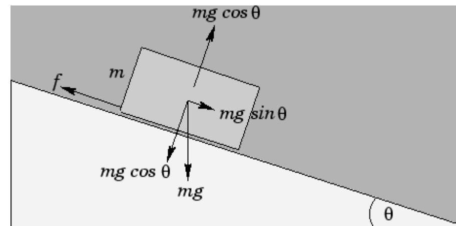
Therefore we model friction with the following equation:

$Fr = \mu * m * g$

The assumption is that the surface is flat. Otherwise the equation needs to be modified:

$$Fr = \mu * m * g * \cos A$$

Where angle A is the inclination of the surface. In this case only the cosine component of the Reaction force affects the friction.



Friction on a slope[v]

- Centripetal and Centrifugal force

Centripetal and Centrifugal forces are very important to the circular motion section of the model. It enables the model to simulate the phenomenon of skidding when an agent is driving around corners.

Centripetal force:

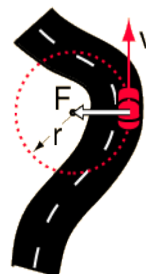
Newton's first law of motion states a moving body travels along a straight path with constant velocity unless there is an external force exerting on the object. For circular motion to occur there must be constant force acting on a body pushing it towards the centre of the circular path. This force is the centripetal or centre seeking force. So according to the Newton's 1st law the object moving must change its velocity. In other words the object must accelerate and since the centripetal force, which causes this change in velocity, is acting towards the centre, the direction of this acceleration must also be towards the centre of the curve. It is quite appropriate to call this the Centripetal Acceleration. The size of this acceleration is:

Ac = (V^2) / r

V is the linear velocity of the agent and the r is the distance to the centre of the curve.

$$F_{centripetal} = m\frac{v^2}{r}$$

$\frac{v^2}{r}$ is the centripetal acceleration



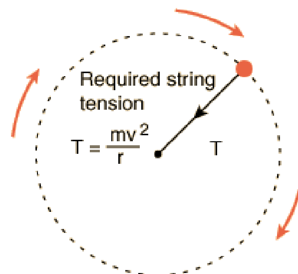
Centripetal force diagram 1[vi]

Since F = m * a

By substitution we can calculate that Centripetal Force is:

Fc = (m * (V^2)) / r

Centripetal force is just a name given to any force that causes an agent to move in a circular path. For example in the planetary motion Centripetal force is the gravitational pull whereas when an agent is moving in a curved path it is friction of the surface or a component of it that becomes the Centripetal force.



In the diagram centripetal force is provided by the tension of the string vii

The whole phenomenon can be summarised as "… *the changing direction of the velocity tells you that the agent is undergoing an acceleration, which must be caused by a force. The force that causes the acceleration is the frictional force acting towards the centre."*[3]

From the above equation we can derive two conditions:

1. Fc α 1 / r

Centripetal force is inversely proportional to the distance between the centre of the curve and the centre of the agent. Since the Centripetal Force in this case is the Friction the greater the friction the closer the agent is going to travel to the centre.

2. V α r

Linear Velocity is directly proportional to the radius. In other words the faster the agent travels on a curved path the further it should move away from the centre of the curve.

So it is these two conditions that are quite important when we model the circular motion.

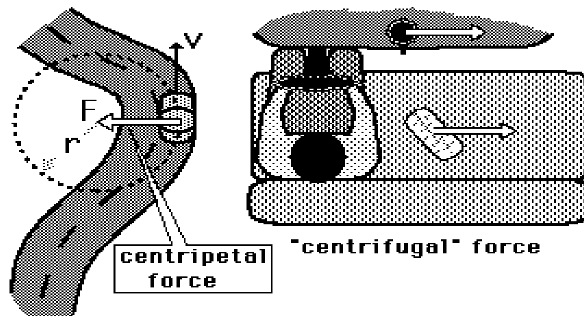
Finally it is also worth mentioning that the agent could also change the velocity by changing the magnitude of the velocity. In this scenario the Friction force

[3] Bicyclist travels in a circle
http://www.physicsforums.com/archive/topic/t-58003_bicyclist_travles_in_a_circle.html

exerts in a direction so that one component opposes the change in linear velocity while the other component becomes the Centripetal force.

Centrifugal force:

According to Newton's 3rd law when the Centripetal Force pushes the agent towards the centre there must be an equal and opposite reaction that the agent exerts a force on the surface. This is known as the Centrifugal Force. Since the centrifugal force does not affect the agent's motion we need not model it in our physics model.



