Steering Behaviors

Overview

Project history

The first version of the “Steering Behaviors” project was created as part of the course on object oriented programming in the winter semester 2000/ 2001 held by Prof. Jürgen Sauer. The goal of the course was to program a Java applet. The “Steering Behaviors” project was based on the Siggraph 2001 paper written by Robin Green. The first version showcased a system with simple behavior combinations. A self-defined scene description language was defined to be able to create user-defined scenarios. The script was passed to the applet as a parameter, parsed and the corresponding scene was created.

Early 2001 the idea of enhancing the project to a thesis work was formed. All of the behaviors described in Craig Reynolds paper would now be implemented. The intelligence of the simulation would be enhanced by a special behavior control system. This would allow adjusting the intelligence according to the setup of the scene.

Instead of using a proprietary description language for the scenes, the meta-language XML was chosen as a replacement. Scenes will now be defined using a standardized and platform independent language. To ease the creation of user-defined scenes, an editor would be programmed. It will allow the reading and storing of XML files and implementation of every scene possible.

Another helpful idea was the implementation of our own rendering system. It can be used in the simulation applet as well as the editor and sped up development time significantly.

Because of its completeness and the possibility to create user-defined scenes, the project “Steering Behaviors” represents a very good reference for people interested in the works of Craig Reynolds.

Steering Behaviors

Steering Behaviors are the next logical step in the further development of the 'boids' model created by Craig Reynolds in 1986. The 'boids' computer based model is a way to simulate the coordinated movements seen in flocks of birds or fish. The name 'boids' identifies here one instance of the simulated creatures. The original flocking model was based on three distinct behaviors. The 'Separation' behavior was used to prevent situations where members of the swarm were crowding each other. 'Aligment' was used to steer all elements of the flock in a common direction. The third behavior, 'Cohesion', steered a single 'boid' to the average position of nearby flockmates. Reynolds was able to create a realistic simulation of the natural behaviors of swarm through a skillful combination of these three simple behaviors. A first example for the usage of this model is the short film "Stanley and Stella in: Breaking the Ice". It was created by the Symbolics Graphics Division in a cooperation with Whitney / Demos Production in the year 1987. The first showing was in the electronic theater at Siggraph 1987.

Figure 1: Scene from 'Stanley and Stella in: Breaking the Ice' (1987)

Craig Reynolds presented the evaluated and implemented algorithms for obstacle avoidance of each individual 'boid' that were used in this production in his paper titled 'Not bumping into things' in the year 1988. He described several possibilities for algorithms and rules for the avoidance of obstacles. The strategies are not based on complex planning strategies and pathfinding algorithms, but instead only use local information, e.g. objects in the near radius around the vehicle.

Reynolds work on steering behaviors, presented at the Games Developer Conference in 1999, enhances the behaviors already presented in the original 'boids' model. New building parts for complex autonomous systems are presented. Each of these new behaviors defines only a specific reaction on the simulated environment of the autonomous system, simply called vehicle in the later course. The results are simple building parts for complex systems. Depending on the combination of behaviors the vehicle can be configured to handle different complex situation. The behaviors grouped under the name 'Steering Behaviors' are only the lowest level for an autonomous system. Several examples for how these can be combined are presented in the paper.
At Siggraph 2000, Robin Green presented a paper titled 'Steering Behaviours'. In this work he builds on the behaviors described in Reynolds paper and brings up examples on how these can be implemented using the C++ language. In addition several possible problems and their corresponding solutions are discussed. This work was created as part of the development of the 'Dungeon Master 2' computer game of Bullfrog Productions Ltd.

Figure 2: Screenshot of Dungeon Master 2, Bullfrog Productions Ltd.

This game can be seen as a successful utilization of steering behaviors. The autonomous characters in the game are controlled using simple behaviors and show what kind of realism can be achieved.

Theorie

Vehicle Model

Craig Reynolds as well as Robin Green is using a simple vehicle model in their works. It is abased on a point mass defining the vehicle. A position and a velocity define a vehicle. For further simplification the mass will be specified as one and therefore does not show up in any further calculations. The orientation of the vehicle is described by the two vectors `m_localx` and `m_locally`. These two vectors represent the local coordinate system and are used to transform between local and world coordinates. To update the position of the vehicle the generated force vector is added to the current velocity and the result added to the position. This calculation scheme called 'euler integration' is very simple to implement and in this case totally sufficient. It can always be replaced by a more stable integration scheme if there is the need to, without having to change anything on the simulation itself. The local coordinate system is adjusted to the velocity vector for each frame. To prevent possible jittering of the vehicle in case of behaviors steering to different directions, the velocity vector is sampled over the last three frames and the result used for updating the local coordinate system.

Behaviors

Seek and Flee behaviors

Application

The seek and flee behaviors implement complementally behavioral patterns. The logic behind it is the same for both of them. The only difference is the resulting steering force. Seek is used to steer a vehicle to the specified target. An example would be a fish swimming directly to its food. The target can be a fixed point in space or another vehicle and moving. The flee behavior is used to simulate a simple form of evasion. As is the case with seek, the target can be stationary or moving.

Implementation

The steering force is calculated based on the values for the current velocity of the vehicle and the position of the specified target. First, the position of the target is expressed as a vector between the target and the vehicle. This represents the desired velocity. The steering force results from the difference between this vector and the current velocity vector.

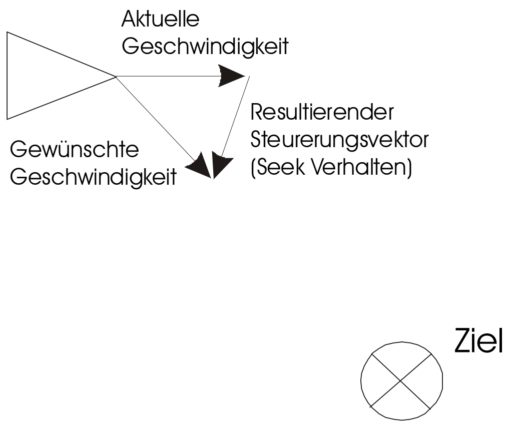
Aktuelle Geschwindigkeit
Resultierender Steurerungsvektor (Seek Verhalten)
Gewünschte Geschwindigkeit
Ziel

Figure 3: Force vector for the seek behavior

In the seek behavior the current velocity of the vehicle is subtracted from the desired velocity. To get the steering force for the flee behavior, the desired velocity is subtracted from the current velocity instead. Using these simple calculations the vehicle can either be steered to its target or forced to avoid it. As with all behaviors, the calculated force should not be allowed to exceed the maximum force of the vehicle.

