Tank Wars! Evolving Steering and Aiming Behaviour for Computer Game Agents

Yvan Bourquin
Essay in Adaptive Systems
Candidate No 81214 for MSc in Evolutionary and Adaptive Systems
Department of Informatics
University of Sussex, Brighton, United Kingdom

1 Abstract

This document describes the methods used to obtain steering and aiming behaviour in autonomous computer game agents. The agents are tank-shaped vehicles equipped with mobile gun turrets. They learn to avoid collisions and fire each other in a two-dimensional maze environment. The tanks' behaviour is not programmed explicitly; it is controlled by continuous time recurrent neural networks (CTRNN) [Beer 1996, Slocum et. al., 2000] whose parameters are self-programmed using evolutionary methods.

2 Introduction

Traditional obstacle avoidance and also discrimination tasks, minimal cognition, short-term memory and selective attention [Beer 1996, Slocum et. al., 2000] are currently achievable by evolutionary robotics. However, to date only simple behaviour can be obtained and there is a long way to go until these techniques can be applied to physical robots in a useful manner. However simple behaviour is precisely what is required in many computer games. For example, playing the famous Tetris or Archanoid games needs nothing more than simple discrimination and sensory-motor coordination. Therefore evolutionary methods using CTRNNs can be used to create artificial opponents or allies for human players in this sort of game and the entertainment industry should be interested in this kind of technology.

Evolutionary algorithms have already been used with success in some computer games, however, so far these methods have not been used to create sensory-motor computer games. I therefore think that this is an interesting subject to explore.

3 Scenario

Many different game scenarios were possible candidates to look into minimal behaviour. A simple multiplayer search and destroy tank fight scenario was chosen because it seemed to require the kind of behaviour that is known to be achievable with CTRNNs according to the literature. The scenario consists of two, or more, tanks moving about and firing at each other in a simple two-dimensional maze environment. Each tank is equipped with a rotating turret holding a gun. There are no teams; every other tank encountered is considered as an enemy. The tank bodies are equipped with six proximity sensors. Those sensors inform the tank of the distance to solid objects so that collision with walls or other tanks can be avoided. The tank turrets are equipped with six vision sensors that allow the tank to track its enemies.

4 Methods

4.1 Simulator

The simulator essentially detects collision by a method based on line intersections. The simulation speed can be increased or decreased during run-time. The simulator can also be switched to an offline mode where it works as fast as possible for the available CPU power.

4.1.1 Proximity Sensors

The proximity sensors measure the distance between the tank and an obstacle located within a restricted range. Every solid object is considered as an obstacle, no distinction is made between a wall and a tank. Two sensors are mounted at the rear and four at the front because when the tank is in motion, a frontal collision is more probable (see Figure 1). The range of each sensor is of 30 units[1]. The sensor output is inversely proportional to the distance to an obstacle and scaled to fit in the range [0, 1]. Therefore an object beyond 30 units generates a sensor activity of 0 and object in contact with the sensor gives 1.

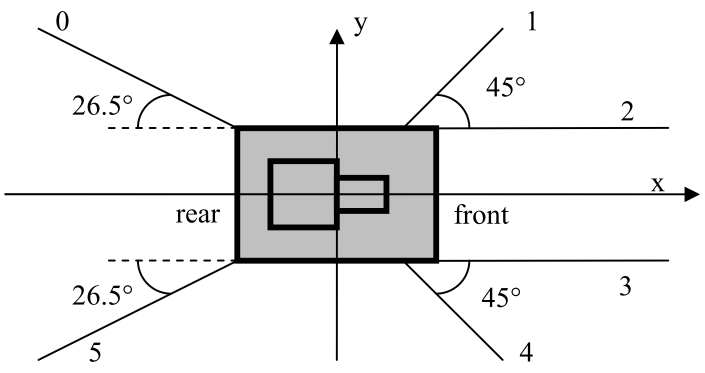

Figure 1: Proximity sensors

Sensors number 2 and 3 are aimed to avoid frontal collision with other tanks while number 1 and 4, placed at 45°, are designed to enable wall following behaviour. The rear sensors aim to avoid rear collision when the tank manoeuvres.

4.1.2 Vision Sensors

The vision sensors are fixed on the turret and move with it. In order to facilitate aiming, the turret rotation is decoupled from the body motion. Therefore, when the tank turns, the turret keeps its global orientation[2]. The vision sensors are placed at close angles from each other in order to enable accurate aiming (see Figure 2). When a vision sensor ray intersects a tank body, it returns a value in the interval [0, 1] inversely proportional to the distance to the tank. In every other case, whether the ray meets a wall or not, the sensor returns 0.

[1] Since this is not a simulation of the real world, the units are arbitrary. However for concreteness distances can be assumed to be in meters and times in seconds.
[2] Note that this is similar to some real tanks where the turret is controlled by gyroscopes; it remains stable and this allows shooting while moving.

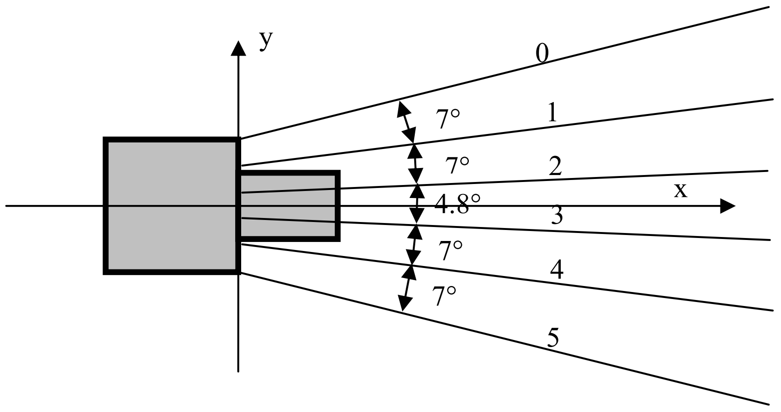

Figure 2: Vision sensors

The vision sensor operates like real vision only in the sense that it is insensitive to objects hidden by other objects. For example it cannot see a tank hidden by a wall.

4.1.3 Tank

The tank motion is controlled by two simulated motors. The neural network outputs are not connected directly to the motors. Instead they are considered as indications of the desired speed. At each time step the effective motor speed is changed so that its difference from the desired speed is halved; a kind of acceleration effect results. In addition, some Gaussian noise is injected into the motors in order to slightly randomize motion and to make the simulation indeterminate.

4.2 Neural Network

4.2.1 CTRNNs

The CTRNN architecture was chosen because it is known to work well for the type of behaviour that is desired for the tanks. For example [Beer 1996] uses CTRNNs for orientation and pointing: similar tasks to those we require of the tank turret. Because the CTRNNs have their own internal dynamics, they are able to display some memory effect. For example, this memory effect enables the emergence of object persistence in Beer's orientation experiments [Beer 1996]. Object persistence is typically what is required for the turret's aiming behaviour. If the turret is aiming at a target that disappears behind a wall, it is very much desired that the turret continues its motion for a while with the same speed and direction, in order to eventually be able to catch sight of the target as it reappears at the other end of the wall.

CTRNNs can also provide efficient obstacle avoidance behaviour. Of course, a purely feed-forward network can implement Braitenberg-vehicle behaviour and avoid obstacles; however it can only determine its behaviour according to the current stimuli. In our setup some of the obstacles are in motion and therefore a network with internal dynamics like a CTRNN is more likely to be able to take that motion into account, at least to a small extent.

4.2.2 Task decomposition

The initial tests showed that evolving a complex task is not straightforward and takes a lot of time. Evolving the steering and aiming behaviour within a single network might require an excessive amount of time or just fail. A CTRNN for controlling both steering and aiming requires twelve sensory inputs, four motor neurons and a number of intermediate neurons. Therefore the parameter search space becomes large. For that reason it was decided to start with two independent neural networks for the two different tasks; one CTRNN for steering and another one for aiming. With this simplification the problem was decomposed into two behaviours which have previously been solved using CTRNNs. Since both the tank body and turret have the same number of sensors and motors, it was possible to use the same architecture for both. On the other hand, the evolutionary process was carried out using two distinct populations and fitness functions.

4.2.3 Synaptic Interconnection

In the literature, the CTRNN neurons' interconnection design varies according to the nature of the problem and the authors. For example [Blynel & Floreano 2002] use neurons which are fully recurrent, self-connected and also connected to every sensor. In the octopod locomotion system of [Jakobi 1998], each leg is controlled by a fully interconnected CTRNN from which two neurons are connected to the leg motors[3].

The "fully interconnected" approach is tempting because it leaves to evolution the task of finding the optimal solution. However it is possible to make some assumptions of what a functional evolved network would look like. These assumptions allow leaving out some connections at the very beginning and therefore reducing the number of parameters to evolve. With less parameters, convergence has more chance to happen and can also happen faster.

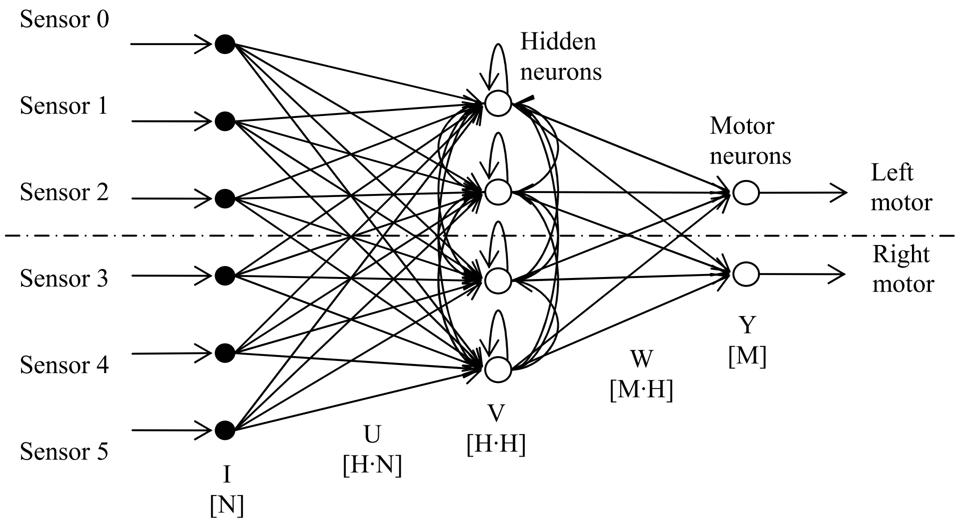

Figure 3: The tank's two-layer CTRNN architecture

The chosen architecture (Figure 3) uses self- and recurrent connections, but only in the hidden layer. Recurrent and self-connections between motor neurons, or backwards, from the motor neurons to the hidden neurons are excluded.

[3] Note that the CTRNNs are also connected with each other. Each one is connected to the CTRNN of the opposite leg, the legs in front and at the rear in a circular manner.

4.2.4 Neuron Model

The neuron model is based on the CTRNN equations described in [Beer, lecture notes]. The state of the neural network shown in Figure 3 is computed in two different phases. The first phase computes the new motor neuron states from the previous motor neuron and hidden neuron states and network parameters: $y = f(y, \gamma, w, \theta, \tau)$, and it is ruled by the equation

$$y_i(n+1) = y_i(n) + \frac{\Delta t}{\tau_i}(-y_i(n) + \sum_{j=1}^{H} w_{ij}\sigma(\gamma_j(n) - \theta_j)) \qquad (1)$$

where i is an index $(i = 1, 2, \ldots, M)$ and M is the number of motor neurons, H is the number of hidden neurons, the y_i are the current states of the motor neurons, Δt is the integration step size, τ_i is a motor neuron time constant, w_{ij} is the weight of the connection from the hidden neuron j to the motor neuron i, $\sigma(x) = 1 / (1 + e^{-x})$ is the standard sigmoid function, and θ_j is a bias term.

The second phase computes the new hidden neuron states from the previous hidden neuron states, from the weighted sensory inputs and from network parameters: $\gamma = f(\gamma, I, u, v, \theta', \tau')$, and it is ruled by the equation

$$\gamma_i(n+1) = \gamma_i(n) + \frac{\Delta t}{\tau'_i}(-\gamma_i(n) + \sum_{j=1}^{H} v_{ij}\sigma(\gamma_j(n) - \theta'_j) + \sum_{k=1}^{N} u_{ik} I_k) \qquad (2)$$

where γ_i is the state of the hidden neuron i, τ'_i is a hidden neuron time constant, v_{ij} is the weight of the connection from hidden neuron j to i, θ'_j is a bias term, N is the number of sensory inputs, u_{ik} is the weight of the connection from the sensory input k to the hidden neuron i, I_k is the sensory input k, and H, Δt and $\sigma(x)$ are defined as before.

Note that this two-layer architecture corresponds to a fully connected CTRNN where, as shown in Figure 3, some connections were removed. The numerical results are exactly the same but less processing is required here because only N·H + H·H + H·M floating point multiplications are necessary for every time step, compared to N·H + $(H+M)^2$ for a fully connected network where the corresponding connections would be set to zero.

The new motor neuron states are computed in the first phase, because this calculation requires knowledge of the previous hidden neuron states. If the sequential order of the two phases was inverted then the new motor neuron state would be calculated from the new hidden neuron states and this would no longer correspond to the correct processing sequence of a CTRNN.

4.2.5 Bilateral Symmetry

Bilateral symmetry was another important technique used to keep the number of evolvable parameters low. Bilateral symmetry was applicable here because both the steering and the aiming tasks are intrinsically symmetrical; a sensory pattern perceived on the left side of the tank (or turret) is expected to produce a motor activation that is symmetrically identical to the same sensory input perceived on the right side.

4.2.6 Genetic Encoding

The neural network parameters are encoded into a genotype of floating point numbers initialized with values in the range [0, 1]. As a result of the bilateral symmetry the genotype size is divided by two because two symmetrical neurons (see the axis of symmetry in Figure 3) are encoded together.

Hidden Neurons 0 and 3												
θ'	τ'	θ	u						v			
0	1	2	3	4	5	6	7	8	9	10	11	12

Hidden Neurons 1 and 2												
θ'	τ'	θ	u						v			
13	14	15	16	17	18	19	20	21	22	23	24	25

Motor Neurons 0 and 1				
τ	w			
26	27	28	29	30

However bilateral symmetry also adds some complexity to the genetic decoding; the weight matrices u, v and w must be decoded using bi-axial symmetry.

4.3 Genetic Algorithm

For all the experiments described in this essay a population of 100 genotypes was evolved. The genetic algorithm divides the genotypes into 50 groups of 2. Two genotypes at a time are embodied into tanks (phenotypes) and placed into the simulator at a random position. According to the setup, 1 or 5 fights take place with the same two genotypes; starting every time with different position and orientation, and the average fitness is calculated. This is better explained with the short pseudo-code below:

```
for (each generation) {
     for (each group) {
          getTwoGenotypes()
          for (each fight) {
               embodyGenotypesIntoTanks()
               simulateFight()
               measureFitness()
               destroyTanks()
          }
          averageFitness()
     }
     sortByFitness()
     reproduce()
     mutate()
}
```

Every fight last for 4000 simulated time units which correspond to about one second of real time.

4.3.1 Mutation

Floating point versus integer encoding was used in order to avoid "hamming cliffs"[4]. Two different types of mutation were used. The first type is a floating point mutation with probability 0.04 of mutation per gene; a random number from a Gaussian distribution with mean 0 and standard deviation 1 is added to the gene value.

[4] Hamming cliffs are a common problem with binary encoding. For example if the binary number 00101101 is mutated to 01101101, although only one bit is changed, the real value difference is large (64).

The second type is the hypersphere mutation; the whole genotype is considered as an N-dimensional vector. A random N-dimensional mutation vector is added to the genotype. The direction of the mutation vector is chosen in every dimension from a uniform distribution, while its length is chosen from a Gaussian distribution with mean 0 and standard deviation 1. Standard deviation 5.0 and 0.2 were also used in some experiments, where specified.

4.3.2 Crossover

In every experiment, a single point crossover was applied with a probability 0.05. No mutation was applied on genotypes created through crossover.