Centrifugal force diagram[viii]

- Momentum

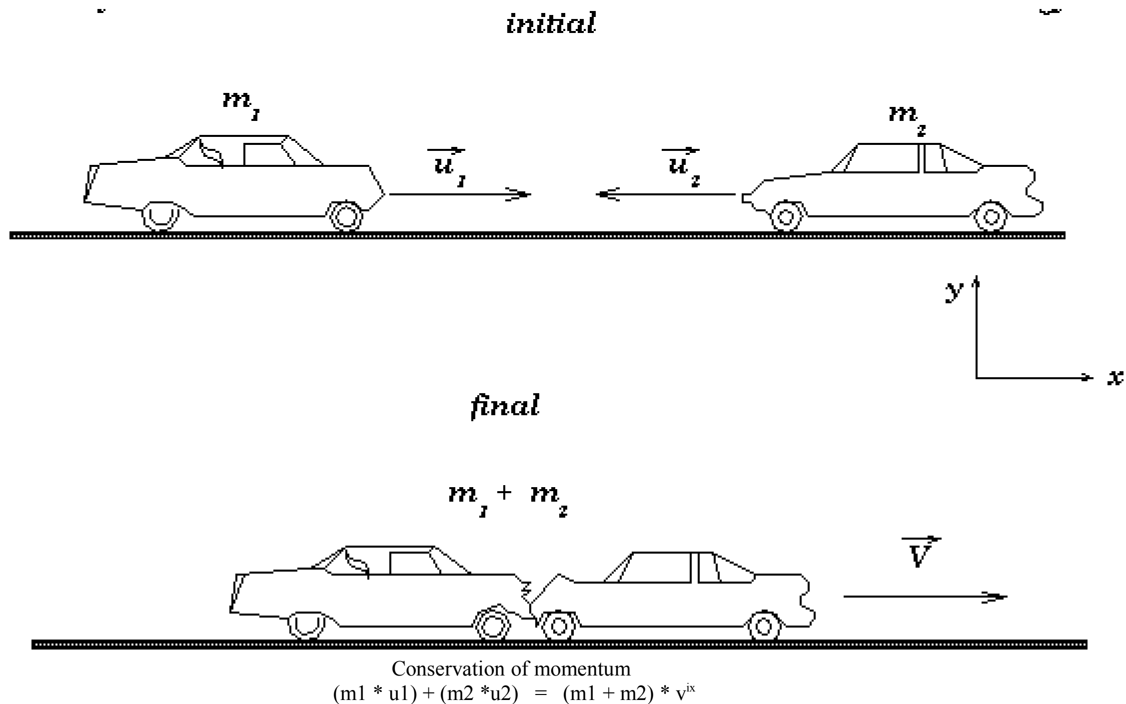
The vector quantity momentum of a particle is defined as the product of its mass times its velocity. It is useful when dealing with a system with agents colliding with each other.

Momentum = mass * velocity

However the momentum of individual agents by themselves is not useful; it is the principle of Conservation of Momentum, which we can use to analyse collisions. This principle states, "*The momentum of an isolated system such the Pop-Racer world is a constant. The vector sum of the momentum of all the objects of a system cannot be changed by interactions within the system. This puts a strong constraint on the types of motions, which can occur, in an isolated system. If one part of the system is given a momentum in a given direction, then some other part or parts of the system must simultaneously be given exactly the same momentum in the opposite direction. As far as we can tell, conservation of momentum is an absolute symmetry of nature. That is, we do not know of anything in nature that violates it.*"[4]

[Total momentum of the system at time t1] + [Total momentum of the system at time t2] = Constant

[4] Conservation of momentum
http://hyperphysics.phy-astr.gsu.edu/hbase/conser.html#conmom



Conservation of momentum
(m1 * u1) + (m2 *u2) = (m1 + m2) * v[ix]

Another value important model is the impulse on the agent during a collision. Impulse is the rate of change of momentum of the agent.
If the momentum of an object M1 before a collision was (M1 * V1) and (M1 * V2) was the momentum after the collision then the Impulse on M1 is:

Impulse = (M1 * V1) – (M1 * V2) / t
= M1 * (V1 – V2)/t

The final principle needed for modelling collisions in our model is the Work-Energy principle. In order to keep our model system simple we will assume that the all the energy that is possessed by the Pop-Racer system is conserved. In such a system the Work-Energy principle states that the net work done by the agent is equal to the change in the kinetic energy of the agent.

For an object of mass M1, the net work done when its velocity changes is:

N-work = 0.5(M1 * (V2^2)) – 0.5(M1 * (V1^1))

The structure of the model

The physics model can be broken down into two parts.