Problems

Since the seek behavior generates a steering force continually, in most cases an undesired effect will happen. If the target is static or moves at a speed lower than the seeking vehicle, the steering force will guide the vehicle to the target at first, but keeps on traveling in the same direction until the target passed. Now a force in the opposite direction is calculated and the vehicle moves back the way it came from. The resulting movement is like a moth dancing around a light source.

Arrive Behavior

Application

The arrive behavior is an extension of the seek behavior. Like the seek behavior it is used for steering the vehicle to a specified target. The important difference is the way it arrives at the destination. The seek behavior meets the destination at full speed, travels some more in the current direction and then goes back again dancing like a moth around a light source. This is normally not the kind of result one was hoping for. The arrive behavior slows the vehicle down in a controlled way according to the specification made by the user and stops its movement at the desired position.

Implementation

The calculation of the steering force starts out the same way as in the seek behaviors. The resulting vector is the difference between the desired velocity and the current velocity.

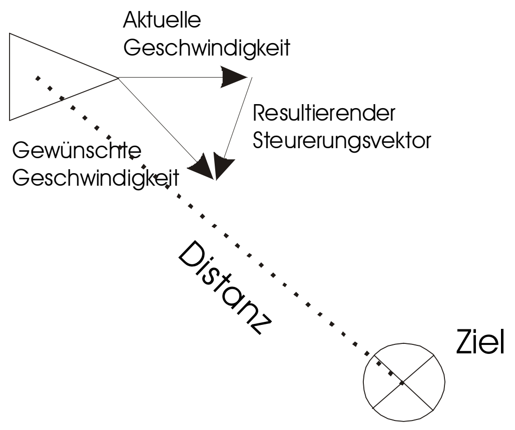

Figure 4: Force vector for the arrive behavior

Up to now there is no difference between the two behaviors. As long as the target is outside the activation distance specified in the attribute named `m_activeDistance`, no steering force will be generated. Inside this area, the force is modified by the distance to the target divided by the desired number of steps. This makes the vehicle slow down according to its current distance to the destination. The result is a vehicle that stops at the specified target.

Problems

There is a possible problem with this algorithm. By using the distance divided by the number of steps to reach the target, the vehicle nears the destination only asymptotically. Normally it will never exactly reach the destination position. From a users point of view there is no visible difference if the exact position is reached, or if the vehicle is moving at an ever-decreasing velocity to the destination. At a certain point in the calculations, the vehicle may suddenly make changes in orientation even if it looked like standing still for a while. A possible solution would be ceasing the updating of the orientation vector if the steering force is below a certain threshold. A second possible solution, used in this implementation, uses the last three orientation vectors to average the orientation over the last few frames of animation. This suppresses fast changes and the jittering as seen in certain behavior combinations is made even automatically.

Wander

Application

The wander behavior empowers vehicles to move autonomously and without a specified target through the scene. It is comparable to ants looking for food. Most animals on their way looking for something do not forage the surrounding area systematically. Instead they move on more random and undefined paths.
The movements in straight lines as seen in other behaviors, like for example Seek or Arrive, can be slightly distorted by using the wander behaviors in conjunction. External influences like wind or terrain features as seen in reality can be simulated this way.

Implementation

Basically the wander behavior is implemented using random steering forces. To create those there are several possibilities. The easiest way would be to add a random vector to the velocity for each frame of animation. The resulting movement will not look very realistic, since the steering force will change direction almost instantaneously. Therefore an algorithm has to be used that prevents this.

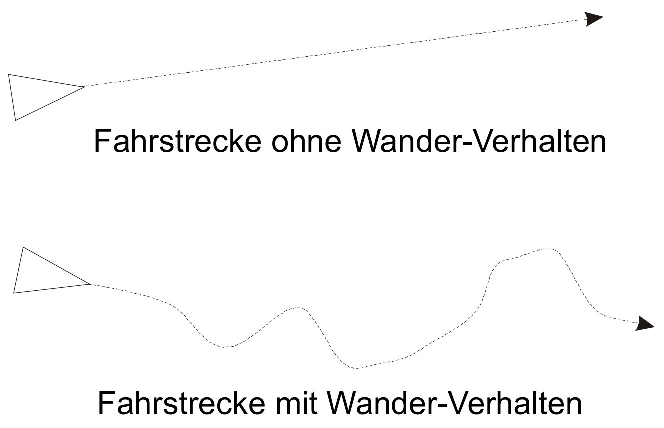
Fahrstrecke ohne Wander-Verhalten
Fahrstrecke mit Wander-Verhalten

Figure 5: Example of movement with and without the wander behavior

Like all other behaviors, wander has to calculate a resulting force vector. In this case the vector only has a certain degree of freedom by constraining the tip of the vector to a circle placed in front of the vehicle. This vector is transformed into world coordinates and used as the steering force for this behavior.

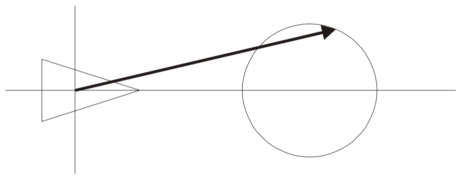

Figure 6: Steering vector constrained to a circle

For each iteration of the simulation a random vector is added to the current steering force. The length of this vector is constrained to a value below `m_rate`. This attribute therefore controls the maximum difference in the steering force. The vector resulting from adding the random vector to the previous steering vector is again constrained to the defined circle.