4.3.3 Selection

Rank based selection was used with a mean viability of 0 offspring for the least fit genotype and 2 offspring for the fittest genotype. An elitist fraction of 5% is copied unchanged (no mutation or crossover) from the current generation to the next.

4.4 Fitness Functions

4.4.1 Steering Fitness

Finding the right fitness function to evolve motion and obstacle avoidance could have been a delicate task. Fortunately there were many examples in the literature. The fitness function from [Mondada & Floreano 2] was used successfully. It is defined as

$$\Phi = V \cdot (1 - \sqrt{\Delta v}) \cdot (1 - i) \qquad (3)$$

where *V* is the average speed of the two motors, *Δv* is the difference between the signed speed value of the two motors (positive values are forward, and negative values are backwards), *i* is the activity value of the most active sensor, meaning the closest to an obstacle (a value of 1 corresponds to a collision and a value of 0 means no obstacle). All three variables were scaled to lie between 0 and 1. Unlike [Mondada & Floreano 1995] a signed value of *V* was used here for the tanks and therefore going backwards counted as negative fitness. Therefore the tanks "preferred" the front direction on account of the fitness function, whereas in [Mondada & Floreano 2] the Khepera robots "preferred" the front motion because they had more sensors at the front and it was easier to avoid collision in that direction.

The modified fitness function without sensor activity also allowed the evolution of fast tanks that were actually less "reluctant" to move close to the walls and therefore were better suited to pass each other in narrow passages.

$$\Phi = V \cdot (1 - \sqrt{\Delta v}) \qquad (4)$$

4.4.2 Aiming Fitness

In the tank game many objects are moving: the shooting tank, the target tank, the turrets and the shells. The shells move in straight lines, but the tanks can have quite unpredictable trajectories. For that reason, counting the fitness as the number of shell hits on the opponent's tank would leave too much to chance. It was not certain that a slightly more evolved neural controller would score significantly better than a less evolved one. For that reason it was decided that the fitness function should reward correct aiming only, without taking into account the actual number of shell hits. Furthermore, it was decided to grant a partial fitness reward for a partially accomplished aiming task; aiming with the lateral vision sensors would be rewarded, but to a lesser extent than aiming with the central ones. The chosen fitness function was:

$$\Phi = \sum_{i=1}^{N} I'_i r_i \quad (5), \text{ where } \quad I'_i = \begin{cases} 1 & I_i > 0 \\ 0 & \textit{otherwise} \end{cases}$$

and where N is the number of vision sensors, I_i is the current activation value of sensor i, r_i is a reward value for the sensor i, and was defined as [0.1, 0.3, 1.0, 1.0, 0.3, 0.1] thus encouraging more strongly activity of central vision sensors. The firing mechanism was designed to be distinct from the neural controller and to be triggered automatically when at least one of the central vision sensors was activated.

5 Training and results

The genotypes were initialized with uniform random numbers in the range [0, 1]. These numbers were scaled to become suitable neural network parameters but were allowed, through mutations, to exceed the initial range. The network weights u, v and w were scaled to lie in the range [-5, 5], allowing bi-stable neurons. The neuron biases were scaled to the range [-2, 2] and the neuron time constant to the range [1, 10]. The integration time step Δt was 0.1. The network states y and γ were initialized to 0.

5.1.1 First experiment

A population of 100 genotypes was evolved in an empty arena of 140 x 80 units. After 200 generations, around 15 hours of real time, the resulting behaviour was fast forward motion, efficient wall avoidance and somewhat less efficient tank avoidance.

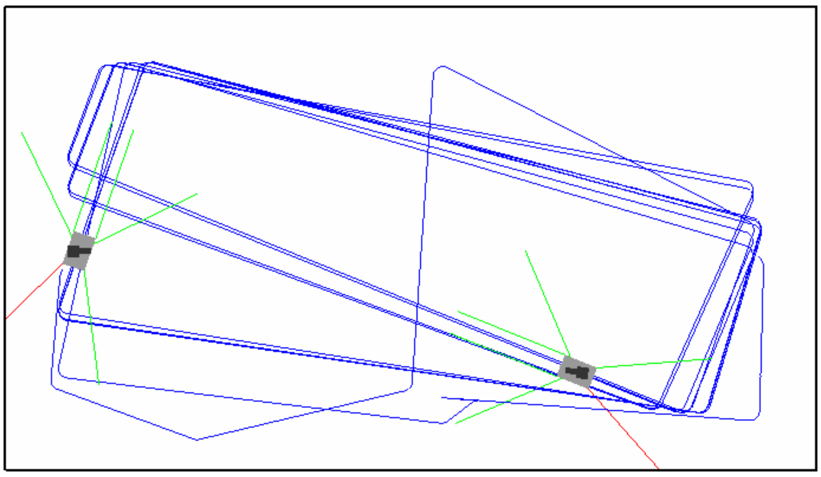

Figure 4: Straight motion and collision avoidance after 200 generations

The tanks learned to move completely straight (Figure 4) and to turn only as a response to sensory activity.

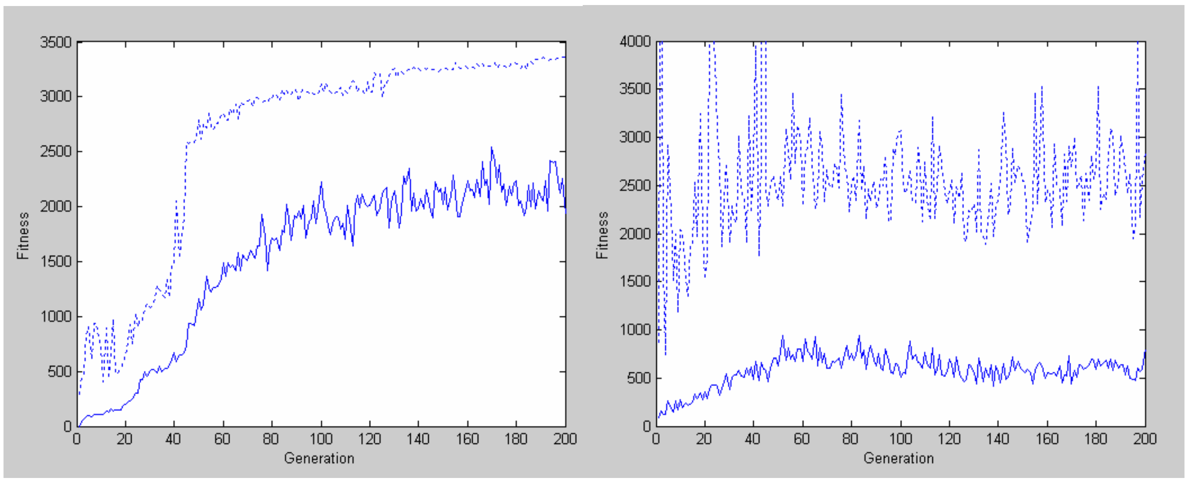

Figure 5: The learning curves: Steering (left), aiming (right). The upper curves show the best individual of each generation.

This resulting behaviour, made of straight runs and sharp turns, is almost the best possible result that can be obtained with the specified fitness function. The best result would be to follow the outer wall and turn 90° at each corner. Aiming behaviour did not work at this point (see Figure 5). Regular rotation of the turrets, clockwise or counter clockwise, was the only result.

5.1.2 Second Experiment

The desired aiming behaviour failed to appear with the previous setup and therefore the task needed to be further simplified. The decision was made to evolve the aiming behaviour in three steps; first starting with an empty environment and immobile tanks, then adding motion and finally, adding walls. The same parameters as in the previous experiment were used, but the tank motors were disconnected and the tanks were placed back to back, one in the upper part of the arena and the other one in the lower part.

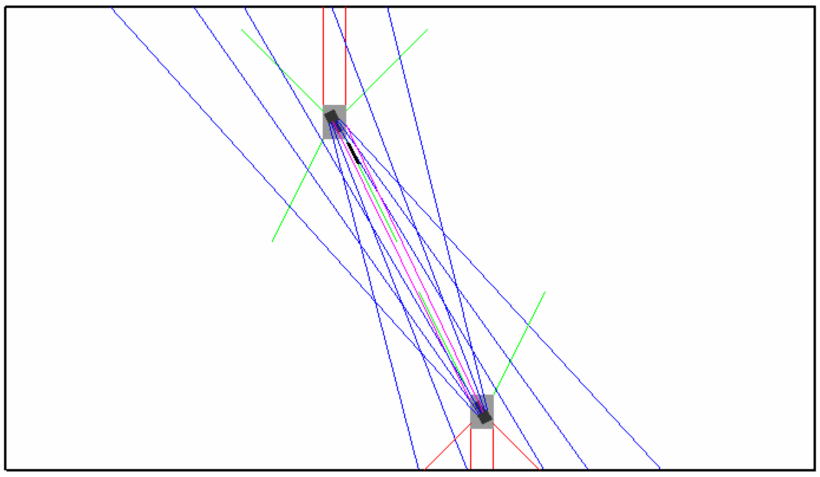

Figure 6: Two motionless tanks "staring" at each other.

At the start of each "fight", the upper tank's turret was oriented straight up and the lower tank's turret was looking straight down. The idea of this configuration was to make it impossible for a turret to have the correct orientation towards the other tank by chance; "deliberate" aiming was required. Though, with this configuration, all the vision sensors remained inactive. No input was injected into the neural network and therefore, because of the bilateral symmetry, exactly the same output was produced for both the left and right turret motors. For that reason the turret did not move at all. The problem was corrected by injecting a small amount of Gaussian noise into the vision sensors. Afterwards the desired behaviour appeared quickly at around the 50th generation. The tanks learned to turn the turret and "stare" at each other as is shown in Figure 6 where the sensor rays are represented.

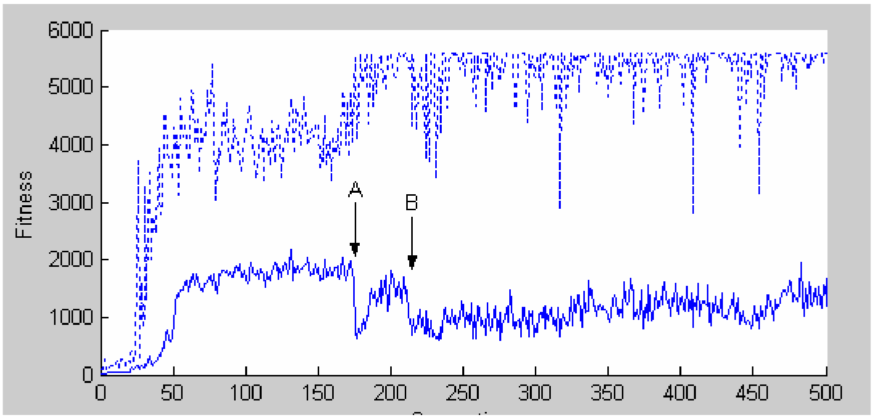

Figure 7: Evolution of aiming behaviour in three phases

After 175 generations the setup was modified and the tanks were allowed to move freely. Since the task of aiming at a mobile object was more difficult, a drop appears in the learning curve at position A (Figure 7). Then again the mean fitness increased until around the 210th generation. Then walls were placed in the simulator resulting in another drop at position B. Afterwards the fitness increased only slightly. Note that the best individuals' fitnesses are bounded at around 5500 points. This maximal score is reached when two tanks collide and then are unable to resume motion. In this case the aiming task is very easy and many points are collected.

5.1.3 Third Experiment

A new experiment was started with a new random population, and an empty arena. This time hypersphere mutation was used. All other parameters were kept as in the previous experiments. The standard deviation of the mutation magnitude was initially 5.0. At the 100th generation (A) the mutation magnitude was lowered to 1.0 and at the 150th generation (B) it was lowered to 0.2 (See Figure 8).

Hypersphere mutation was more successful than the random gene mutation. A good aiming behaviour was obtained after 220 generations without the initial phase of immobilization. Obstacle avoidance was flawless; the tanks moved in straight lines and turned just in front of obstacles. When collision occurred the tanks would reverse, turn and resume the forward motion. A steep fitness increase can be observed in (A) where the mutation vector magnitude is decreased. The initial mutation magnitude of 5.0 was apparently too disruptive.

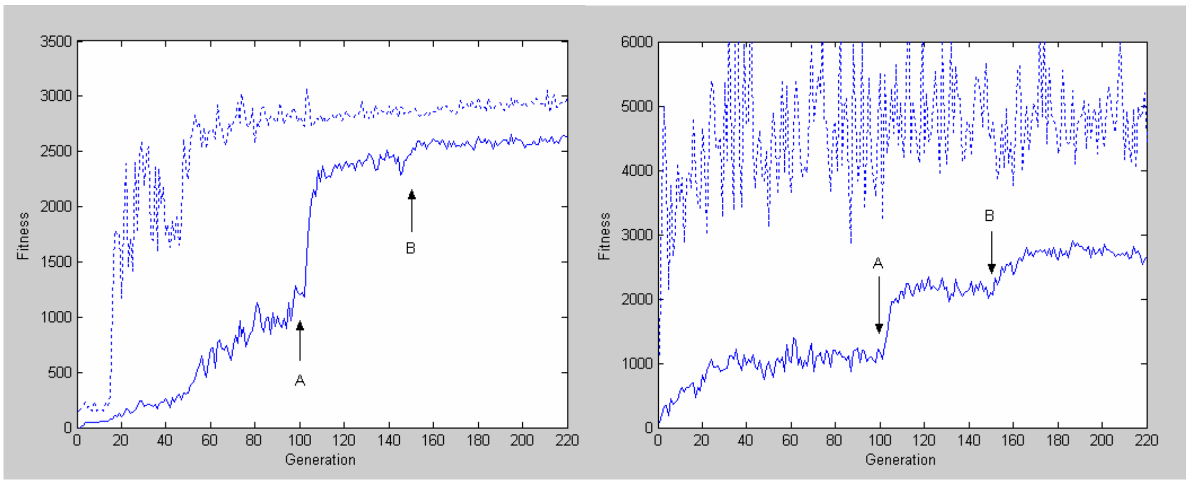

Figure 8: Progression of the average and maximum fitness for steering (left) and aiming (right). Changes to lower mutation magnitude appear in A and B.

With this setup, because there were no walls and because the vision sensor length was 140, just like the arena width, it was theoretically possible for a tank to see the enemy from everywhere but from two opposite corners of the arena. Since correct aiming was possible at almost every instant of a fight, it was therefore interesting to measure the aiming performance. Figure 9 and Figure 10 show the aiming angle error measured from one of the tanks. The aiming angle error indicates the difference between the current turret angle and the actual enemy sight angle computed exactly by the simulator. The horizontal axis represents a run of 2500 simulated time units (a complete fight is 4000 units) and the vertical axis shows the aiming angle error measured between –π and π.

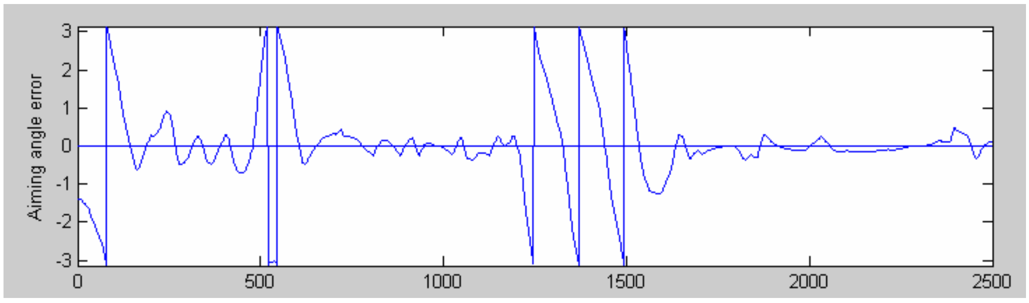

Figure 9: Turret angle error versus time in an 80 x 140 arena.