1. Basic procedures

The first step was to write small procedures that correspond to basic physical equation. This was relatively straightforward.

2. Main procedure

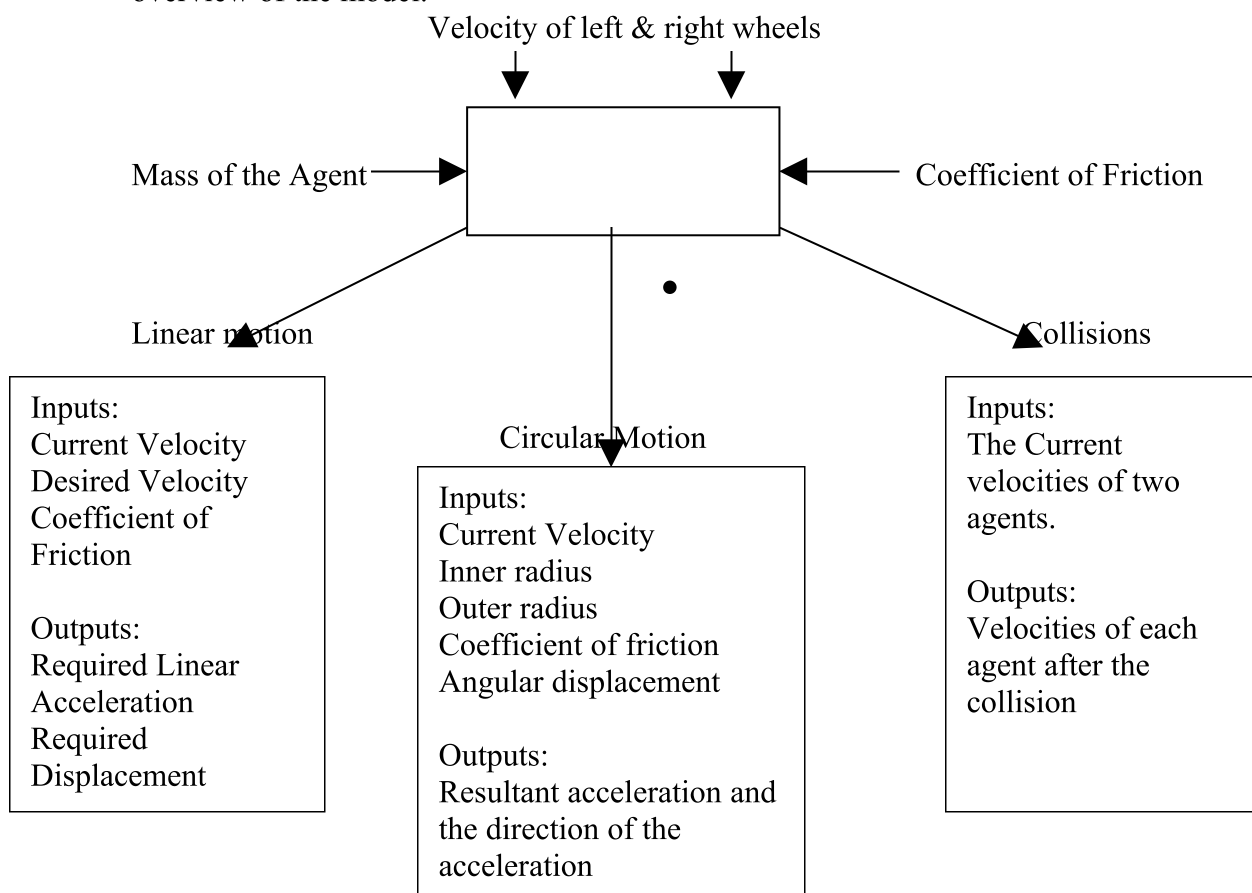
The main procedure was built using the basic procedures. As detailed in the project this procedure will take certain values of the agent and the system as inputs and using these values it will return output values back to the agent. The output values reflect the physical constraints that are being applied to the agent.
The main procedure itself was modelled as three parts.

1. Linear motion section
This is to implement the agent's behaviour when moving in a straight or almost straight path.

2. Circular motion section
This is to implement the behaviour when the agent is driving along a curve

3. Collisions section
This section implements the behaviour when agents collide. In order to keep the model simple we assume that at any given instance only two agents collides.

The idea was to have an input argument which, depending on its value, would trigger the relevant section of the physics model. The following diagram gives a general overview of the model.



Progress made

To date the sections on linear motion and circular motion has been completed. Due to time constraints the section on collisions has not reached completion.