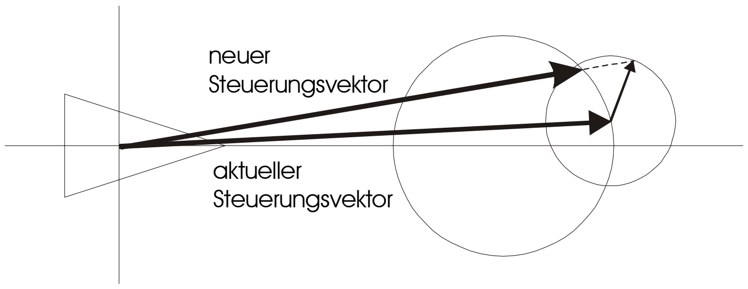

Figure 7: Calculation of the new steering force

The behavior can be adjusted to the needs of the simulation by using different values for the radius of the circle, defined in `m_radius`, the position of the circle, defined in `m_pos`, and the maximum change of direction specified in `m_rate`. If the circle is placed near the vehicle, the changes in the direction of movement will be faster. If it is placed farther away, the resulting movement will be more linear. Placing the circle over or below the local x-axis can specify a slight tendency for left or right movements.

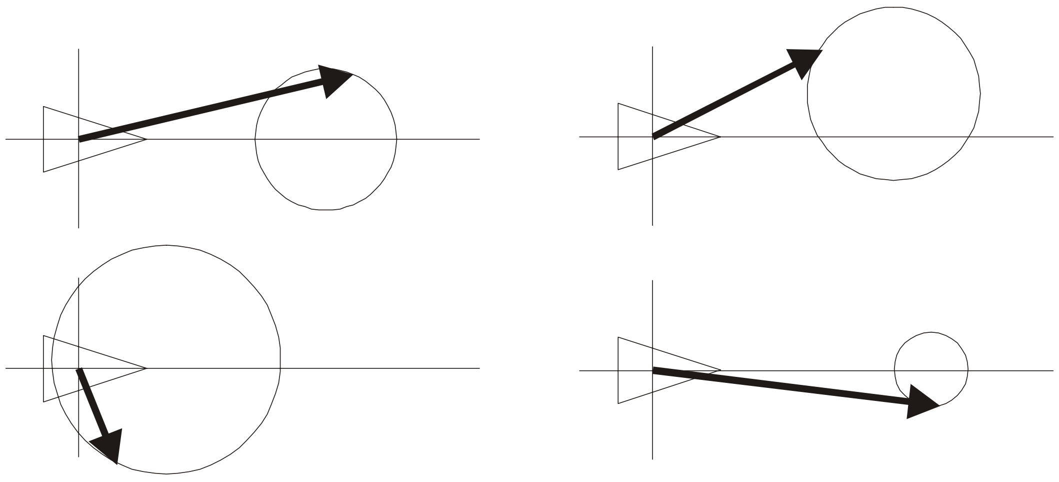

Figure 8: Examples for different attribute settings

Separation

Application

The separation behavior is used to make vehicles keep a certain distance to each other. This helps in preventing collisions of vehicles. Human beings do the same thing automatically. Walking through a crowded place he avoids bumping into other persons without a second thought. Only the persons in the immediate distance are considered in his actions, most of the time this set is reduced to only the people in front of him.

Implementation

The basis for the separation behavior is the theory on electrical charged particles that avoid crowding together by repelling each other. First thing to do is to find all vehicles within a certain distance of our vehicle. This is done using the simulation object 'Neighborhood'. The method

```
getNearVehicles(v, m_nearAreaRadius)
```

returns an iterator over all the vehicles within the specified radius. Then a repelling force is created for each of the neighboring objects. It is the result of subtracting the position of the vehicle v$_2$ from the position of the vehicle v$_1$:

$$\vec{a} = \begin{pmatrix} x_{v1} - x_{v2} \\ y_{v1} - y_{v2} \end{pmatrix}$$

This vector is normalized and scaled to the length $\frac{1}{r}$, where r represents the distance between the two vehicles.

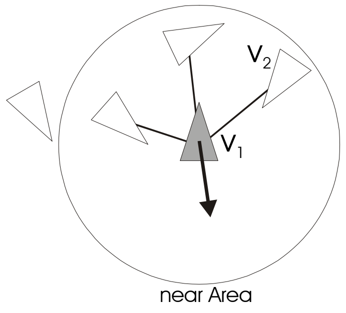

Figure 9: Calculating the steering force for separation

According to the distance between the two objects the force will be stronger if the distance is low and softer if it is farther away. All calculated force are added together and form the steering force. Therefore the return value of the `calculate()` method is defined as: $Steeringvector = \sum \vec{a}_n$

Since the getNearVehicles() method of the neighborhood object also considers objects behind the vehicle in the calculations, strange vehicle movements can result in certain configurations. Somebody driving a car only considers the vehicles in front of him in his actions most of the time. To simulate this, the `lookaheadonly` attribute is checked to see if the getNearVehiclesFront() method should be used to only get the objects in front of the vehicle. This makes the vehicle behind us brake only if it gets too near. For a simulation scene showing the queuing in front of a small passage, setting the `lookaheadonly` flag to true is a necessity.

Alignment

Application

Vehicles using the alignment behavior try to form a group with other vehicles. An application for this is the flocking behavior, which is a combination of this behavior with cohesion and separation.

Implementation

To realize the grouping the vehicle is steered to the average direction of the surrounding vehicles. For this an iterator over all the vehicles within `m_nearAreaRadius` is created.

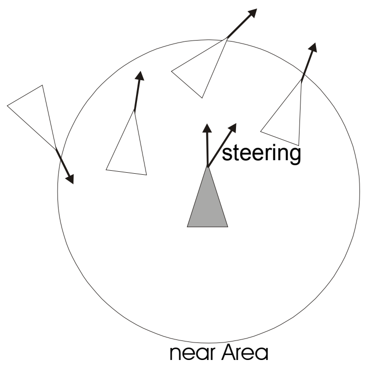

Figure 10: Calculating the force vector for Alignment

The velocity vectors of the resulting set of vehicles are summed up. After that the result is divided by the number of vehicles to calculate the average direction.