Figure 9 shows a run in an arena of standard size (80 x 140). After an initial search and "scanning" the turret finds the enemy which is then tracked with some oscillations. The enemy sight is lost two times. The first time, at around 500 time units, the turret faces in the opposite direction but then recovers. The second time the turret does three full rotations of 360° without noticing the enemy. The tracking is imperfect because when the two tanks are far from each other, the sensor signals are weak due to the distance. And furthermore, even if the turret is looking in precisely the correct direction, there is not always an intersection with a vision sensor ray because the space between the rays becomes larger with the distance.

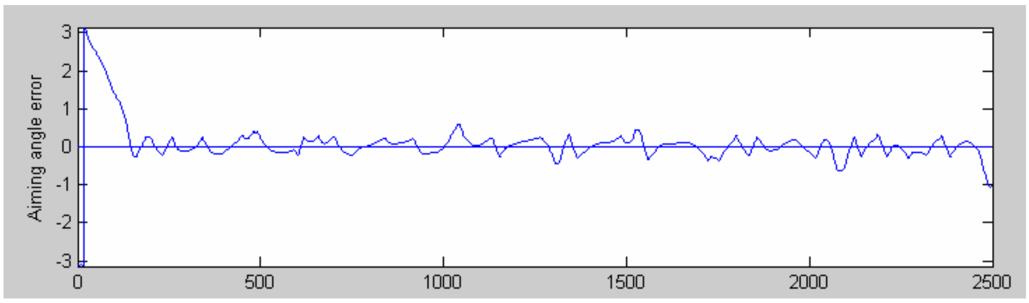

Figure 10: Turret angle error versus time in an 80 x 80 arena.

Figure 10 shows the turret angle error measured in a reduced arena of 80 x 80. As opposed to what happens in the larger arena, the aiming system does not lose the enemy's track with this setup because the signals are stronger and the rays closer.

5.1.4 Fourth Experiment

The previous experiment worked very well and the tanks developed remarkable steering and aiming performance. Now the last step was to put them in the final game environment with walls. For this experiment, the fitness was no longer averaged but rather computed for one fight only, in order to obtain results more quickly. The hypersphere mutation magnitude was reset to mean 0 and standard deviation 1. Four walls as shown in Figure 12 were introduced. The other parameters were kept as before.

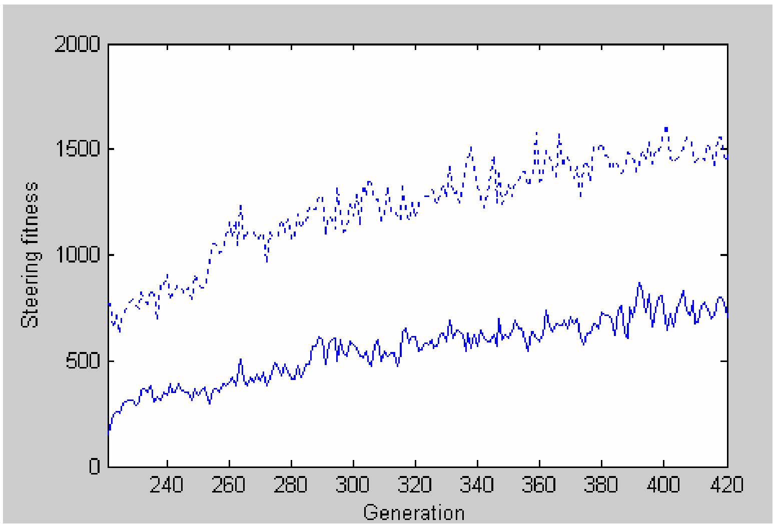

Figure 11: Re-adaptation of the steering behaviour to a walled environment.

With this new setup, the tanks started very badly with a steering fitness near 200, while it was around 2500 in the empty arena. Their training in the open environment was not very compatible with the narrow passages of the walled environment. The tanks were too "afraid" of entering and "hesitating" to move into narrow passages. However after around 200 generations they adapted and Figure 12 shows that the tank tracks remain safely away from the walls.

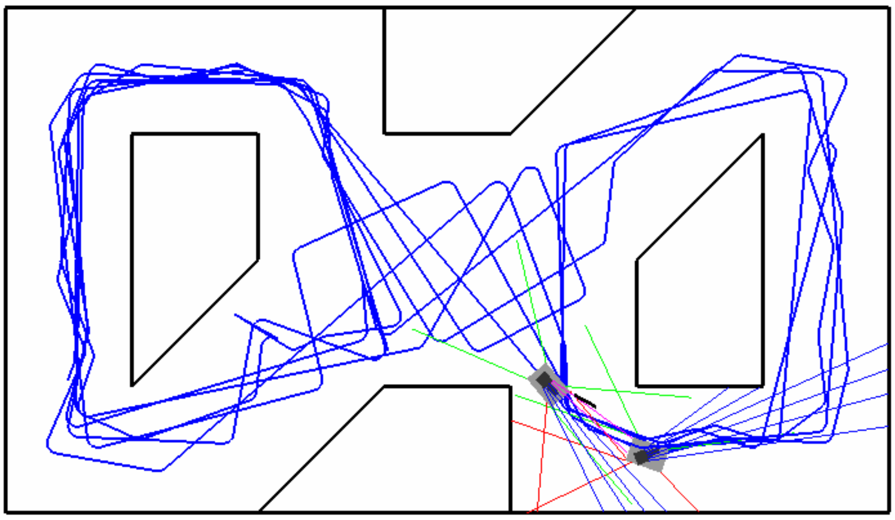

Figure 12: Tank fight into an arena with walls.

At this point the objective was reached, the tanks had developed good fighting skills and watching them was quite entertaining.

5.1.5 Fifth Experiment

At this time it was decided to check the earlier assumption that a single network would be less appropriate to solve the tasks than two separate networks. The tanks were modified in order to host a single neural network with 8 hidden neurons. The 12 sensor inputs and 4 motor outputs were connected to this network respecting bilateral symmetry. Then a new experiment was started, using exactly the same simulator and arena as before. The aiming behaviour developed, but only in an empty arena of immobile tanks. When the tanks were transferred to a mobile environment the aiming ability failed to readapt. On the other hand steering developed quickly and was able to readapt to every environment.

6 Discussion

6.1 Genotypes

In order to evaluate the suitability of the chosen initial parameter ranges, the two genotype populations were examined more closely. The mean value and standard deviation of each gene was computed and compared (see values in the appendix). There is no noticeable similitude in weight, bias or time constant between the two networks and therefore it is difficult to make any assumption as to what would have been more appropriate initial ranges. Figure 13 shows both populations as grey-level bitmaps; each line represents a single genotype. Both bitmaps share the same scale of grey-levels and therefore they can be compared; visually speaking, they differ totally. Because both populations encode exactly the same network architecture this dissimilarity might reflect the different solutions to the different tasks, but note evolution can also lead to a different outcome every time. For example the roles of two hidden neurons in the network can be swapped if all their parameters are equally and adequately swapped.

On the other hand, inside each population, a high degree of homogeneity is observed even though several solutions are theoretically possible. Eventually a larger population and a spatially distributed genetic algorithm as in [Cliff & Miller 1996], where crossover takes place primarily with neighbours would provide a less homogenous set of solutions.

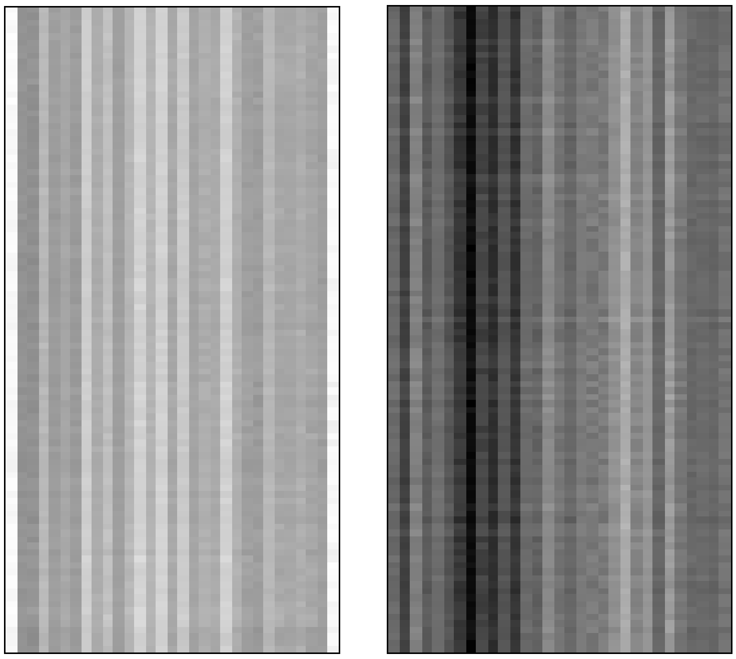
Figure 13: The final population of the fourth experiment, shown as grey levels: steering (left) and aiming (right). Each population is made of 100 genotypes (lines) and 31 genes (columns).

6.2 Two versus one network

The choice of using two distinct neural networks to evolve the game agents turned out to be the right one. The fifth experiment that used a single network did not allow obtaining results of the same quality as the previous design. In real tanks, steering and shooting are also carried out by two different operators. However many real tanks must stop before being able to shoot because accurate aiming is difficult from a vehicle in motion. In the simulation this type of behaviour would also have been an advantage. With a different design this kind of behaviour might have evolved by itself or could also have been programmed specifically in a kind of "subsumption architecture" where the different behaviour would have different priorities and therefore firing would pre-empt motion.

7 Conclusions

The steering behaviour evolved beyond expectations and was able to avoid static as well as mobile obstacles. It is however difficult to say if this performance was superior to that of a purely reactive system. To make sure it would be interesting to compare the performance of evolved feed-forward and CTRNN systems.

The aiming behaviour worked equally fine, but nothing like short-time memory seemed to appear. As soon as the targeted tank disappeared behind a wall, tracking was interrupted and the scanning behaviour resumed. Maybe the memory effect was too short for human eyes to notice. In any case the reason was not that the initial time constants were too small, because the evolutionary process tended to make them even shorter than the initial values.

In the tank design, the fitness for both steering and aiming was computed exclusively from information available to the agent through its sensors and motors. There was no reference to the external world or simulator data; therefore the tanks were learning autonomously. For that reason, this functionality is, in theory, transposable to physical robots.

Evolving a neural network to accomplish even a simple task can be tricky. Sometimes the evolution process needs a bit of supervision from the human who can set up intermediate goals or provide a fitness function that partially rewards partially fulfilled objectives. At other times, the evolution process is too "intelligent"; it every opportunity to do the most unexpected. We are always too naïve about the expected outcome of evolution.

8 References

R. D. Beer (1996). *Towards the Evolution of Dynamical Neural Networks for Minimally Cognitive Behavior*. In P. Maes, M. Mataric, J. Meyer, J. Pollack and S. Wilson (Eds.), From animals to animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (pp. 421-429). MIT Press.

A. C. Slocum, D. C. Downey, R .D. Beer (2000). *Further Experiments in the Evolution of Minimally Cognitive Behavior: From Perceiving Affordances to Selective Attention*. In J. Meyer, A. Berthoz, D. Floreano, H. Roitblat and S. Wilson (Eds.), *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior.* Cambridge, MA: MIT Press.

F. Mondada & D. Floreano (1995). *Evolution of Neural Control Structures: Some Experiments on Mobile Robots*. In Robotics and Autonomous Systems, 16, 183-195.

J. Blynel & D. Floreano (2002). Levels of Dynamics and Adaptive Behavior in Evolutionary Neural Controllers. In B. Hallam, D. Floreano, J. Hallam, G. Hayes, and J.-A. Meyer, editors. *From Animals to Animats 7: Proceedings of the Seventh International Conference on Simulation on Adaptive Behavior*, MIT Press.

D. Floreano (1998). Evolutionary *Robotics in Behavior Engineering and Artificial Life*. In T. Gomi (Ed.), *Evolutionary Robotics*, Ontario (Canada): AAI Books, 1998.

D. Cliff & G. F. Miller (1996). *Co-evolution of Pursuit and Evasion II: Simulation Methods and Results*. In P. Maes et al. (Eds.). *From Animals to Animats IV, Procs. of the Fourth International Conerence on Simulation of Adaptive Behaviour*, MIT Press, pp. 506-515.

N. Jakobi (1998). *Running Across the Reality Gap: Octopod Locomotion Evolved in a Minimal Simulation.* Proceedings of the First European Workshop on Evolutionary Robotics: EvoRobot'98, 1998.

R. D. Beer. Lecture notes: Continuous-Time Recurrent Neural Networks (CTRNNs)
http://vorlon.ces.cwru.edu/~beer/EECS477/CTRNNIntro.pdf

9 Appendix

9.1 Web link

A demo version of the program is available at:
http://www.yvanbourquin.com/tanks

9.2 Evolved Genotype Populations

Gene number	Steering (mean)	Aiming (mean)	Steering (stdev)	Aiming (stdev)
0	11.111	-9.7205	0.4143	0.7477
1	-3.6949	-15.5612	0.3741	0.7167
2	-3.9531	-6.3987	0.5726	0.7822
3	1.5967	-11.9028	0.4640	0.6662
4	-2.3576	-9.1755	0.3734	0.5114
5	-0.92907	-12.6139	0.3539	0.6825
6	-2.4717	-16.9800	0.4348	0.8969
7	4.758	-23.3588	0.3696	0.8477
8	-0.10602	-15.1250	0.3186	0.9489
9	2.7485	-18.3080	0.4574	0.8646
10	-2.0064	-12.8052	0.2630	0.6905
11	1.3315	-17.4364	0.3739	0.8181
12	5.4557	-9.6166	0.4839	0.6171
13	1.0301	-10.6721	0.3643	0.8235
14	5.3569	-4.9885	0.4225	0.6353
15	-0.71664	-8.1946	0.4686	0.7447
16	4.1917	-10.0580	0.4398	0.6578
17	-1.2546	-7.2639	0.3280	0.5008
18	0.27051	-7.5120	0.4868	0.9902
19	-0.37244	-7.3169	0.3630	1.0154
20	5.1884	-4.1023	0.4962	0.5868
21	0.36488	-0.0295	0.4202	0.6166
22	-1.8732	-5.7948	0.4388	0.8130
23	-2.3007	-3.4411	0.5233	0.3902
24	1.5081	-10.2852	0.3819	0.6085
25	-0.78753	-2.6291	0.5681	0.8954
26	-0.90902	-7.8090	0.3712	0.6850
27	0.01729	-10.0884	0.5173	0.5261
28	-1.039	-9.7704	0.5536	0.5573
29	-1.9452	-10.3467	0.3975	0.4823
30	11.009	-8.5732	0.5742	0.8793

9.3 Listings

All the listing have been written exclusively by the author and for the purpose of this essay, with the exception of the two files "Vector2.h" and "Vector2.cpp" which were taken over from the author's former work, and also with the exception of the method for generating random Gaussian numbers in the file "Random.cpp" which was taken over from Dr. Everett. F. Carter Jr., Generating Gaussian Random Numbers: http://www.taygeta.com/random/gaussian.html

9.3.1 AutoTank.h

```
#ifndef AutoTank_H
#define AutoTank_H

// Description: Autonomous Tank Controlled by 2 CTRNNs
// Author:      Yvan Bourquin

class NeuralNetwork;
class Vector2;

#include "Tank.h"

class AutoTank : public Tank
{
public:
     // constructor
     AutoTank(const Vector2 &initialPosition,
          double alpha, const double *steeringCode,
          const double *aimingCode);

     // destructor
     virtual ~AutoTank();

     // update neural sensors, neural network and position
     virtual void update();

     // draw the neural network
     void drawNetwork(const Graphics *graphics) const;

     // get the code size (number of double) to encode
     // the steering controller
     static int getSteeringCodeSize();

     // get the code size (number of double) to encode
     // the aiming controller
     static int getAimingCodeSize();

private:
     NeuralNetwork *_steeringController;
     NeuralNetwork *_aimingController;
};

#endif
```