In addition attempts were made to implement this physics model in to the main program but unfortunately it was not successful. Similar time constraints prevented modification of the model sufficiently to produce a satisfactory result.

© Team popCorn

[i] Picture of the car:
http://www.hawaii.gov/dbedt/ert/activitybook/pg02-car.gif

Picture of the bicycle
http://www.bikelite.com/images/ibl-logo.gif

[ii] Car accident picture
http://www.nimmer.net/legalgraphics/IMAGES/accident.gif

[iii] Different forces on an agent
http://members.aol.com/asa55net/pic/161a.gif

[iv] Friction diagram
http://www.visionengineer.com/ref/friction.gif

[v] Friction on a slope
http://farside.ph.utexas.edu/teaching/301/lectures/img518.png

[vi] Centripetal diagram
http://hyperphysics.phy-astr.gsu.edu/hbase/cf.html

[vii] Centripetal diagram 2
http://hyperphysics.phy-astr.gsu.edu/hbase/cf.html

[viii] Centrifugal diagram
http://hyperphysics.phy-astr.gsu.edu/hbase/corf.html

[ix] Conservation of momentum
http://dept.physics.upenn.edu/courses/gladney/mathphys/images/p74.gif

V. User Guide

Running PopRacer

Load a terminal session

Change directory to the directory where PopRacer is located.
Type ‘setup Poplog’ to ensure Pop-11 is set up.
Type ‘pop11 main.p’

This loads up ‘PopRacer’ with default settings.

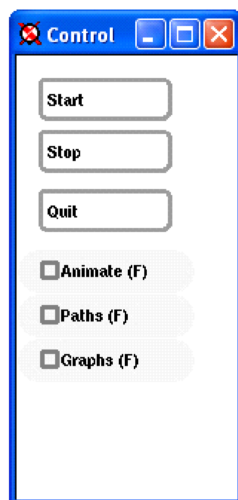
- Figure of Eight Track
- Standard Friction Coefficient of 0.002
- Neural Networks with 1 Layer of 7 Hidden Units.
- A simulation length of 2500 cycles.
- A quick train limit of 500 cycles.

Using PopRacer

When you load PopRacer, you are presented with a command line interface.
Typing ‘help’ will list the available commands.

Starting the Simulation

You can simply type ‘run’ to start the simulation, this will present you with the Main Simulation Window (with a figure of eight track), three graphing windows and a control panel (shown below).



- To start the learning click the ‘Start’ button.
- PopRacer will now run the simulation using some randomly generated cars (without drawing the cars or paths).When at least two cars have learnt to navigate the track within 500 cycles of the simulation, they will then be animated.
- You can also toggle this during the learning process by clicking on ‘Animate’.
- It is also possible to have the paths drawn as the cars move (which is a good compromise between speed and enabling you to see the learning process).
- The simulation will also begin to graph the average fitness and best fitness values for the current population of cars being trained.

Finally you can enable real-time graphing (by clicking ‘Graphs’) of the best car's velocity, which enables you to see how the cars slow down for bends and speed up on straights.
Once these cars are trained up you can then save the population to a file.

Saving Car/Genetic Algorithm Populations

It is easy to save a trained set of cars to a file, the procedure is below:

- •Click Stop on the Control Panel, type ‘savecars’ followed by the file
- •In the terminal type ‘savecars’ followed by a space and the file name.
- •PopRacer will also ask in the terminal if you want to add any notes to the population file, type something in and press enter.
- •The current car population will now be saved to a file.

Loading a Car/Genetic Algorithm Population

Once you have trained up a set of cars, you can test the cars on different tracks with different friction values.
You first have to quit the simulation by clicking ‘Quit’, and then reload the simulation.
At the PopRacer command line type ‘loadcars’ followed by a space and the file name of a previously saved population.

Tracks

You can now run these cars on different tracks; either a using a hard-coded or custom built one.

There are several harded coded tracks which can be selected at the PopRacer command line by typing ‘settrack’ followed by the name of a track which is listed below:

- eight
- straight
- rally
- silverstone
- hamburg

Using the Track Editor

You can also design your own track using the track editor.
To load the track editor type ‘createtrack’ at the PopRacer command line.
This will present you with a new window ‘Track Editor’.

To create a track simply click points (using the left mouse button) on the window, when you are done use the PopRacer terminal to save the track by typing ‘savetrack’ followed by a space and the file name.

If you wish to start again click the right mouse button.

This track is now also in PopRacer’s memory, so when you type ‘run’ the simulation will use the track you have designed (if you do not want this to happen use ‘settrack’ to change the track).