The final steering force is determined by scaling with the influence factor `m_influence`.

Cohesion

Application

The cohesion behavior is responsible for keeping the members of a group together. The vehicle is steered to the average position of the neighboring vehicles. Cohesion is part of the flocking behavior.

Implementation

First an interator over the neighboring vehicles is created using the `getNearVehicles(…)` method of the neighborhood object. After that, the positions of the vehicles represented as base vectors are summed up and then divided by the number of found vehicles. The resulting position is the target for the vehicle and the steering force is calculated like in the seek behavior.

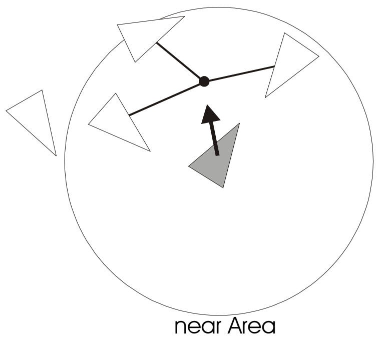

Figure 11: The steering force for Cohesion

Flocking

Application

Flocking provides vehicles with a behavior that gives them the potential to form a group with other vehicles. An example in nature would be swarms of birds. Often hundreds of birds are part of a swarm, flying close together without colliding. They match their velocity and flight direction and in addition try to avoid collisions. Reynolds describes this kind of behavior as a combination of the separation, alignment and cohesion behaviors.

While separation prevents collisions within the swarm, cohesion creates a certain affinity between the vehicles. Vehicles that get near the swarm get sucked in and follow the average motion of the flock. Using the alignment behavior the members steer in the average direction of their flockmates.

Implementation

The steering force is the result of combining the forces generated by the Alignment, Cohesion and Separation behaviors. A vector of length zero is created and the single force vectors are added to it. There are three additional influence factors defined. Using the attributes named “Alignment”, “Cohesion” and “Separation” the distribution of forces within the flocking behavior can be fine-tuned. The denseness of the swarm can be adjusted this way or flock members at the edge of the swarm will not follow the average movement as exact as the other members.

Obstacle Avoidance

Application

To steer a vehicle through a predefined landscape in a realistic way it is necessary to move it the way a spectator would expect it to. Therefore it is necessary to define a behavior that allows the handling of obstacles. In normal life humans and animals avoid bumping into obstacles without gibing it a second thought. When somebody feels the craving for a cold beer, he will not move to the fridge in a straight line. Unconsciously he will choose a path that will lead him around all the obstacles in the world, like tables, chairs and other things lying around in a well kept household, so he is not forced to give up his primary task to nurse his broken toe. The obstacle avoidance behavior implements this unconscious avoiding of predefined obstacles.

Implementation

First thing when implementing the obstacle avoidance behavior is the definition of the obstacle within the virtual environment. Since this simulation is based inside a two dimensional scene, using simple geometric forms as substitution for complex real life obstacles. The possibly simplest representation of an obstacle is the circle. The circle should enclose the whole obstacle. It should be taken care not to waste too much unnecessary space on the representation. For long walls a circle can be a really bad approximation. The advantages in using a circle are the simple formulas used in determining intersections with other geometric bodies since those are used a lot in the behavior. A disadvantage is the bad approximation of the complex shape. Unfortunately this is the case for most shapes used in this simulation.

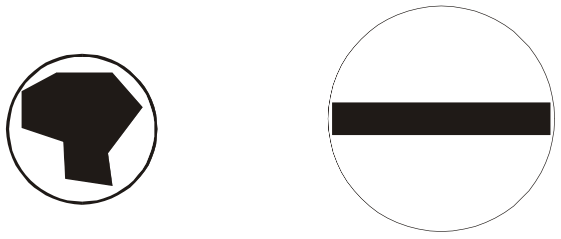

Figure 12: Example for good and bad approximation using a circle

Since the disadvantages of using circles as placeholders clearly outweigh the advantages, the choice was made to use polygons instead. The advantage being that elements of real life, like chairs or tables for instance, can be better approximated. The results achieved in two-dimensional space can be applied to three-dimensional space without any problems.
The first step for the Obstacle Avoidance behavior is to find out which objects are to be found in its close vicinity. To get this information it queries the Neighborhood object whose job is to speed up spatial queries. By reducing the number of objects to be tested in the first step further test are sped up already. In large scenes the time used for making the spatial query is certainly less than the time used for the following intersection tests.

Calculating the angle of intersection and the distance

The `getCollisionDetails(..)` function is used to calculate the angle of intersection and the distance to the obstacle. The steering vector is based on the resulting values. To achieve this the testing vector is first enlarged by the distance to the intersection and after that projected vertically back onto the edge of the polygon. This vector is then used for the steering force.

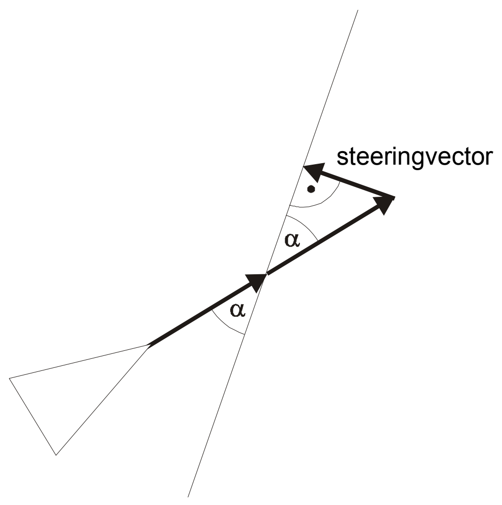

Figure 13: Steering force to avoid an obstacle

Optimization using "exclusion tests"

If there are obstacles within close vicinity of the vehicle is does not necessary imply that the testing vector is actually intersecting all of them. For instance if its position to one of the sides of the vehicle it will definitely not intersect the testing vector.
Since the Obstacle Avoidance behavior is used by many vehicles and calculating the steering force is a very computation intensive task, using so called "exclusion tests" can significantly speed up calculations. This kind of test only consist of simple "if" statements and are therefore pretty fast. For each edge of an obstacle these tests have to be performed.