9.3.2 AutoTank.cpp

```
#include "AutoTank.h"
#include "NeuralNetwork.h"
#include "ProximitySensor.h"
#include "VisionSensor.h"
#include "Random.h"

const int NUM_HIDDEN = 4;
const int NUM_TURRET_HIDDEN = 4;

AutoTank::AutoTank(const Vector2 &initialPosition, double alpha,
                      const double *steeringCode, const double *aimingCode)
     : Tank(initialPosition, alpha)
{
     _steeringController = new NeuralNetwork(NUM_PROXIMITY_SENSORS,
          NUM_HIDDEN, NUM_BODY_MOTORS, steeringCode);
     _aimingController = new NeuralNetwork(NUM_VISION_SENSORS,
          NUM_TURRET_HIDDEN, NUM_TURRET_MOTORS, aimingCode);
}

AutoTank::~AutoTank()
{
     delete _steeringController;
     delete _aimingController;
}

int AutoTank::getSteeringCodeSize()
{
     return NeuralNetwork::getCodeSize(NUM_PROXIMITY_SENSORS,
          NUM_HIDDEN, NUM_BODY_MOTORS);
}

int AutoTank::getAimingCodeSize()
{
     return NeuralNetwork::getCodeSize(NUM_VISION_SENSORS,
          NUM_TURRET_HIDDEN, NUM_TURRET_MOTORS);
}

void AutoTank::update()
{
     // connect proximity sensors to steering controller
     for (int i = 0; i < NUM_PROXIMITY_SENSORS; i++)
          _steeringController->setInput(i, getSensorOutput(i));

     _steeringController->update();

     // connect steering controller output to motors + inject some noise
     setLeftSpeed(_steeringController->getOutput(0)
          + Random::getGaussian() / 50.0);
     setRightSpeed(_steeringController->getOutput(1)
          + Random::getGaussian() / 50.0);

     // connect vision sensors to aiming controller + inject some noise
     for (int j = 0; j < NUM_VISION_SENSORS; j++)
          _aimingController->setInput(j, getTurretSensorOutput(j)
          + Random::getGaussian() / 50.0);
     _aimingController->update();

     // connect aiming controller output to turret
     setTurretRotationSpeed((_aimingController->getOutput(0) -
          _aimingController->getOutput(1)) / 5.0);

     Tank::update();
}

void AutoTank::drawNetwork(const Graphics *graphics) const
{
     _steeringController->draw(graphics);
}
```

9.3.3 Genotype.h

```
#ifndef Genotype_H
#define Genotype_H

// Description: General-purpose genotype with mutation and crossover operations
// Author:      Yvan Bourquin

#include <stdio.h>

class Genotype
{
public:
     // constructor
     Genotype(int size);

     // copy constructor
     Genotype(const Genotype &);

     // destructor
     virtual ~Genotype();

     // assignment operator
     Genotype &operator = (const Genotype &);

     // mutation and crossover
     void hypersphereMutate();
     void locusMutate();
     Genotype crossover(const Genotype &other) const;

     // set/get fitness
     void setFitness(double fitness) { _fitness = fitness; }
     double getFitness() const { return _fitness; }

     // get array of floating points
     const double *getGenes() const { return _genes; }

     // read genotype from file
     void read(FILE *file);

     // write genotype to file
     void write(FILE *file) const;
private:
     double *_genes;   // genome
     int _size;        // genome length
     double _fitness;
};

#endif
```

9.3.4 Genotype.cpp

```
#include "Genotype.h"
#include "Random.h"
#include <math.h>
#include <assert.h>

Genotype::Genotype(int size)
     : _size(size), _fitness(0.0)
{
     _genes = new double[_size];

     // initialize with random uniform numbers in the range [0,1]
     for (int i = 0; i < _size; i++)
          _genes[i] = Random::getUniform();
}

Genotype::Genotype(const Genotype &other)
     : _size(other._size), _fitness(other._fitness)
{
     _genes = new double[_size];

     for (int i = 0; i < _size; i++)
          _genes[i] = other._genes[i];
}

Genotype &Genotype::operator = (const Genotype &other)
{
     // avoid crash in case of inadvertant: a = a
     if (&other == this)
          return *this;

     delete [] _genes;

     _size = other._size;
     _genes = new double[_size];
     _fitness = other._fitness;

     for (int i = 0; i < _size; i++)
          _genes[i] = other._genes[i];

     return *this;
}

Genotype::~Genotype()
{
     delete [] _genes;
}
```

```
void Genotype::hypersphereMutate()
{
     double length = Random::getGaussian() / 5.0;
     double *mutation = new double[_size];

     double sum = 0.0;
     for (int i = 0; i < _size; i++)
     {
          mutation[i] = Random::getUniform();
          sum += mutation[i] * mutation[i];
     }

     double ratio = length / sqrt(sum);
     for (i = 0; i < _size; i++)
          _genes[i] += mutation[i] * ratio;
}

// mutate a every gene with probability 0.04
void Genotype::locusMutate()
{
     for (int i = 0; i < _size; i++)
          if (Random::getUniform() < 0.04)
               _genes[i] += Random::getGaussian() / 4.0;
}

// single-point crossover
Genotype Genotype::crossover(const Genotype &other) const
{
     Genotype child(_size);

     // make sure we don't always start with the same guy
     const double *mom, *dad;
     if (Random::getInteger(2) == 0)
     {
          mom = this->_genes;
          dad = other._genes;
     }
     else
     {
          mom = other._genes;
          dad = this->_genes;
     }

     int locus = Random::getInteger(_size);

     for (int i = 0; i < _size; i++)
          if (i < locus)
               child._genes[i] = mom[i];
          else
               child._genes[i] = dad[i];

     return child;
}

void Genotype::write(FILE *file) const
{
     fprintf(file, "%d %.3f", _size, _fitness);

     for (int i = 0; i < _size; i++)
          fprintf(file, " %.3f", _genes[i]);
}

void Genotype::read(FILE *file)
{
     int size;
     fscanf(file, "%d", &size);

     assert(size == _size);

     fscanf(file, "%lf", &_fitness);

     for (int i = 0; i < _size; i++)
          fscanf(file, "%lf", &_genes[i]);
}
```

9.3.5 Graphics.h

```
#ifndef Graphics_H
#define Graphics_H

// Description: Graphic primitives
// Author:      Yvan Bourquin

class Vector2;
class Quad2;
class Line2;

class Graphics
{
public:
     // constructor
     Graphics();

     // before rendering: organize coodinate system
     void preRender() const;

     // change current RGB color
     void setColor(float red, float green, float blue) const;

     // draw a line
     void drawLine(double x1, double y1, double x2, double y2) const;
     void drawLine(const Vector2 &a, const Vector2 &b) const;
     void drawLine(const Line2 &line) const;

     // draw a quad (4 sided polygon) as alines
     void drawLineQuad(const Quad2 &q) const;

     // draw a filled quad
     void drawSolidQuad(const Quad2 &q) const;

     // change line width
     void setLineWidth(double width) const;
     // flush all the pending graphical operations
     void flush() const;
};

#endif
```

9.3.6 Graphics.cpp

```
#include "Graphics.h"
#include "Vector2.h"
#include "Quad2.h"
#include "Line2.h"
#include <glut.h>
#include <gl/gl.h>

Graphics::Graphics()
{
}

void Graphics::preRender() const
{
     // white background
     glClearColor(1, 1, 1, 0);
     glClear(GL_COLOR_BUFFER_BIT);

     // reset the model matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

     // define coordinate system
    glTranslatef(-0.98, -0.98, 0);
     glScaled(0.0140, 0.0245, 1);
}

void Graphics::flush() const
{
     glFlush();
}

void Graphics::setColor(float red, float green, float blue) const
{
     glColor3f(red, green, blue);
}

void Graphics::drawLine(double x1, double y1, double x2, double y2) const
{
     glBegin(GL_LINES);
          glVertex2d(x1, y1);
          glVertex2d(x2, y2);
     glEnd();
}

void Graphics::drawLine(const Vector2 &a, const Vector2 &b) const
{
     glBegin(GL_LINES);
          glVertex2d(a.x(), a.y());
          glVertex2d(b.x(), b.y());
    glEnd();
}

void Graphics::drawLine(const Line2 &line) const
{
    glBegin(GL_LINES);
        glVertex2d(line.a().x(), line.a().y());
        glVertex2d(line.b().x(), line.b().y());
    glEnd();
}

void Graphics::drawLineQuad(const Quad2 &q) const
{
    glBegin(GL_LINES);
        glVertex2d(q.a().x(), q.a().y());
        glVertex2d(q.b().x(), q.b().y());
        glVertex2d(q.b().x(), q.b().y());
        glVertex2d(q.c().x(), q.c().y());
        glVertex2d(q.c().x(), q.c().y());
        glVertex2d(q.d().x(), q.d().y());
        glVertex2d(q.d().x(), q.d().y());
        glVertex2d(q.a().x(), q.a().y());
    glEnd();
}

void Graphics::drawSolidQuad(const Quad2 &q) const
{
    glBegin(GL_QUADS);
        glVertex2d(q.a().x(), q.a().y());
        glVertex2d(q.b().x(), q.b().y());
        glVertex2d(q.b().x(), q.b().y());
        glVertex2d(q.c().x(), q.c().y());
        glVertex2d(q.c().x(), q.c().y());
        glVertex2d(q.d().x(), q.d().y());
        glVertex2d(q.d().x(), q.d().y());
        glVertex2d(q.a().x(), q.a().y());
    glEnd();
}

void Graphics::setLineWidth(double width) const
{
    glLineWidth((float)width);
}
```

9.3.7 Line2.h

```
#ifndef Line2_H
#define Line2_H

// Description: Abstraction of a two-dimensional line
// Author:      Yvan Bourquin

#include "Vector2.h"
#include <iostream.h>

class Line2
{
```

```
public:
     // constructors
     Line2() : _a(0, 0), _b(0, 0) {}
     Line2(const Vector2 &a, const Vector2 &b) : _a(a), _b(b) {}
     Line2(double x1, double y1, double x2, double y2) : _a(x1, y1), _b(x2, y2)
          {}

     // compute the distance from this line to the specified point
     // "infinite" indicated whether the line should be considered
     // as a line segment or an infinite line
     double distanceToPoint(const Vector2 &point, bool infinite = false) const;

     // return the slope m of the line which equation is in the form:
     // y = m * x + b
     double slope() const;

     // compute b, the height of the intersection of the line with the
     // y-axis: y = m * x + b
     double intercept() const;

     // line length
     double length() const;

     // translation
     void translate(const Vector2 &trans);

     // rotate line around origin
     // angle is given in radian
     void rotate(double angle);

     // compute and return the intersection point between two lines
     // if there is no intersection return NO_INTERSECT.
     Vector2 intersects(const Line2 &that) const;
     static const Vector2 NO_INTERSECT;

     // return read-only line endpoints
     const Vector2 &a() const { return _a; }
     const Vector2 &b() const { return _b; }

     // return assignable line endpoints
     Vector2 &a() { return _a; }
     Vector2 &b() { return _b; }

     // print line to output stream
     friend ostream &operator << (ostream &, const Line2 &);

private:
     Vector2 _a, _b;  // line endpoints
};

inline double Line2::slope() const
{
     return (_b.y() - _a.y()) / (_b.x() - _a.x());
}

inline double Line2::intercept() const
{
     return _a.y() + (-_a.x() / (_b.x() - _a.x())) * (_b.y() - _a.y());
}

inline void Line2::translate(const Vector2 &trans)
{
     _a += trans;
     _b += trans;
}

inline ostream &operator << (ostream &os, const Line2 &l)
{
     cout << "[" << l._a << l._b << "]";
     return os;
}

#endif
```

9.3.8 Line2.cpp

```
#include "Line2.h"

const Vector2 Line2::NO_INTERSECT(-999999, -999999);

// Compute the length of vector (x, y).
double Line2::length() const
{
     return (_b - _a).length();
}

void Line2::rotate(double angle)
{
     _a.rotate(angle);
     _b.rotate(angle);
}

Vector2 Line2::intersects(const Line2 &that) const
{
    // compute first edge equation
    double c1 = intercept();
    double m1 = slope();

    // compute second edge equation
    double c2 = that.intercept();
    double m2 = that.slope();

    // are the lines parallel ?
    if (m1 == m2)
      return NO_INTERSECT;

     double x1 = _a.x();
     double x2 = _b.x();
     double x3 = that._a.x();
     double x4 = that._b.x();
     double y1 = _a.y();
     double y2 = _b.y();
     double y3 = that._a.y();
     double y4 = that._b.y();
```

```
// make sure x1 < x2
if (x1 > x2) {
  double temp = x1;
  x1 = x2;
  x2 = temp;
}

// make sure x3 < x4
if (x3 > x4) {
  double temp = x3;
  x3 = x4;
  x4 = temp;
}

 // make sure y1 < y2
if (y1 > y2) {
  double temp = y1;
  y1 = y2;
  y2 = temp;
}

// make sure y3 < y4
if (y3 > y4) {
  double temp = y3;
  y3 = y4;
  y4 = temp;
}

 // intersection point in case of infinite lines
 double x;
 double y;

 // infinite slope m1
 if (x1 == x2)
 {
      x = x1;
      y = m2 * x1 + c2;
      if (x > x3 && x < x4 && y > y1 && y <y2)
           return Vector2(x, y);
      else
           return NO_INTERSECT;
 }

 // infinite slope m2
 if (x3 == x4)
 {
      x = x3;
      y = m1 * x3 + c1;
      if (x > x1 && x < x2 && y > y3 && y < y4)
           return Vector2(x, y);
      else
           return NO_INTERSECT;
 }

 // compute lines intersection point
x = (c2 - c1) / (m1 - m2);
```

```
    // see whether x in in both ranges [x1, x2] and [x3, x4]
    if (x > x1 && x < x2 && x > x3 && x < x4)
         return Vector2(x, m1 * x + c1);

     return NO_INTERSECT;
}
```

9.3.9 Main.cpp

```
// Description: Tank Wars!
// Author:      Yvan Bourquin

#include <glut.h>
#include <gl/gl.h>
#include <stdlib.h>
#include <time.h>

#include "Simulator.h"
#include "Graphics.h"
#include "ProximitySensor.h"
#include "VisionSensor.h"

static Simulator *simulator = 0;
static Graphics *graphics = 0;
static bool visible = true;
static double delay = 50;
static int mode = 0;

void displayFunction(void)
{
     if (visible)
     {
          switch (mode)
          {
          case 0:
               simulator->draw(graphics);
               break;
          case 1:
               simulator->drawNetwork(graphics);
               break;
          }

          glutSwapBuffers();
     }
}

void reshapeFunction(int width,int height)
{
     glViewport(0, 0, width, height);
     displayFunction();
}

void keyboardFunction(unsigned char key, int x, int y)
{
     if (key >= '0' && key <= '9')
          simulator->numberPressed(key - '0');
     switch (key)
     {
     case 'p':
          ProximitySensor::toggleVisibility();
          break;

     case 'v':
          VisionSensor::toggleVisibility();
          break;

     case '+':
          delay /= 2;
          break;

     case '-':
          delay *= 2;
          break;
     }
}

void specialFunction(int key,int x,int y)
{
     switch (key)
     {
     case GLUT_KEY_F1:
          mode = 0;
          break;

     case GLUT_KEY_F2:
          mode = 1;
          break;
     }
}

void idleFunction()
{
     if (! visible)
          simulator->update();
}

void timerFunction(int value)
{
     if (visible)
     {
          glutTimerFunc(delay, &timerFunction, 0);
          simulator->update();
          glutPostRedisplay();
     }
}

void visibilityFunction(int state)
{
     switch (state)
     {
     case GLUT_VISIBLE:
          glutTimerFunc(delay, &timerFunction, 0);
          visible = true;
          break;

     case GLUT_NOT_VISIBLE:
          visible = false;
          break;
     }
}

#include "Quad2.h"

int main(int argc, char* argv[])
{
     // initialize random number generator
     srand((unsigned)time(NULL));

     glutInit(&argc, argv);

     // get simulator instance
     simulator = Simulator::instance();

     // create graphic context
     graphics = new Graphics();

     // create and position window
     glutInitWindowSize(700, 400);
     glutInitWindowPosition(0, 0);
     int win = glutCreateWindow("Tanks!");

     // install callbacks
     glutDisplayFunc(&displayFunction);
     glutReshapeFunc(&reshapeFunction);
     glutKeyboardFunc(&keyboardFunction);
     glutSpecialFunc(&specialFunction);
     glutVisibilityFunc(&visibilityFunction);
     glutIdleFunc(&idleFunction);
     glutTimerFunc(0, &timerFunction, 0);

     // start event processing
     glutMainLoop();

     return 0;
}
```