Loading a saved track

Now you have designed a track and saved it, you can load the track into PopRacer.
This is done by simply typing ‘loadtrack’ followed by a space and the file name at the PopRacer command line.

Other Settings and Commands

You can also specify many other settings such as the friction coefficient, structure of the neural networks and simulation length.
All these commands are found with an explanation by typing 'help' at the PopRacer command line.

VI. General Evaluation

In the project's specification document, we set some specific aims and objectives for the project. We said that the result would be considered as successful if:

- Driving paths are represented in a clear and realistic way, so that it is possible to judge how well the agents drive the car
- The agents can effectively learn and this within a reasonable amount of cycles and time (30 minutes training for example)
- The agents drive successfully around the track following a near optimal racing line without an inordinate amount of problems (spinning round in circles, running into barriers, etc…)
- We can produce different driving algorithms for individual agents and run them simultaneously in order to compare their performances (this being an extra if time allows)

From those criteria for success, we can list the following achievements:

- The graphical representation of the track exceeds our expectations. We drew many circuits (Silverstone, Imola, and many more) and it is even possible for the user to draw their own track
- Given a set of points, the agent can find its way around them quite quickly and just after few mutations. Once it has been well trained on a track it performs equally well on any given track given time to adapt
- The agents can complete the tour of the circuit, remaining on the track and if for some reason they slide and go off the track they are able to compensate and correct their position
- The GUI has been represented in such a way that any user is able to run the software, providing several performances' graphs, as well as an inbuilt user guide and help menu facility to navigate through the options

Despite all these achievements, there are still room for improvement on the project because the following problems are still present:

- Some populations tend to hold on to some bad behaviour such as the tendency to spin around, or to wriggle like a fish. This is because we do not want to apply too much restriction on the agent and therefore interfere and reduce the impact of the neural network that is responsible for generating of the input responsible for such behaviours
- Due to the very short period of time allocated, it hasn't been possible to implement some of the physics models we intended to use such as the differential equation to control the speed of the agent in different part of the tracks, and also the collision detection and included in the Bezier
- The unreliability in some features (eg. When using buttons) of the software used to implement the project i.e. Pop 11 meant that we had to cut very short on the number of buttons, and use mainly the terminal to run the system

VII. Conclusion

The aim was to produce agents that can drive around the track in an optimal way. We needed to produce:

- A racing track drawn using the Bezier curve algorithm combined with pixel colour recognition for collision detection
- Some learning algorithms such as the genetic algorithm for selecting and evolving the best cars, multi-layers perceptron Neural networks and instance-based learning (K-neighbours) for the collision detection
- Physics models to constrain the agent to its environment, making use of the friction, the acceleration, the velocity, the differential steering, etc…

Considering the aim and objectives set for this project as well as the achievements, we can safely and proudly say that the project was a very big success knowing that we managed to implement all or nearly all the specifications set except for the extra.
The project however is still open for improvement and can be further developed in lots of different ways; particularly it will be interesting to see how agents implemented by different AI algorithms can compete against each other.
It has a massive potential.

VIII. Acknowledgments and Bibliography

We would like to thanks everybody who participates in the development of this project. We are very grateful for you support and disponibility throughout the life of the project. Special thoughts to :

Dr Mark Lee for supervision and the assistance in the management of the project
Mr Aleem Hussain for his advice and support
Dr Aaron Sloman for the technical support on the use of the programming language Pop-11

The sources used to complete this piece of work are from:

Stuart Russell & Peter Norvig (2003, 1995) *Artificial Intelligence A Modern Approach*. Pearson Education Inc, Upper Saddle River, New Jersey, USA.

Aaron Sloman (1997) *Teach Primer- An Overview Of Pop-11*, 2nd Edition, The University of Birmingham, UK

Dr Ata Kaban (2005), *Lecture's Handout on Instance-Based Learning*, the University of Birmingham, Uk.

Dr Sorge, V & Styles, I (2005). *"Raster conversion algorithms for curves: 2D splines"*

David B. Leake : http://www.cs.indiana.edu/~leake/papers/p-01-07/p-01-07.html

http://www.ai-junkie.com

the project blog at http://pop11.blogspot.com

the project manager at http://www.aceproject.com/server01/Login.asp

but you need a login

the mail: pop11@tamias.co.uk

IX. Appendixes

Appendix 1: The Self Assessment Forms of the team members detailing the breakdown of tasks

Appendix 2: The printouts of the source code

Appendix 3: Minutes of the team meetings.