Test 1:

First thing is calculating a test point P at the end of the testing vector. Now all edges with points outside of the defined area can be removed from further testing (see picture).

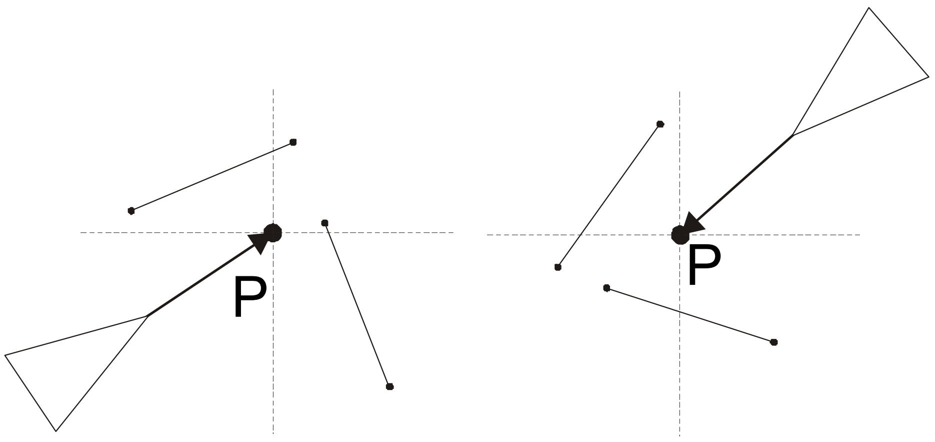

Figure 14: Edges that can be discarded after the first test

Test 2:

If the edge has passed the first test it will be transformed into the local space of the vehicle. The testing vector now represents the x-axis. If both endpoints of the edge are either above or below the x-axis, the edge does not intersect the testing vector. These edges can be discarded.

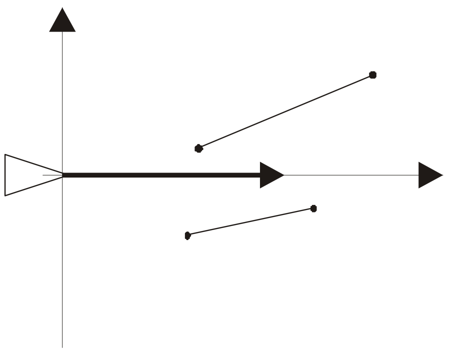

Figure 15: Edges discarded in the second test

Test 3:

The third test is used to determine if the backside of the edge is facing the vehicle. This is often the case with small obstacles. These edges can be discarded as well. The edge points of a polygon are defined counterclockwise. Therefore edges with point 1 above and point 2 below the x-axis can be disregarded.

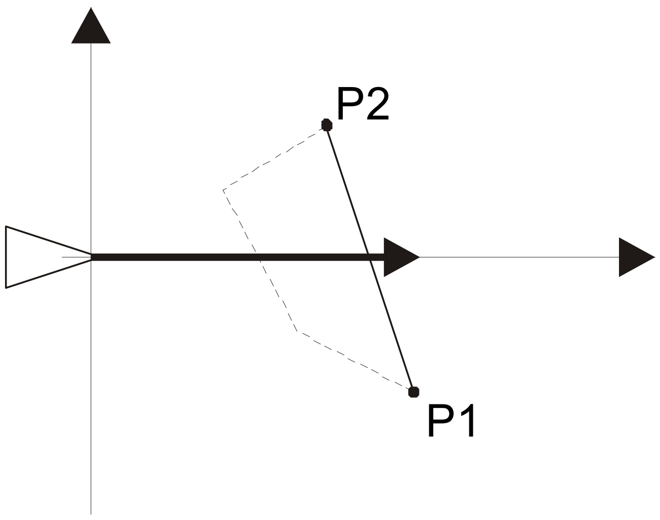

Figure 16: Edges discarded in the third test

Test 4:

The fourth test is used to determine if both of the ends of the edge have x positions greater than the length of the testing vector. Also edges having x positions left of the y-axis can be discarded. In this case the obstacle is behind the vehicle.

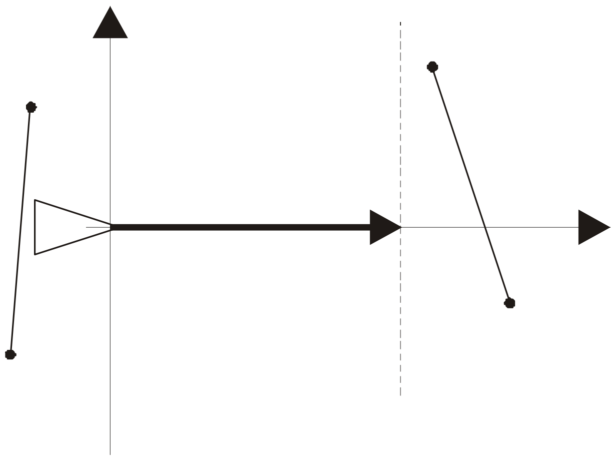

Figure 17: Edges discarded in the fourth test

After these four tests most of the edges have been discarded and some computation time has been saved. For the remaining edges the angle and distance for the intersection point has to be calculated. Basis for this is the formula: $y = a(x - x_1) + y_1$

Here x_1 and y_1 define the coordinates of the first edge point. The slope a is calculated using $a = \frac{(y_2 - y_1)}{(x_2 - x_1)}$. If this is inserted into the first formula, solved for x and y is set to zero, the resulting formula will be $x = \frac{(-y_1) \cdot (x_2 - x_1)}{(y_2 - y_1)} + x_1$

The resulting value for x is the distance of the intersection with the obstacle form the vehicle. To get the angle the arcustangens function is used.