9.3.10 Matrix.h

```
#ifndef Matrix_H
#define Matrix_H

// Description: General purpose N x M matrix and associated operators
// Author:      Yvan Bourquin

#include <assert.h>
#include <iostream.h>
#include "Vector.h"

class Matrix
```

```
{
public:
     // constructs a uninitialized N x M matrix
     Matrix(int lines, int columns);

     // copy constructor
     Matrix(const Matrix &other);

     // destructor
     ~Matrix();

     // return number of lines in the matrix
     int size() const { return L; }

     // matrix line access
     const Vector &operator [] (int index) const;
     Vector &operator [] (int index);

     // assignement operator
     Matrix &operator = (const Matrix &);

     // unary + and - operator
     Matrix operator + () const { return *this; }
    Matrix operator - () const { return 0.0 - *this; }

     // matrix-matrix addition and substraction
     Matrix operator + (const Matrix &) const;
     Matrix operator - (const Matrix &) const;

     // matrix-vector multiplication
     Vector operator * (const Vector &v) const;

     // matrix-scalar operations
     Matrix operator + (double) const;
     Matrix operator - (double) const;
     Matrix operator * (double) const;
     Matrix operator / (double) const;

     // scalar matrix operations
     friend Matrix operator + (double, const Matrix &);
     friend Matrix operator - (double, const Matrix &);
     friend Matrix operator * (double, const Matrix &);
     friend Matrix operator / (double, const Matrix &);

     // print matrix to output stream
     friend ostream &operator << (ostream &os, const Matrix &m);

private:
     Vector *a;
     const int L;
};

inline ostream &operator << (ostream &os, const Matrix &m)
{
     os << "[ " << endl;
     for (int l = 0; l < m.L; l++)
          os << m.a[l] << endl;
     os << "]";

    return os;
}

inline const Vector &Matrix::operator [] (int index) const
{
     assert(index >= 0 && index < L);
     return a[index];
}

inline Vector &Matrix::operator [] (int index)
{
     assert(index >= 0 && index < L);
     return a[index];
}

inline Matrix::Matrix(int lines, int columns)
     : L(lines)
{
     Vector::setDefaultSize(columns);
     a = new Vector[L];
}

inline Matrix::Matrix(const Matrix &other)
     : L(other.L)
{
     a = new Vector[L];
     for (int l = 0; l < L; l++)
          a[l] = other.a[l];
}

inline Matrix::~Matrix()
{
     delete [] a;
}

inline Matrix &Matrix::operator = (const Matrix &other)
{
     assert(L == other.L);

     for (int l = 0; l < L; l++)
          a[l] = other.a[l];

     return *this;
}

inline Matrix Matrix::operator + (const Matrix &other) const
{
     assert(L == other.L);

     Matrix b(L, other[0].size());
     for (int l = 0; l < L; l++)
          b.a[l] = a[l] + other.a[l];

     return b;
```

```
}

inline Matrix Matrix::operator - (const Matrix &other) const
{
     assert(L == other.L);

     Matrix b(L, other[0].size());
     for (int l = 0; l < L; l++)
          b.a[l] = a[l] - other.a[l];

     return b;
}

inline Vector Matrix::operator * (const Vector &v) const
{
     assert(a[0].size() == v.size());

     Vector b(L);
     for (int l = 0; l < L; l++)
          b[l] = a[l].dot(v);

     return b;
}

inline Matrix Matrix::operator + (double d) const
{
     Matrix b(L, a[0].size());
     for (int l = 0; l < L; l++)
          b.a[l] = a[l] + d;

     return b;
}

inline Matrix Matrix::operator - (double d) const
{
     Matrix b(L, a[0].size());
     for (int l = 0; l < L; l++)
          b.a[l] = a[l] - d;

     return b;
}

inline Matrix Matrix::operator * (double d) const
{
     Matrix b(L, a[0].size());
     for (int l = 0; l < L; l++)
          b.a[l] = a[l] * d;

     return b;
}

inline Matrix Matrix::operator / (double d) const
{
     Matrix b(L, a[0].size());
     for (int l = 0; l < L; l++)
          b.a[l] = a[l] / d;
     return b;
}

inline Matrix operator + (double d, const Matrix &a)
{
     Matrix b(a.L, a[0].size());
     for (int l = 0; l < a.L; l++)
          b.a[l] = d + a.a[l];

     return b;
}

inline Matrix operator - (double d, const Matrix &a)
{
     Matrix b(a.L, a[0].size());
     for (int l = 0; l < a.L; l++)
          b.a[l] = d - a.a[l];

     return b;
}

inline Matrix operator * (double d, const Matrix &a)
{
     Matrix b(a.L, a[0].size());
     for (int l = 0; l < a.L; l++)
          b.a[l] = d * a.a[l];

     return b;
}

inline Matrix operator / (double d, const Matrix &a)
{
     Matrix b(a.L, a[0].size());
     for (int l = 0; l < a.L; l++)
          b.a[l] = d / a.a[l];

     return b;
}

#endif
```

9.3.11 NeuralNetwork.h

```
#ifndef NeuralNetwork_H
#define NeuralNetwork_H

// Description: General purpose CTRNN type neural network with
//              customizable number of inputs, hidden and output neurons
//              Biases, time constant and connection weights are initialized
//              in fixed ranges
// Author:      Yvan Bourquin

#include "Vector.h"
#include "Matrix.h"

class Graphics;
class NeuralNetwork
{
public:
     // constructor: create a CTRNN  with N inputs, H hidden neurons,
     // M output neurons, and initialize it with genotype "code"
     NeuralNetwork(int n, int h, int m, const double *code);

     // set sensory input I
     void setInput(int index, double value) { I[index] = value; }

     // compute new network state Y(t+1)=Y(Y(t),I(t))
     void update();

     // get network state Y
     double getOutput(int index) const { return Y[index]; }

     // draw neural network current state
     void draw(const Graphics *graphics) const;

     // compute required genotype size for N, H and M
     static int getCodeSize(int N, int H, int M);

private:
     const int N; // number of inputs
     const int H; // number of hidden neurons
     const int M; // number of output neurons

     Vector I;  // input vector
     Vector Yh; // hidden layer neuron states
     Vector Y;  // ouput layer neuron states
     Vector Th; // time constants of hidden layer
     Vector To; // time constants of output layer
     Vector Bh; // biases of hidden layer
     Vector Bo; // biases of output layer

     Matrix U; // input to hidden neuron connections
     Matrix V; // hidden neurons recurrent and self-connections
     Matrix W; // hidden to output neuron connections

     // local methods
     void setCode(const double *code);
     void drawNeuron(const Graphics *graphics, double x,
          double y, double activity) const;
};

#endif
```

9.3.12 NeuralNetwork.cpp

```
#include "NeuralNetwork.h"
#include <assert.h>
#include "Quad2.h"
#include "Graphics.h"

// intial parameter ranges
static const double MIN_WEIGHT = -5.0;
static const double MAX_WEIGHT =  5.0;
static const double MIN_BIAS =   -2.0;
static const double MAX_BIAS =    2.0;
static const double MIN_TAU =     1.0;
static const double MAX_TAU =    10.0;

NeuralNetwork::NeuralNetwork(int n, int h, int m, const double *code)
: N(n), H(h), M(m), I(n), Yh(h), Y(m), Th(h), To(m), Bh(h),
     Bo(h), U(h, n), V(h, h), W(m, h)
{
     // intialize to zero:
     I.zero();   // inputs
     Yh.zero();  // hidden neuron states
     Y.zero();   // output neuron states

     setCode(code);
}

// compute sigmoid for a vector
Vector sigmoid(const Vector &X)
{
     Vector Y(X.size());
     for (int l = 0; l < X.size(); l++)
          Y[l] = 1.0 / (1.0 + exp(-X[l]));

     return Y;
}

// compute new neural network states
// the code is very short here because + - * / and = are overloaded
// operators that handle matrix and vector operations
void NeuralNetwork::update()
{
     // compute output neuron states
     Y = Y + 0.1 / To * (-Y + W * sigmoid(Yh - Bo));

     // compute hidden neuron states
     Yh = Yh + 0.1 / Th * (-Yh + V * sigmoid(Yh - Bh) + U * I);
}

// draw neurons and synaptic connections
void NeuralNetwork::draw(const Graphics *graphics) const
{
     graphics->setLineWidth(2.0);
     graphics->setColor(0, 0, 0);

     for (int n = 0; n < N; n++)
     {
          double x1 = 35;
          double y1 = (n + 1) * 80 / (N + 1);

          for (int h = 0; h < H; h++)
          {
               double x2 = 70;
               double y2 = (h + 1) * 80 / (H + 1);
               graphics->drawLine(x1, y1, x2, y2);
          }
     }
     for (int h = 0; h < H; h++)
     {
          double x1 = 70;
          double y1 = (h + 1) * 80 / (H + 1);

          for (int m = 0; m < M; m++)
          {
               double x2 = 105;
               double y2 = (m + 1) * 80 / (M + 1);
               graphics->drawLine(x1, y1, x2, y2);
          }
     }
     double dy = 80 / (N + 1);

     for (n = 0; n < N; n++)
     {
          double x = 35;
          double y = (n + 1) * 80 / (N + 1);
          drawNeuron(graphics, x, y, I[n]);
     }

     for (h = 0; h < H; h++)
     {
          double x = 70;
          double y = (h + 1) * 80 / (H + 1);
          drawNeuron(graphics, x, y, Yh[h]);
     }

     for (int m = 0; m < M; m++)
     {
          double x = 105;
          double y = (m + 1) * 80 / (M + 1);
          drawNeuron(graphics, x, y, Y[m]);
     }
}

// draw a single neuron
void NeuralNetwork::drawNeuron(const Graphics *graphics, double x,
                               double y, double activity) const
{
     float a = (float)activity;
     Quad2 q(2, 2, 2, -2, -2, -2, -2, 2);
     q.translate(x, y);
     graphics->setColor(a, a, a);
     graphics->drawSolidQuad(q);
     graphics->setColor(0, 0, 0);
     graphics->drawLineQuad(q);
}

int NeuralNetwork::getCodeSize(int N, int H, int M)
{
     return (3 * H + H * N + H * H + M + M * H) / 2;
}

// genotype to controller decoding:
// bilateral symmetry decoding and initial parameter scaling are carried out
// in this function
void NeuralNetwork::setCode(const double *code)
{
     const double *c = code;

     // hidden neurons
     for (int h = 0; h < H / 2; h++)
     {
          int k = H - h - 1;

          Bh[h] = Bh[k] = *c++ * (MAX_BIAS - MIN_BIAS) + MIN_BIAS;

          double t = *c++ * (MAX_TAU - MIN_TAU) + MIN_TAU;
          if (t < 0.2)
               t = 0.2;

          Th[h] = Th[k] = t ;
          Bo[h] = Bo[k] = *c++ * (MAX_BIAS - MIN_BIAS) + MIN_BIAS;

          for (int n = 0; n < N; n++)
               U[h][n] = U[k][N - n - 1] = *c++ * (MAX_WEIGHT - MIN_WEIGHT)
                    + MIN_WEIGHT;

          for (int g = 0; g < H; g++)
               V[h][g] = V[k][H - g - 1] = *c++ * (MAX_WEIGHT - MIN_WEIGHT)
                    + MIN_WEIGHT;
     }

     // motor neurons
     for (int m = 0; m < M / 2; m++)
     {
          int k = M - m - 1;

          double t = *c++ * (MAX_TAU - MIN_TAU) + MIN_TAU;
          if (t < 0.2)
               t = 0.2;

          To[m] = To[k] = t * (MAX_TAU - MIN_TAU) + MIN_TAU;

          for (int h = 0; h < H; h++)
               W[m][h] = W[k][H - h -1] = *c++ * (MAX_WEIGHT - MIN_WEIGHT)
                    + MIN_WEIGHT;
     }

     assert(c - code == getCodeSize(N, H, M));
}
```

9.3.13 Obstacle.h

```
#ifndef Obstacle_H
#define Obstacle_H

// Description: Abstract base class for quad-based collision detection
// Author:      Yvan Bourquin

class Graphics;
```

```
#include "Quad2.h"

class Obstacle {
public:
     // get collision detection bounds
     const Quad2 &getBounds() const { return _bounds; }

     // draw bounds
     virtual void draw(const Graphics *graphics) const = 0;

protected:
     Quad2 _bounds;
};

#endif
```

9.3.14 Population.h

```
#ifndef Population_H
#define Population_H

// Description: Genotype population with sorting and reproduction
// Author:      Yvan Bourquin

#include "Genotype.h"

class Population
{
public:
     // constructor
     Population(int popSize, int genSize);

     // destructor
     virtual ~Population();

     // load/save from file
     void save(const char *filename) const;
     void load(const char *filename);

     // sort population from most fit to least fit individual
     void sort();

     // change generation
     // precondition:
     // population must be sorted before this function is called
     void reproduce();

     // current generation number
     int getGeneration() const { return _generation; }

     // get genetic code
     const double *getGenes(int index) const
          { return _genotypes[index]->getGenes(); }

     // get/set fitness
     void setFitness(int index, double fitness)
          { _genotypes[index]->setFitness(fitness); }
```

```
	double getFitness(int index) const
		{ return _genotypes[index]->getFitness(); }

private:
	Genotype **_genotypes; // genotypes
	int _size;             // population size
	int _genotypeSize;     // size of each genotype
	int _generation;       // current generation
	double *_meanFitness;  // vector of mean fitness
	double *_bestFitness;  // vector of best fitness

	const Genotype *chooseParent() const;
};

#endif
```