Comment:

Since obstacles can be overlapping, all edges of these obstacles have to be considered in the tests. The edge with the smallest distance for the intersection is used for calculating the steering force. Therefore always the nearest obstacle is used.

Containment

Application

The Containment behavior is an enhanced version of the Obstacle Avoidance behavior. Instead of using just one testing vector for collision detection, two additional vectors are added resulting in: one vector testing the front, one to the left and one to the right.

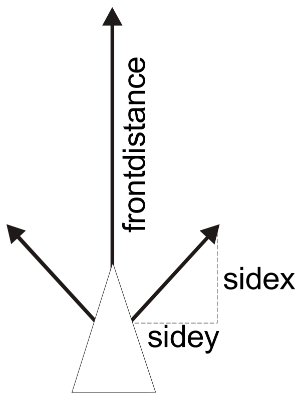

Figure 18: Testvectors for containment

The vectors are parameterized with the attributes `sidex`, `sidey` and `frontdistance` (see picture). Each of these vectors is tested for intersection with an obstacle. If there is an imminent collsion, the angle of intersection and the distance to the intersection is calculated. According to these values a steering force is determined to steer the vehicle away from the obstacle.

Implementation

As already stated, the intersection angle and distance has to be calculated for each of the three testing vectors. The `getCollsionDetails(…)` function is used for this. By specifying the vehicle, the obstacles within a certain range and a testing vector the function performs all necessary calculations and returns the intersection distance and angle and the two endpoints of the intersection edge. To speed up calculation time, the Neighborhood object is queried for the obstacles within the length of the testing vector. This way only a reduced subset of the obstacles in the scene is used in the intersection tests.
The advantage in using three testing vectors instead of just a single one is the improved handling of situations where the vehicle has to skim along obstacles. When using ObstacleAvoidance situations may occur where the vehicle is already inside the obstacle before a collision is detected. The two vectors to the side of the vehicle specifically test the area neglected by ObstacleAvoidance and allow for improved handling of complex scenes.

Containment using Fuzzy-Logic

Application

Contrary to the simple Containment behavior (see chapter 2.2.8), the fuzzy version determines the steering vector based on fuzzy logic. As input values the angle of intersection and the distance is used. To get these values the `getCollisionDetails(…)` function is called.

The fuzzy sets

Fuzzy systems encode rules in a numerical way. The fuzzy set theory is based on the assumption that all things only apply to a certain degree. Therefore three different definitions can represent the distance to the obstacle:
small distance (PS)
middle distance (PM)
big distance (PB)

The diffuse borders of these three distance definitions can be displayed with the following diagram:

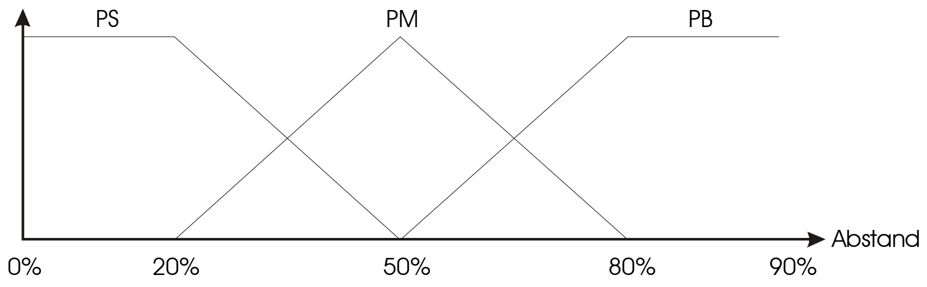

Figure 19: Fuzzy sets for the distance

It can be seen that a distance of 50% of the length of the testing vector is considered as middle distance. A distance of 75% results in a half middle and half small distance. The closer to an obstacle the stronger should the steering force be.
Also the angle of intersection is an important factor in determining how strong the vehicle should evade. A driver that is only slightly of the track makes small corrections while driving into a frontal obstacle makes him do strong evasive maneuvers.

The intersection angle is divided into five parts:

Negative middle (NM)
Negative small (NS)
Perpendiculary, or zero (ZE)
Positive small (PS)
Positive middle (PM)

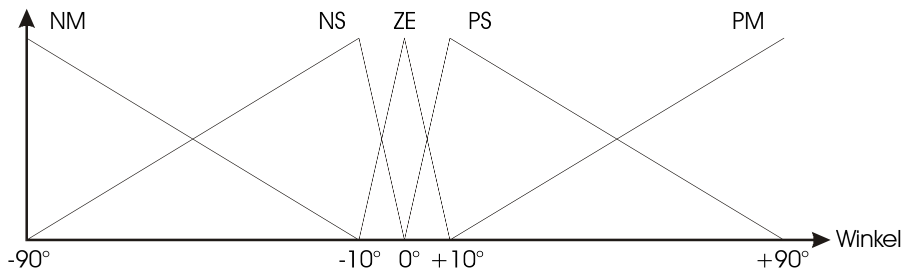

Figure 20: Fuzzy sets for the angle

The rule table

To be able to act according to the entry values it is necessary to define a rule table. It defines the resulting value according to the combination of distance and angle values. The rule table for the FuzzyContainment looks like this:

Distance ↓ / ← Angle

	NM	NS	ZE	PS	PM
PS	ZE	PS	PS	NS	ZE
PM	ZE	PS	ZE	NS	ZE
PB	ZE	PS	ZE	NS	ZE

The defuzzification

The result of the defuzzification is an angle between –90 and +90 degrees describing the direction for the evasive action. The length of the steering vector is predefined and always constant.

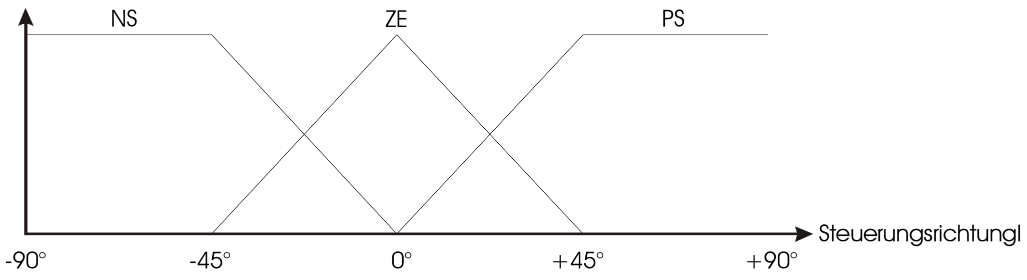

Figure 21: The defuzzification

Example avoidance situation

For example the vehicle is in the situation as seen in picture 22. A distance of 60% and an angle of 36° is measured.

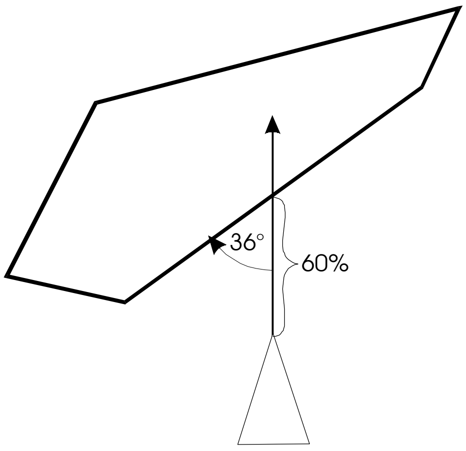

Figure 22: Vehicle in an avoidance situation

These values are now used as entry values for the fuzzy sets.

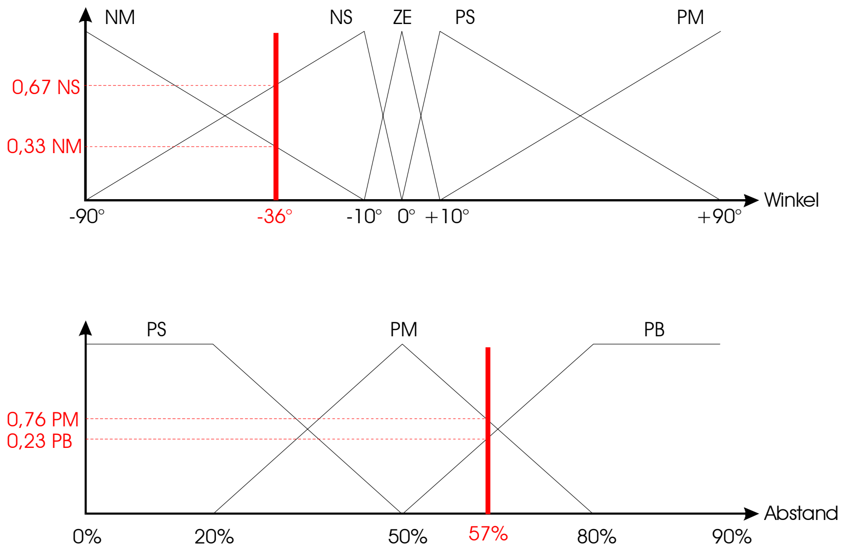

Figure 23: Example for a fuzzy set

The resulting values of 0.67 NS and 0.33NM for the angle, 0.76 PM and 0.23 PB for the distance are analysed using the rule table and drawn into the defuzzification diagram.

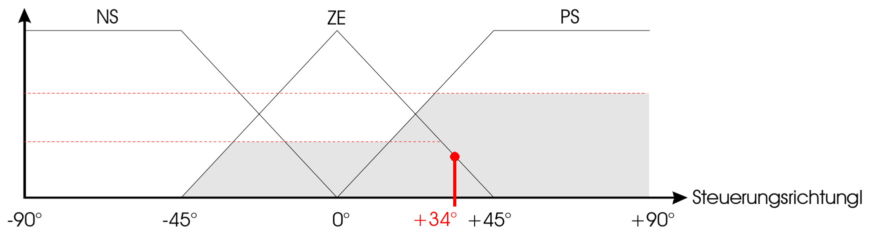

Figure 24: Defuzzification

After calculating the center of gravity for the enclosed area the resulting angle of 34° describes the steering direction for the vehicle.

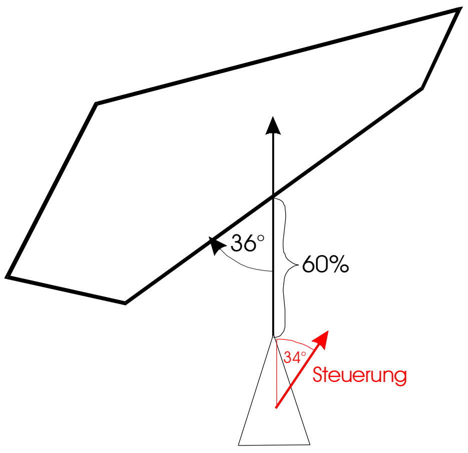

Figure 25: The steering force generated by the fuzzy logic system

Problems occur with this configuration of the fuzzy sets when encountering concave obstacles. Since the behavior only steers to the left or right and does not produce any recoiling force it will get stuck if the angle between two edges is below a certain threshold. The problem is that the behavior first recognizes the edge to the left and steers to the right. Then it encounters the edge to the right and steers back to the left (see picture 26).

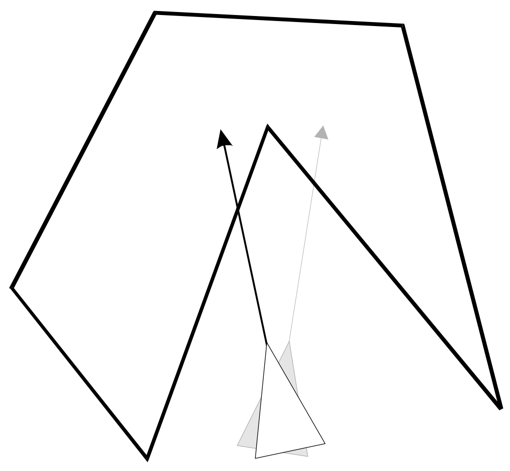

Figure 26: Problems with concave obstacles

Using a slightly different fuzzy set configuration might reduce this effect. Another possibility would be to use the testing vectors to the right and left of the vehicle to determine this sort of narrow passage and act accordingly.