9.3.15 Population.cpp

```
#include "Population.h"
#include "Random.h"
#include <stdlib.h>
#include <fstream.h>
#include <assert.h>

Population::Population(int populationSize, int genotypeSize)
{
	_size = populationSize;
	_genotypeSize = genotypeSize;
	_generation = 0;
	_genotypes = new Genotype*[_size];
	_meanFitness = new double[5000];
	_bestFitness = new double[5000];

	for (int i = 0; i < _size; i++)
		_genotypes[i] = new Genotype(genotypeSize);
}

Population::~Population()
{
	delete [] _bestFitness;
	delete [] _meanFitness;

	for (int i = 0; i < _size; i++)
		delete _genotypes[i];

	delete [] _genotypes;
}

void Population::load(const char *filename)
{
	FILE *file = fopen(filename, "r");
	if (! file)
	{
		printf("failed to open for reading: %s\n", filename);
		return;
	}
```

```
     int i;
     fscanf(file, "%d", &_generation);
     for (i = 0; i < _generation; i++)
          fscanf(file, "%lf", &_bestFitness[i]);

     for (i = 0; i < _generation; i++)
          fscanf(file, "%lf", &_meanFitness[i]);

     int size;
     fscanf(file, "%d", &size);

     assert(size == _size);

     for (i = 0; i < _size; i++)
          _genotypes[i]->read(file);

     fclose(file);
}

void Population::save(const char *filename) const
{
     FILE *file = fopen(filename, "w");
     if (! file)
     {
          printf("failed to open for writing: %s\n", filename);
          return;
     }

     int i;
     fprintf(file, "%d\n", _generation + 1);

     for (i = 0; i < _generation + 1; i++)
          fprintf(file, "%.3f ", _bestFitness[i]);

     fprintf(file, "\n");

     for (i = 0; i < _generation + 1; i++)
          fprintf(file, "%.3f ", _meanFitness[i]);

     fprintf(file, "\n%d\n", _size);

     for (i = 0; i < _size; i++)
     {
          _genotypes[i]->write(file);
          fprintf(file, "\n");
     }

     fclose(file);
}

static int compare(const void *a, const void *b)
{
     return (*(Genotype**)a)->getFitness() >
          (*(Genotype**)b)->getFitness() ? -1 : +1;
}

const Genotype *Population::chooseParent() const
```

```
{
     while (true)
     {
          int index = Random::getInteger(_size);
          if (index <= Random::getInteger(_size))
               return _genotypes[index];
     }
}

void Population::sort()
{
     // sort for rank selection
     qsort(_genotypes, _size, sizeof(Genotype*), &compare);

     for (int k = 0; k < _size; k++)
          cout << _genotypes[k]->getFitness() << " ";

     cout << endl;

     _bestFitness[_generation] = _genotypes[0]->getFitness();

     double sum = 0.0;
     for (int i = 0; i < _size; i++)
          sum += _genotypes[i]->getFitness();

     _meanFitness[_generation] = sum / _size;
}

void Population::reproduce()
{
     Genotype **_nextGeneration = new Genotype*[_size];
     for (int i = 0; i < _size; i++)
     {
          Genotype *child = new Genotype(_genotypeSize);

          if (i < 5)
               *child = *_genotypes[i];    // elitism
          else
          {
               const Genotype *mom = chooseParent();
               if (Random::getUniform() < 0.05)
               {
                    const Genotype *dad = chooseParent();
                    *child = mom->crossover(*dad); // sexual reproduction
               }
               else
               {
                    *child = *mom;  // asexual reproduction
                    child->hypersphereMutate();
               }
          }

          child->setFitness(0.0);
          _nextGeneration[i] = child;
     }

     for (int j = 0; j < _size; j++)
```

```
          delete _genotypes[j];

     delete [] _genotypes;

     _genotypes = _nextGeneration;
     _generation++;
}
```

9.3.16 ProximitySensor.h

```
#ifndef ProximitySensor_H
#define ProximitySensor_H

// Description: Simulated infrared sensor
// Author:      Yvan Bourquin

class Tank;
class Graphics;

#include "Vector2.h"
#include "Line2.h"

class ProximitySensor
{
public:
     // constructor:
     // tank: the tank which this sensor belongs to
     // x, y: position of the sensor in the tank coordinate system
     // dx, dy: direction of the sensor ray in tank coordinate system
     ProximitySensor(const Tank *tank, double x, double y,
          double dx, double dy);

     // draw sensor ray as a line
     // The color changes when the ray intesects an obstacle
     void draw(const Graphics *graphics) const;

     // check intersection and update output
     void update();

     // return a number in the range [0,1] inversely proportional
     // to the distance to the nearest obstacle.
     // returns 0.0 if there is no obstacle in the sensed range
     // returns 1.0 if there is an obstacle right in front of sensor
     double getOutput() const;

     // toggle visibility flag of all proximity sensors
     static void toggleVisibility() { _visible = ! _visible; }

private:
     const Tank *_tank;
     Vector2 _position;
     Vector2 _direction;
     Line2 _ray;

     static bool _visible;
};
```

```
#endif
```

9.3.17 ProximitySensor.cpp

```
#include "ProximitySensor.h"
#include "Graphics.h"
#include "Tank.h"
#include "Simulator.h"

static const double MAX_RANGE = 20.0;

bool ProximitySensor::_visible = true;

ProximitySensor::ProximitySensor(const Tank *tank, double x, double y,
                                  double dx, double dy)
: _tank(tank), _position(x, y)
{
     _direction = Vector2(dx, dy).normalized();
}

void ProximitySensor::draw(const Graphics *graphics) const
{
     if (! _visible)
          return;

     graphics->setLineWidth(1.0);
     if (_ray.length() < MAX_RANGE)
          graphics->setColor(1, 0, 0);  // red
     else
          graphics->setColor(0, 1, 0);  // green

     graphics->drawLine(_ray);
}

double ProximitySensor::getOutput() const
{
     return 1.0 - _ray.length() / MAX_RANGE;
}

void ProximitySensor::update()
{
     // compute new ray position and direction
     _ray = Line2(_position, _position + _direction * MAX_RANGE);
     _ray.rotate(-_tank->getAlpha());
     _ray.translate(_tank->getPosition());

     Simulator *simulator = Simulator::instance();
     int N = simulator->getNumObstacles();

     // for each potential obstacle
     for (int i = 0; i < N; i++)
     {
          const Obstacle *obstacle = simulator->getObstacle(i);

          // don't check collision with own tank
          if (obstacle != _tank)
          {
              Quad2 bounds = obstacle->getBounds();
              for (int j = 0; j < 4; j++)
              {
                   // check intersection
                   Vector2 intersect = _ray.intersects(bounds.getLine(j));

                   if (intersect != Line2::NO_INTERSECT)
                   {
                        double distance = _ray.a().distance(intersect);
                        if (distance < _ray.length())
                             _ray.b() = intersect;
                   }
              }
         }
     }
}
```

9.3.18 Quad2.h

```
#ifndef Quad2_H
#define Quad2_H

// Description: Four-side polygon used as a primitive for collision
//              detection and for graphics.
// Author:      Yvan Bourquin

#include "Vector2.h"
#include "Line2.h"

class ostream;

class Quad2
{
public:
     // constructors
     Quad2() : _a(-1, 1), _b(1, 1), _c(1, -1), _d(-1, -1) {}
     Quad2(const Quad2 &q) : _a(q._a), _b(q._b), _c(q._c), _d(q._d) {}
     Quad2(const Vector2 &a, const Vector2 &b,
          const Vector2 &c, const Vector2 &d);
     Quad2(double ax, double ay, double bx, double by,
          double cx, double cy, double dx, double dy);

     // assignement operator
     Quad2 &operator = (const Quad2 &q);

     // geometrical operations
     void translate(Vector2 t);
     void translate(double tx, double ty);
     void rotate(double angle);

     // read-only members access
     const Vector2 &a() const { return _a; }
     const Vector2 &b() const { return _b; }
     const Vector2 &c() const { return _c; }
     const Vector2 &d() const { return _d; }

     // return two quad corners as a Line2 object
     Line2 getLine(int index) const;

     // intersection detection
     bool intersects(const Quad2 &quad) const;
     bool intersects(const Line2 &line) const;

     // print quad to output stream
     friend ostream &operator << (ostream &, const Quad2 &);

private:
     Vector2 _a, _b, _c, _d;
};

inline Quad2 &Quad2::operator = (const Quad2 &q)
{
    _a = q._a;
     _b = q._b;
     _c = q._c;
     _d = q._d;
    return *this;
};

#endif
```

9.3.19 Quad2.cpp

```
#include "Quad2.h"

Quad2::Quad2(const Vector2 &a, const Vector2 &b,
               const Vector2 &c, const Vector2 &d)
: _a(a), _b(b), _c(c), _d(d) {}

Quad2::Quad2(double ax, double ay, double bx, double by,
               double cx, double cy, double dx, double dy)
: _a(ax, ay), _b(bx, by), _c(cx, cy), _d(dx, dy) {}

void Quad2::translate(Vector2 t)
{
     _a += t;
     _b += t;
     _c += t;
     _d += t;
}

void Quad2::translate(double tx, double ty)
{
     Vector2 t(tx, ty);
     _a += t;
     _b += t;
     _c += t;
     _d += t;
}

void Quad2::rotate(double angle)
{
     _a.rotate(angle);
     _b.rotate(angle);
     _c.rotate(angle);
     _d.rotate(angle);
}

Line2 Quad2::getLine(int index) const
{
     switch (index)
     {
     case 0: return Line2(_a, _b);
     case 1: return Line2(_b, _c);
     case 2: return Line2(_c, _d);
     case 3: return Line2(_d, _a);
     }

     return Line2(_d, _a);
}

bool Quad2::intersects(const Quad2 &that) const
{
     for (int i = 0; i < 4; i++)
          for (int j = 0; j < 4; j++) {
               Line2 a = getLine(i);
               Line2 b = that.getLine(j);
               if (a.intersects(b) != Line2::NO_INTERSECT)
                    return true;
          }

     return false;
}

bool Quad2::intersects(const Line2 &b) const
{
     for (int i = 0; i < 4; i++)
     {
          Line2 a = getLine(i);
          if (a.intersects(b) != Line2::NO_INTERSECT)
               return true;
     }

     return false;
}

ostream &operator << (ostream &os, const Quad2 &q)
{
     cout << "[" << q._a << q._b << q._c << q._d << "]";
     return os;
}
```

9.3.20 Random.h

```
#ifndef Random_H
#define Random_H

// Description: Random number utilities
// Author:      Yvan Bourquin
//              Implementation of getGaussian()
//              taken over from Dr. Everett F. Carter Jr.
//              http://www.taygeta.com/random/gaussian.htm

class Random
{
public:
     // return random number between [0;1] from a uniform
     // distribution
     static double getUniform();

     // return random integer number between [0;max-1] from a
     // uniform distribution
     static int getInteger(int max);

     // return random number from a Gaussian distribution with
     // mean 0 and standard deviation 1
     static double getGaussian();

private:
     Random() {} // disabled constructor
};

#endif
```

9.3.21 Random.cpp

```
#include "Random.h"
#include <math.h>
#include <stdlib.h>

const double PI = 3.1415926535;

double Random::getUniform()
{
     return (double)rand() / (double)RAND_MAX;
}

double Random::getGaussian()
{
     static bool flag = true;
     static double y2;

     if (flag)
     {
          double x1, x2, w;
          do
          {
               x1 = 2.0 * getUniform() - 1.0;
               x2 = 2.0 * getUniform() - 1.0;
               w = x1 * x1 + x2 * x2;
          } while (w >= 1.0);

          w = sqrt((-2.0 * log(w)) / w);
          double y1 = x1 * w;
          y2 = x2 * w;

          flag = false;
          return y1;
     }

     flag = true;

     return y2;
}

int Random::getInteger(int max)
{
     return rand() % max;
}
```

9.3.22 Shell.h

```
#ifndef Shell_H
#define Shell_H

// Description: Simulated tank gun shell
// Author:      Yvan Bourquin

#include "Line2.h"
#include "Vector2.h"

class Graphics;
class Tank;

class Shell
{
public:
     // create shell with initial position and orientation
     Shell(const Tank *tank, const Vector2 &position,
          double directionAngle);

     // move shell of one step
     void update();

     // draw shell as a black this line
     void draw(const Graphics *graphics) const;

     // set to true after the shell has collided with an obstacle
     bool isExploded() const { return _exploded; }

private:
     const Tank *_tank;   // tank that fired the shell
     Line2 _body;         // shell body
     Vector2 _direction;  // flight direction
     bool _exploded;      // collided
};

#endif
```

9.3.23 Shell.cpp

```
#include "Shell.h"

#include "Simulator.h"
```

```
#include "Obstacle.h"
#include "Quad2.h"
#include "Graphics.h"

static const Line2 BODY(-2, 0, 2, 0); // heading east
static const double SPEED = 3.0;      // shorter than body length

Shell::Shell(const Tank *tank, const Vector2 &position,
               double directionAngle)
     : _tank(tank), _direction(1, 0), _exploded(false)
{
     _body = BODY;
     _body.rotate(-directionAngle);
     _body.translate(position);

     _direction.rotate(-directionAngle);
     _direction.normalize();
     _direction = _direction * SPEED;
}

void Shell::update()
{
     // fly
     _body.translate(_direction);

     Simulator *simulator = Simulator::instance();

     // for every obstacle in the simulator ...
     for (int i = 0; i < simulator->getNumObstacles(); i++)
     {
          Obstacle *obstacle = simulator->getObstacle(i);

          // don't check collision with own tanks
          if (obstacle != (Obstacle*)_tank)
          {
               // check collision with other object
               const Quad2 &bounds = obstacle->getBounds();
               if (bounds.intersects(_body))
                    _exploded = true;
          }
     }
}

void Shell::draw(const Graphics *graphics) const
{
     graphics->setColor(0, 0, 0);
     graphics->setLineWidth(3.0);
     graphics->drawLine(_body);
}
```

9.3.24 Simulator.h

```
#ifndef Simulator_H
#define Simulator_H

// Description: Simulator for tanks with a rectangular arena and walls
// Author:      Yvan Bourquin
```

```
class Tank;
class AutoTank;
class Graphics;
class Wall;
class Obstacle;
class Population;

class Simulator
{
public:
     // draw simulation in normal mode
     void draw(const Graphics *graphics) const;

     // draw simulation in neural network mode
     void drawNetwork(const Graphics *graphics) const;

     // update position of every object
     void update();

     // a numerical key was pressed
     void numberPressed(int num);

     // access obstacles (walls and tanks)
     int getNumObstacles() const;
     Obstacle *getObstacle(int index) const;

     // access tanks of a group
     int getNumTanks() const;
     Tank *getTank(int index) const;

     // return unique simulator instance
     static Simulator *instance();

     // number of time steps in a single fight
     static const int NUM_UPDATES;

private:
     Wall **_walls;
     AutoTank **_tanks;
     Population *_driverPopulation; // population of steering genotypes
     Population *_gunnerPopulation; // population of aiming genotypes
     int _groupCount;               // group number of current generation
     int _fightCount;               // fight number of the current group
     int _updateCount;              // update number of the current fight
     double *_steeringFitness;      // temporary array for computing mean
     double *_aimingFitness;        // temporary array for computing mean

     // disabled constructor and destructor
     Simulator();
     virtual ~Simulator();

     // local methods
     void startGeneration();
     void endGeneration();
     void startGroup();
     void endGroup();
     void startFight();
     void endFight();
};

#endif
```

9.3.25 Simulator.cpp

```
#include "Simulator.h"

#include "AutoTank.h"
#include "Wall.h"
#include "Graphics.h"
#include "Population.h"
#include "Random.h"

const int Simulator::NUM_UPDATES = 4000; // # of time steps in a fight
static const int POPULATION_SIZE = 100;  // # of genotypes in population
static const int GROUP_SIZE = 2;   // # of protagonists in a fight
static const int NUM_FIGHTS = 5;   // # of fights to average fitness
static const int NUM_WALLS = 5;    // # of walls in the arena

static const double PI = 3.1415926535;

static const char *DRIVERS_POPULATION_FILE = "drivers.txt";
static const char *GUNNERS_POPULATION_FILE = "gunners.txt";

static Simulator *theInstance = NULL;

Simulator *Simulator::instance()
{
     if (! theInstance)
          theInstance = new Simulator();

     return theInstance;
}

Simulator::Simulator()
{
     _walls = new Wall*[NUM_WALLS];

     _walls[0] = new Wall(140, 80, 140, 0, 0, 0, 0, 80);
     _walls[1] = new Wall(40, 60, 40, 40, 20, 20, 20, 60);
     _walls[2] = new Wall(100, 80, 80, 60, 60, 60, 60, 80);
     _walls[3] = new Wall(80, 20, 80, 0, 40, 0, 60, 20);
     _walls[4] = new Wall(120, 60, 120, 20, 100, 20, 100, 40);

     _tanks = new AutoTank*[GROUP_SIZE];

     _driverPopulation = new Population(POPULATION_SIZE,
          AutoTank::getSteeringCodeSize());
     _driverPopulation->load(DRIVERS_POPULATION_FILE);

     _gunnerPopulation = new Population(POPULATION_SIZE,
          AutoTank::getAimingCodeSize());
     _gunnerPopulation->load(GUNNERS_POPULATION_FILE);
     _steeringFitness = new double[GROUP_SIZE];
     _aimingFitness = new double[GROUP_SIZE];

     startGeneration();
     startGroup();
     startFight();
}

Simulator::~Simulator()
{
     delete _aimingFitness;
     delete _steeringFitness;

     delete _gunnerPopulation;
     delete _driverPopulation;

     delete [] _tanks;

     for (int i = 0; i < NUM_WALLS; i++)
          delete _walls[i];

     delete _walls;
}

Obstacle *Simulator::getObstacle(int index) const
{
     if (index < NUM_WALLS)
          return _walls[index];

     return _tanks[index - NUM_WALLS];
}

int Simulator::getNumObstacles() const
{
     return NUM_WALLS + GROUP_SIZE;
}

int Simulator::getNumTanks() const
{
     return GROUP_SIZE;
}

Tank *Simulator::getTank(int index) const
{
     return _tanks[index];
}

void Simulator::startGeneration()
{
     _groupCount = 0;
}

void Simulator::startGroup()
{
     _fightCount = 0;

     for (int i = 0; i < GROUP_SIZE; i++)
     {
          _steeringFitness[i] = 0.0;
          _aimingFitness[i] = 0.0;
     }
}

void Simulator::startFight()
{
     _updateCount = 0;

     for (int i = 0; i < GROUP_SIZE; i++)
     {
          // place tank 0 on the left and tank 1 on the right of the arena
          double x = i == 0 ? 10 : 130;
          double y = Random::getUniform() * 60.0 + 10.0;

          // give initial random orientation
          double alpha = Random::getUniform() * 2 * PI;

          // get genotype from population
          const double *driverCode =
               _driverPopulation->getGenes(_groupCount * GROUP_SIZE + i);
          const double *gunnerCode =
               _gunnerPopulation->getGenes(_groupCount * GROUP_SIZE + i);

          // create phenotype using genotype
          _tanks[i] = new AutoTank(Vector2(x, y),
               alpha, driverCode, gunnerCode);
     }

     cout << "generation: [ " << _driverPopulation->getGeneration()
          << ' ' << _gunnerPopulation->getGeneration() << " ]"
          << ", clique: " << _groupCount << ", fight: "
          << _fightCount << endl;
}

void Simulator::endFight()
{
     for (int i = 0; i < GROUP_SIZE; i++)
     {
          double steeringFitness = _tanks[i]->getSteeringFitness();
          double aimingFitness = _tanks[i]->getAimingFitness();

          cout << "tanks[" << i << "]: fight fitness: driver: "
               << steeringFitness << ", gunner: " << aimingFitness << endl;

          // add to total fitness for averaging
          _steeringFitness[i] += steeringFitness;
          _aimingFitness[i] += aimingFitness;

          // detroy phenotypes
          delete _tanks[i];
     }

     _fightCount++;
}
```

```
void Simulator::endGroup()
{
     for (int i = 0; i < GROUP_SIZE; i++)
     {
          // compute average fitness
          _steeringFitness[i] /= NUM_FIGHTS;
          _aimingFitness[i] /= NUM_FIGHTS;

          cout << "tanks[" << i << "]: mean fitness: driver: "
               << _steeringFitness[i] << ", gunner: "
               << _aimingFitness[i] << endl;

          // store fitness in population for rank selection later on
          _driverPopulation->setFitness(
               _groupCount * GROUP_SIZE + i, _steeringFitness[i]);
          _gunnerPopulation->setFitness(
               _groupCount * GROUP_SIZE + i, _aimingFitness[i]);
     }

     _groupCount++;
}

void Simulator::endGeneration()
{
     _driverPopulation->sort();
     _driverPopulation->save(DRIVERS_POPULATION_FILE);
     _driverPopulation->reproduce();

     _gunnerPopulation->sort();
     _gunnerPopulation->save(GUNNERS_POPULATION_FILE);
     _gunnerPopulation->reproduce();
}

// iteration controller
void Simulator::update()
{
     for (int i = 0; i < GROUP_SIZE; i++)
          _tanks[i]->update();

     _updateCount++;

     if (_updateCount >= NUM_UPDATES)
     {
          endFight();
          if (_fightCount >= NUM_FIGHTS)
          {
               endGroup();
               if (_groupCount >= POPULATION_SIZE / GROUP_SIZE)
               {
                    endGeneration();
                    startGeneration();
               }

               startGroup();
          }

          startFight();
     }
}

void Simulator::draw(const Graphics *graphics) const
{
     graphics->preRender();

     for (int i = 0; i < getNumObstacles(); i++)
          getObstacle(i)->draw(graphics);

     graphics->flush();
}

void Simulator::drawNetwork(const Graphics *graphics) const
{
     graphics->preRender();

     _tanks[0]->drawNetwork(graphics);

     graphics->flush();
}

void Simulator::numberPressed(int num)
{
     if (num < GROUP_SIZE)
          _tanks[num]->toggleTrackVisible();
}
```

9.3.26 Tank.h

```
#ifndef Tank_H
#define Tank_H

// Description: Base class for tanks, from this class
//              autonomous or user-contolled tanks can be derived
// Author:      Yvan Bourquin

class Graphics;
class ProximitySensor;
class VisionSensor;
class Track;
class Shell;

#include "Obstacle.h"
#include "Vector2.h"
#include "Quad2.h"

class Tank : public Obstacle
{
public:
     // destructor
     virtual ~Tank();

     // update sensors, move to next position unless a collision occurs
     virtual void update();

     // draw at current angle and position
     virtual void draw(const Graphics *graphics) const;

     // get current position
     // position of the centre of the body
     const Vector2 &getPosition() const { return _position; }

     // tank/turret rotation angle.
     // zero when facing east, then clockwise in radians.
     // values are not bounded and might exceed range [-pi;pi]
     double getAlpha() const { return _alpha; } // tank
     double getBeta() const { return _beta; }   // turret

     // return fitness
     double getSteeringFitness() const { return _steeringFitness; }
     double getAimingFitness() const { return _aimingFitness; }

     // make tank trace visible/invisible
     void toggleTrackVisible() { _trackVisible = ! _trackVisible; }

     // for derived class
     static const int NUM_PROXIMITY_SENSORS;
     static const int NUM_BODY_MOTORS;
     static const int NUM_VISION_SENSORS;
     static const int NUM_TURRET_MOTORS;

protected:
     // constructor available for derived classes only
     Tank(const Vector2 &initialPosition, double alpha);

     // set motor speed and turret speed
     void setLeftSpeed(double speed);
     void setRightSpeed(double speed);
     void setTurretRotationSpeed(double speed);

     // get sensor output
     double getSensorOutput(int index) const;
     double getTurretSensorOutput(int index) const;

private:
     double _alpha;      // body rotation angle
     double _beta;       // turret rotation angle
     double _leftSpeed;  // left motor speed
     double _rightSpeed; // right motor speed
     double _turretRotationSpeed;
     Vector2 _position;
     ProximitySensor **_proximitySensors;
     VisionSensor **_visionSensors;
     double _steeringFitness;
     double _aimingFitness;
     Track *_track;
     bool _trackVisible;
     Quad2 _turretBounds;
     Quad2 _gunBounds;
     Shell *_shell;

     // local methods
     void bodyTransform();
     void turretTransform();
     bool willCollide(const Vector2 &destination, double angle) const;
};

#endif
```

9.3.27 Tank.cpp

```
#include "Tank.h"
#include "Graphics.h"
#include "ProximitySensor.h"
#include "VisionSensor.h"
#include "Simulator.h"
#include "Track.h"
#include "Shell.h"

const double WIDTH = 4;  // tank body width
const double LENGTH = 6; // tank body length

const int Tank::NUM_PROXIMITY_SENSORS = 6;
const int Tank::NUM_BODY_MOTORS       = 2;
const int Tank::NUM_VISION_SENSORS    = 6;
const int Tank::NUM_TURRET_MOTORS     = 2;

static const Quad2 BOUNDS(LENGTH / 2, WIDTH / 2, LENGTH / 2,
                              -WIDTH / 2, -LENGTH / 2, -WIDTH / 2,
                              -LENGTH / 2, WIDTH / 2);
static const Quad2 TURRET_BOUNDS(0, 1, 0, -1, -2, -1, -2, 1);
static const Quad2 GUN_BOUNDS(2, 0.5, 2, -0.5, 0, -0.5, 0, 0.5);

static const double TURRET_SENSOR_WEIGHTS[Tank::NUM_VISION_SENSORS]
     = { 0.1, 0.3, 1.0, 1.0, 0.3, 0.1 };
static const double MAX_SPEED = 0.4;
static const double MAX_BACK_SPEED = 0.2;
static const double MAX_TURRET_ROTATION_SPEED = 0.1;
static const double PI = 3.141592;

Tank::Tank(const Vector2 &position, double alpha)
: _alpha(alpha), _beta(alpha), _leftSpeed(0.0), _rightSpeed(0.0),
     _turretRotationSpeed(0.0),
_position(position), _steeringFitness(0.0), _aimingFitness(0.0)
{
     _track = new Track(position);
     _trackVisible = false;
     _shell = NULL;

     bodyTransform();
     turretTransform();

     _proximitySensors = new ProximitySensor*[NUM_PROXIMITY_SENSORS];

     _proximitySensors[0] = new ProximitySensor(this, -3,  2, -2,  1);
     _proximitySensors[1] = new ProximitySensor(this,  2,  2,  1,  1);
     _proximitySensors[2] = new ProximitySensor(this,  3,  2,  1,  0);
     _proximitySensors[3] = new ProximitySensor(this,  3, -2,  1,  0);
     _proximitySensors[4] = new ProximitySensor(this,  2, -2,  1, -1);
     _proximitySensors[5] = new ProximitySensor(this, -3, -2, -2, -1);
     _visionSensors = new VisionSensor*[NUM_VISION_SENSORS];

     _visionSensors[0] = new VisionSensor(this, 0,  1.0,  4,  1);
     _visionSensors[1] = new VisionSensor(this, 0,  0.6,  8,  1);
     _visionSensors[2] = new VisionSensor(this, 0,  0.2, 24,  1);
     _visionSensors[3] = new VisionSensor(this, 0, -0.2, 24, -1);
     _visionSensors[4] = new VisionSensor(this, 0, -0.6,  8, -1);
     _visionSensors[5] = new VisionSensor(this, 0, -1.0,  4, -1);
}

Tank::~Tank()
{
     if (_shell)
          delete _shell;

     for (int j = 0; j < NUM_VISION_SENSORS; j++)
          delete _visionSensors[j];

     delete [] _visionSensors;

     for (int i = 0; i < NUM_PROXIMITY_SENSORS; i++)
          delete _proximitySensors[i];

     delete [] _proximitySensors;

     delete _track;
}

void Tank::update()
{
     double averageSpeed = (_leftSpeed + _rightSpeed) / 2.0;
     double alpha = _alpha + (_leftSpeed - _rightSpeed) / 10.0;

     Vector2 direction;
     direction.x() = cos(- alpha);
     direction.y() = sin(- alpha);

     Vector2 destination = _position + direction * averageSpeed;
     if (willCollide(destination, alpha))
     {
          setLeftSpeed(0.0);
          setRightSpeed(0.0);
          averageSpeed = 0.0;
     }
     else
     {
          _alpha = alpha;
          _position = destination;
          _track->addStep(_position);
          bodyTransform();
     }

     _beta += _turretRotationSpeed;

     turretTransform();
     double maxActivity = -999.9;
     for (int i = 0; i < NUM_PROXIMITY_SENSORS; i++)
     {
          _proximitySensors[i]->update();
          if (_proximitySensors[i]->getOutput() > maxActivity)
               maxActivity = _proximitySensors[i]->getOutput();
     }

     double deltaAimingFitness = 0.0;
     for (int j = 0; j < NUM_VISION_SENSORS; j++)
     {
          _visionSensors[j]->update();
          double output = _visionSensors[j]->getOutput() > 0.0 ? 1.0 : 0.0;
          deltaAimingFitness += output * TURRET_SENSOR_WEIGHTS[j];
     }

     if (_shell)
     {
          if (_shell->isExploded())
          {
               delete _shell;
               _shell = NULL;
          }
          else
               _shell->update();
     }
     else if (deltaAimingFitness >= 1.0)
          _shell = new Shell(this, _position, _beta);

     // steering fitness function
     double deltaSpeed = fabs(_leftSpeed - _rightSpeed) /
          (MAX_SPEED + MAX_BACK_SPEED);
     double deltaSteeringFitness = averageSpeed / MAX_SPEED
          * (1.0 - sqrt(deltaSpeed)) * (1.0 - maxActivity);

     _steeringFitness += deltaSteeringFitness;
     _aimingFitness += deltaAimingFitness;
}

void Tank::setLeftSpeed(double speed)
{
     _leftSpeed += (speed - _leftSpeed) / 2.0;

     if (_leftSpeed > MAX_SPEED)
          _leftSpeed = MAX_SPEED;
     else if (_leftSpeed < -MAX_BACK_SPEED)
          _leftSpeed = -MAX_BACK_SPEED;
}

void Tank::setRightSpeed(double speed)
{
     _rightSpeed += (speed - _rightSpeed) / 2.0;

     if (_rightSpeed > MAX_SPEED)
          _rightSpeed = MAX_SPEED;
     else if (_rightSpeed < -MAX_BACK_SPEED)
          _rightSpeed = -MAX_BACK_SPEED;
```

```
}

void Tank::setTurretRotationSpeed(double speed)
{
     if (fabs(speed) < MAX_TURRET_ROTATION_SPEED)
          _turretRotationSpeed = speed;
}

bool Tank::willCollide(const Vector2 &destination, double angle) const
{
     if (destination == _position && angle == _alpha)
          return false;

     Quad2 bounds = BOUNDS;
     bounds.rotate(-angle);
     bounds.translate(destination);
     Simulator *simulator = Simulator::instance();

     // for every object in the simulator ...
     for (int j = 0; j < simulator->getNumObstacles(); j++)
     {
          const Obstacle *obstacle = simulator->getObstacle(j);

          // don't detect collision with self
          if (obstacle != this)
          {
               if (bounds.intersects(obstacle->getBounds()))
                    return true;
          }
     }

     // there was no collision detected
     return false;
}

// compute new body coordinates
void Tank::bodyTransform()
{
     _bounds = BOUNDS;
     _bounds.rotate(-_alpha);
     _bounds.translate(_position);
}

// compute new turret coordinates
void Tank::turretTransform()
{
     _turretBounds = TURRET_BOUNDS;
     _turretBounds.rotate(-_beta);
     _turretBounds.translate(_position);

     _gunBounds = GUN_BOUNDS;
     _gunBounds.rotate(-_beta);
     _gunBounds.translate(_position);
}

void Tank::draw(const Graphics *graphics) const
{
     if (_trackVisible)
          _track->draw(graphics);

     // draw tank body
     graphics->setColor(0.6f, 0.6f, 0.6f);
     graphics->drawSolidQuad(_bounds);

     // draw tank turret
     graphics->setColor(0.2f, 0.2f, 0.2f);
     graphics->drawSolidQuad(_turretBounds);
     graphics->drawSolidQuad(_gunBounds);

     // draw body sensors
     for (int i = 0; i < NUM_PROXIMITY_SENSORS; i++)
          _proximitySensors[i]->draw(graphics);

     // draw turret sensors
     for (int j = 0; j < NUM_VISION_SENSORS; j++)
          _visionSensors[j]->draw(graphics);

     if (_shell)
          _shell->draw(graphics);
}

double Tank::getSensorOutput(int index) const
{
     return _proximitySensors[index]->getOutput();
}

double Tank::getTurretSensorOutput(int index) const
{
     return _visionSensors[index]->getOutput();
}
```

9.3.28 Track.h

```
#ifndef Track_H
#define Track_H

// Description: Tank track for drawing in simulator
// Author:      Yvan Bourquin

#include "Vector2.h"

class Graphics;

class Track
{
public:
     // constructor: create track with initial step
     Track(const Vector2 &initialStep);

     // destructor
     virtual ~Track();

     // add tank current position
     void addStep(const Vector2 &step);
     // draw track using graphic context
     void draw(const Graphics *graphics) const;

private:
     Vector2 *_steps;
     int _numSteps;
};

#endif
```

9.3.29 Track.cpp

```
#include "Track.h"
#include "Simulator.h"
#include "Graphics.h"
#include <assert.h>

Track::Track(const Vector2 &initialStep)
{
     _steps = new Vector2[Simulator::NUM_UPDATES + 1];
     _numSteps = 0;
     addStep(initialStep);
}

Track::~Track()
{
     delete [] _steps;
}

void Track::addStep(const Vector2 &step)
{
     assert(_numSteps < Simulator::NUM_UPDATES + 1);

     // if coordinates haven't changed, there is no need to store them
     if (step == _steps[_numSteps - 1])
          return;

     _steps[_numSteps] = step;
     _numSteps++;
}

void Track::draw(const Graphics *graphics) const
{
     graphics->setColor(0, 0, 1);
     graphics->setLineWidth(2.0);

     for (int i = 1; i < _numSteps; i++)
          graphics->drawLine(_steps[i - 1], _steps[i]);
}
```

9.3.30 Vector.h

```
#ifndef Vector_H
#define Vector_H

// Description: General purpose N-dimensions vector
// Author:      Yvan Bourquin

#include <assert.h>

class ostream;

class Vector
{
public:
     // constructors
     Vector();
     Vector(int size);
     Vector(const Vector &other);

     // destructor
     ~Vector();

     // read-only element access
     double operator [] (int index) const;

     // read/write element access
     double &operator [] (int index);

     // assignment operator
     Vector &operator = (const Vector &);

     // unary + and - operators
     Vector operator + () const { return *this; }
    Vector operator - () const { return 0.0 - *this; }

     // vector-vector operations
     Vector operator + (const Vector &) const;
     Vector operator - (const Vector &) const;
     Vector operator * (const Vector &) const;
     Vector operator / (const Vector &) const;

     // dot product
     double dot(const Vector &) const;

     // vector-scalar operations
     Vector operator + (double) const;
     Vector operator - (double) const;
     Vector operator * (double) const;
     Vector operator / (double) const;

     // print vector to output stream
     friend ostream &operator << (ostream &, const Vector &);

     // scalar-vector operations
     friend Vector operator + (double, const Vector &);
     friend Vector operator - (double, const Vector &);
     friend Vector operator * (double, const Vector &);
     friend Vector operator / (double, const Vector &);

     // vector size
     int size() const { return N; }
     // set all elements to zero
     void zero();

     // set default vector size when calling default constructor
     static void setDefaultSize(int size) { defaultSize = size; }

private:
     double *a;   // elements
     const int N; // size

     static int defaultSize;
};

inline void Vector::zero()
{
     for (int i = 0; i < N; i++)
          a[i] = 0.0;
}

inline double Vector::operator [] (int index) const
{
     assert(index >= 0 && index < N);
     return a[index];
}

inline double &Vector::operator [] (int index)
{
     assert(index >= 0 && index < N);
     return a[index];
}

inline Vector::Vector()
     : N(defaultSize)
{
     a = new double[N];
}

inline Vector::Vector(int size)
     : N(size)
{
     a = new double[N];
}

inline Vector::Vector(const Vector &other)
     : N(other.N)
{
     a = new double[N];
     for (int n = 0; n < N; n++)
          a[n] = other.a[n];
}

inline Vector::~Vector()
{
     delete [] a;
}

inline Vector &Vector::operator = (const Vector &other)
```

```
{
     assert(N == other.N);
     for (int n = 0; n < N; n++)
          a[n] = other.a[n];

     return *this;
}

inline Vector Vector::operator + (const Vector &other) const
{
     assert(N == other.N);

     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] + other.a[n];

     return y;
}

inline Vector Vector::operator - (const Vector &other) const
{
     assert(N == other.N);

     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] - other.a[n];

     return y;
}

inline Vector Vector::operator * (const Vector &other) const
{
     assert(N == other.N);

     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] * other.a[n];

     return y;
}

inline Vector Vector::operator / (const Vector &other) const
{
     assert(N == other.N);

     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] / other.a[n];

     return y;
}

inline double Vector::dot(const Vector &other) const
```

```
{
     assert(N == other.N);

     double d = 0.0;
     for (int n = 0; n < N; n++)
          d += a[n] * other.a[n];

     return d;
}

inline Vector Vector::operator + (double d) const
{
     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] + d;

     return y;
}

inline Vector Vector::operator - (double d) const
{
     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] - d;

     return y;
}

inline Vector Vector::operator * (double d) const
{
     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] * d;

     return y;
}

inline Vector Vector::operator / (double d) const
{
     Vector y(N);

     for (int n = 0; n < N; n++)
          y.a[n] = a[n] / d;

     return y;
}

inline Vector operator + (double d, const Vector &other)
{
     Vector y(other.N);

     for (int n = 0; n < other.N; n++)
          y.a[n] = d + other.a[n];
     return y;
}

inline Vector operator - (double d, const Vector &other)
{
     Vector y(other.N);

     for (int n = 0; n < other.N; n++)
          y.a[n] = d - other.a[n];

     return y;
}

inline Vector operator * (double d, const Vector &other)
{
     Vector y(other.N);

     for (int n = 0; n < other.N; n++)
          y.a[n] = d * other.a[n];

     return y;
}

inline Vector operator / (double d, const Vector &other)
{
     Vector y(other.N);

     for (int n = 0; n < other.N; n++)
          y.a[n] = d / other.a[n];

     return y;
}

#endif
```

9.3.31 Vector.cpp

```
#include "Vector.h"
#include <iostream.h>

int Vector::defaultSize = 1;

ostream &operator << (ostream &os, const Vector &v)
{
     os << "[ ";
     for (int n = 0; n < v.N; n++)
          os << v.a[n] << " ";
     os << "]";

    return os;
}
```

9.3.32 Vector2.h

```
#ifndef Vector2_H
#define Vector2_H
```

```
// Description: Two-dimensional vector and operations
// Author:      Yvan Bourquin

#include <math.h>

class ostream;
class istream;

class Vector2 {
public:
    // default constructor, constructs a null vector
    Vector2(): _x(0.0), _y(0.0) {}

    // constructors given vector components
    Vector2(double x, double y) : _x(x), _y(y) {}
    Vector2(const Vector2 &v) : _x(v._x), _y(v._y) {}

    // read-only element access
    double x() const { return _x; }
    double y() const { return _y; }

    // read-write element access
    double &x() { return _x; }
    double &y() { return _y; }

    // component-wise equality comparison operators.
    int operator == (const Vector2 &) const;
    int operator != (const Vector2 &) const;

    // length-wise comparison operators.
    // example: if (v > w) ...
    int operator > (const Vector2 &) const;
    int operator < (const Vector2 &) const;
    int operator >= (const Vector2 &) const;
    int operator <= (const Vector2 &) const;

    // component-wise binary vector addition and subtraction operators.
    // examples: v + w, v - w
    Vector2 operator + (const Vector2 &) const;
    Vector2 operator - (const Vector2 &) const;

    // component-wise binary scalar multiplication and division operators.
    // examples: v * 10.0, 5.0 * v, v / 12.0
    Vector2 operator * (double) const;
    friend Vector2 operator * (double, const Vector2 &);
    Vector2 operator / (double) const;

    // non-destructive unary + (plus) and - (minus) operators
    // examples: -v, +v
    Vector2 operator + () const { return *this; }
    Vector2 operator - () const { return Vector2(-_x, -_y); }

    // component-wise vector addition and subtraction operators
    // combined with assignement.
    // examples: w = v, v += w, v -= w
    Vector2 &operator = (const Vector2 &);
    Vector2 &operator += (const Vector2 &);
    Vector2 &operator -= (const Vector2 &);

    // component-wise scalar multiplication and division operators
    // combined with assignement.
    // examples: v *= 2.0, v /= 45.0
    Vector2 &operator *= (double d);
    Vector2 &operator /= (double d);

   // Returns geometric length of vector.
    double length() const;

     // Returns geometric distance between this vector and point.
     double distance(const Vector2& point) const;

    // Changes vector to be unit length (default) or specified length
    void normalize(double length = 1.0);

    // Creates a new normalized vector (unit length) based on this one
    Vector2 normalized() const;

     // rotate vector around origin
     void rotate(double alpha);

     // return rotated around origin version of vector
     Vector2 rotated(double alpha) const;

    // print to output stream: format: [x y]
    friend ostream &operator << (ostream &, const Vector2 &);

private:
     double _x;
     double _y;
};

inline Vector2 Vector2::operator + (const Vector2 &b) const
{
    return Vector2(_x + b._x, _y + b._y);
}

inline Vector2 Vector2::operator - (const Vector2 &b) const
{
    return Vector2(_x - b._x, _y - b._y);
}

inline Vector2 Vector2::operator * (double d) const
{
    return Vector2(_x * d, _y * d);
}

inline Vector2 Vector2::operator / (double d) const
{
    return Vector2(_x / d, _y / d);
}

inline Vector2 operator * (double d, const Vector2 &v)
{
    return Vector2(d * v._x, d * v._y);
```

```
}

inline Vector2 & Vector2::operator = (const Vector2 &v)
{
    _x = v._x;
     _y = v._y;
    return *this;
}

inline Vector2 & Vector2::operator += (const Vector2 &v)
{
    _x += v._x;
     _y += v._y;
    return *this;
}

inline Vector2 & Vector2::operator -= (const Vector2 &v)
{
    _x -= v._x;
     _y -= v._y;
    return *this;
}

inline Vector2 & Vector2::operator *= (double d)
{
    _x *= d;
     _y *= d;
    return *this;
}

inline Vector2 & Vector2::operator /= (double d)
{
    _x /= d;
     _y /= d;
    return *this;
}

inline int Vector2::operator > (const Vector2 &b) const
{
    return length() > b.length() ? 1 : 0;
}

inline int Vector2::operator < (const Vector2 &b) const
{
    return length() < b.length() ? 1 : 0;
}

inline int Vector2::operator >= (const Vector2 &b) const
{
    return length() >= b.length() ? 1 : 0;
}

inline int Vector2::operator <= (const Vector2 &b) const
{
    return length() <= b.length() ? 1 : 0;
}
```

```
inline double Vector2::length() const
{
    return sqrt(_x * _x + _y * _y);
}

inline double Vector2::distance(const Vector2& point) const
{
    return (*this - point).length();
}

inline int Vector2::operator == (const Vector2 &v) const
{
    return _x == v._x && _y == v._y;
}

inline int Vector2::operator != (const Vector2 &v) const
{
    return _x != v._x || _y != v._y;
}

inline void Vector2::normalize(double newLength)
{
    double d = 1.0 / length() * newLength;
    _x *= d;
     _y *= d;
}

inline Vector2 Vector2::normalized() const
{
    Vector2 r(*this);
    r.normalize();
    return r;
}

#endif
```

9.3.33 Vector2.cpp

```
#include "Vector2.h"
#include <iostream.h>

void Vector2::rotate(double alpha)
{
     Vector2 b;
     b._x = _x * cos(alpha) - _y * sin(alpha);
     b._y = _x * sin(alpha) + _y * cos(alpha);
     *this = b;
}

Vector2 Vector2::rotated(double alpha) const
{
     Vector2 b;
     b._x = _x * cos(alpha) - _y * sin(alpha);
     b._y = _x * sin(alpha) + _y * cos(alpha);
     return b;
}
```

```
ostream &operator << (ostream &os, const Vector2 &v)
{
    os << '[' << v._x << ", " << v._y << ']';
    return os;
}
```

9.3.34 VisionSensor.h

```
#ifndef VisionSensor_H
#define VisionSensor_H

// Description: Sensor sensitive to tanks only
// Author:      Yvan Bourquin

class Tank;
class Graphics;

#include "Line2.h"

class VisionSensor
{
public:
     // constructor:
     // tank: the tank which this sensor belongs to
     // x, y: position of the sensor in tank coordinate system
     // dx, dy: direction of the sensor ray in tank coordinate system
     VisionSensor(const Tank *tank, double x, double y,
          double dx, double dy);

     // draw using graphic context
     void draw(const Graphics *graphics) const;

     // update sensor output
     void update();

     // return a number in the range [0,1] inversely proportional
     // to the distance to the nearest tank in the sensor ray
     // returns 0.0 if there is no tank the sensed range
     // returns 1.0 if there is a tank right in front of the sensor
     double getOutput() const { return _output; }

     // toggle visibility flag of all vision sensors
     static void toggleVisibility() { _visible = ! _visible; }

private:
     const Tank *_tank;
     Vector2 _position;
     Vector2 _direction;
     Line2 _ray;
     double _output;
     static bool _visible;
};

#endif
```

9.3.35 VisionSensor.cpp

```
#include "VisionSensor.h"
#include "Graphics.h"
#include "Tank.h"
#include "Simulator.h"

static const double MAX_RANGE = 140.0;
bool VisionSensor::_visible = true;

VisionSensor::VisionSensor(const Tank *tank, double x, double y,
                                double dx, double dy)
     : _tank(tank), _output(0.0), _position(x, y)
{
     _direction = Vector2(dx, dy).normalized();
}

void VisionSensor::draw(const Graphics *graphics) const
{
     if (! _visible)
          return;

     graphics->setLineWidth(1.0);
     if (_output)
          graphics->setColor(1, 0, 1);
     else
          graphics->setColor(0, 0, 1);

     graphics->drawLine(_ray);
}

void VisionSensor::update()
{
     // compute new ray position and direction
     _ray = Line2(_position, _position + _direction * MAX_RANGE);
     _ray.rotate(-_tank->getBeta());
     _ray.translate(_tank->getPosition());

     const Obstacle *_nearestObstacle = NULL;

     Simulator *simulator = Simulator::instance();
     int N = simulator->getNumObstacles();

     // for every object in the simulator
     for (int i = 0; i < N; i++)
     {
          const Obstacle *obstacle = simulator->getObstacle(i);

          // don't check intersection with own tank
          if (obstacle != _tank)
          {
               Quad2 bounds = obstacle->getBounds();
               for (int j = 0; j < 4; j++)
               {
                    // check intersection
                    Vector2 intersect = _ray.intersects(bounds.getLine(j));
                    if (intersect!= Line2::NO_INTERSECT)
                    {
                         double distance = _ray.a().distance(intersect);
                         if (distance < _ray.length())
                         {
                              _ray.b() = intersect;

                              // remember nearest object
                              _nearestObstacle = obstacle;
                         }
                    }
               }
          }
     }

     if (_nearestObstacle)
          _output = dynamic_cast<const Tank*>(_nearestObstacle) ?
               1.0 - _ray.length() / MAX_RANGE : 0.0;
     else
          _output = 0.0;
}
```

9.3.36 Wall.h

```
#ifndef Wall_H
#define Wall_H

// Description: Wall for the tank simulator
// Author:      Yvan Bourquin

#include "Obstacle.h"

class Graphics;

class Wall : public Obstacle
{
public:
     // construct a closed wall from [ax,ay] to [bx,by]
     // to [cx,cy] to [dx,dy] and back to [ax,ay]
     Wall(double ax, double ay, double bx, double by,
          double cx, double cy, double dx, double dy);

     // construct a wall of width "width" from [x1,y1] to [x2,y2]
     Wall(double x1, double y1, double x2, double y2, double width);

     // draw using graphic context
     void draw(const Graphics *graphics) const;

private:
};

#endif
```

9.3.37 Wall.cpp

```
#include "Wall.h"
#include "Graphics.h"
```

```
#include "Wall.h"

static const double PI = 3.14159265358979;

Wall::Wall(double ax, double ay, double bx, double by,
            double cx, double cy, double dx, double dy)
{
     _bounds = Quad2(ax, ay, bx, by, cx, cy, dx, dy);
}

Wall::Wall(double x1, double y1, double x2, double y2, double width)
{
     Vector2 a(x1, y1);
     Vector2 b(x2, y2);
     Vector2 r = b - a;
     r.rotate(PI / 2.0);
     r.normalize(width / 2.0);

     _bounds = Quad2(a + r, a - r, b - r, b + r);
}

void Wall::draw(const Graphics *graphics) const
{
     graphics->setColor(0, 0, 0);
     graphics->setLineWidth(3.0);
     graphics->drawLineQuad(_bounds);
}
```