SimplePathFollowing

Application

Another behavior described in Reynolds work is called “PathFollowing”. Here the vehicle is steered along a predefined path. The path itself is made out of single waypoints that have to be visited in a certain predefined order. Robin Green describes three possible ways of handling these waypoints. They are called “Dog on a string”, “Reprojection” and “Grappling Hooks”. The easiest way to follow a path is the last one of the three.

Implementation

The “Dog on a string” way of following a path uses a fixed destination for the vehicle. It is moved along the path linearly with a predefined velocity. The problem here is choosing the right velocity. If chosen too low the same phenomena of the vehicle circling around the destination point like a moth around a light source as seen in the seek behavior will occur here as well. If it

is chosen too fast there might be situations where the vehicle will try to steer through obstacles since the destination point has already passed the edge.
The so called “Reprojection” method looks for the next intersection with the path along the current velocity vector. The vehicle does look a bit more intelligent that way since it does not always simply go through the waypoints starting at the first one but can start out with a later one according with the current position of the vehicle. Loops in the path can be a problem with this method. One has to make sure that the vehicle does not suddenly follow the path in the wrong direction or important parts of the path are not covered.
The “Grappling Hooks” method works best if there is a clear line of sight between two consecutive waypoints. The vehicle simply steers to the first waypoint. After reaching it, the next waypoint is selected and steered to. The SimplePathFollowing behavior implements this method. The TileNeighborhood class handles the path creation. It is provided with the starting and ending point of the path. It then automatically generates the necessary waypoints for the vehicle. The SimplePathFollowing behavior simply accesses the waypoints one after the other steers the vehicle to the desired destination.

Mind

Introduction

The Mind class implements the “action selection” part of the Steering Behaviors logic as described in Reynolds paper. It is responsible for determining the final steering force based on predefined rules. Therefore it plays the role of the mind of the vehicle. The single behaviors like Seek, ObstacleAvoidance or Separation can be considered as sensors. Each behavior can analyze its surroundings in a special way and create a steering force based on its internal logic. This force is based only on the way the behavior makes assumptions about its surroundings. The Mind class receives the “suggestions” from the behaviors and decides what kind of decision has to be made. There are several possible ways of implementing the decision making process each with its advantages and disadvantages.

SimpleMind

The SimpleMind class implements the easiest way of deciding on the combined steering force. Each behavior has an attribute called “influence” (called “`m_influence`” in the source code). According to the value of this attribute the influence of the behavior on the final steering force is higher or lower. The steering force generated by the behavior is scaled by the influence factor. The final force is the combination of the single forces scaled by their influence value. Using this system the behaviors can be adjusted according to priority.

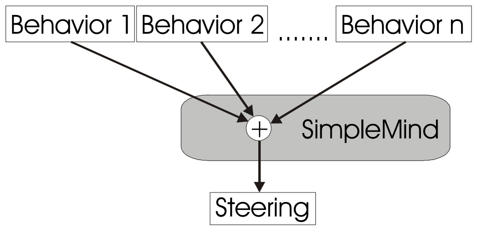

Figure 27: Workings of the SimpleMind

For more important behaviors like Separation or ObstacleAvoidance a higher influence value is chosen. A behavior like Wander, which is not so important in most of the cases, is normally set to have a lower influence. This way it is assured that for example in front of an obstacle the major part of the steering force is made of the ObstacleAvoidance force.

PriorityMind

The PriorityMind class can be used to suppress the forces from one or more behaviors in certain situations. It also uses the influence attribute of the behaviors to achieve this goal. Its value represents the percentage of the maximum vehicle steering force the behavior can generate. A value of for example 0.8 represents 80%. Since this value can be freely chosen values higher than 100% are possible. The PriorityMind class distributes the maximum usable force to the single behaviors. When the maximum force is reached, the behavior is left with either using the remaining percentage if there is any left or it is ignored.

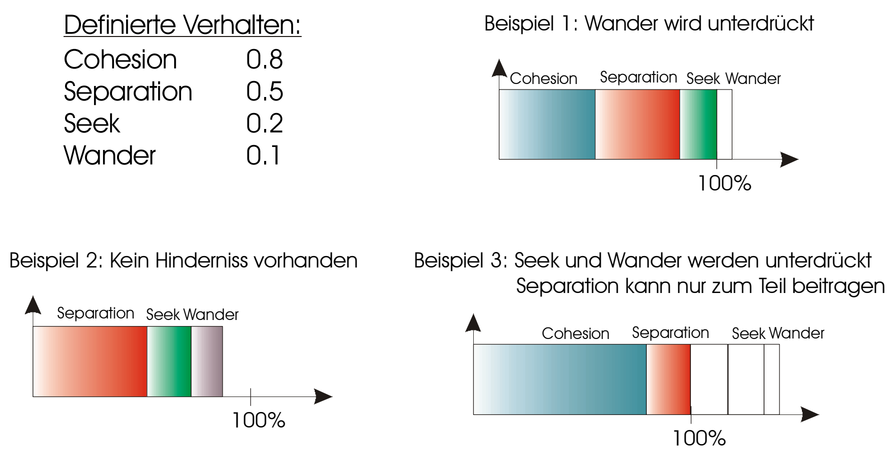
Definierte Verhalten:
Cohesion 0.8
Separation 0.5
Seek 0.2
Wander 0.1
Beispiel 1: Wander wird unterdrückt
Cohesion
Separation
Seek
Wander
100%
Beispiel 2: Kein Hinderniss vorhanden
Separation
Seek
Wander
100%
Beispiel 3: Seek und Wander werden unterdrückt
Separation kann nur zum Teil beitragen
Cohesion
Separation
Seek
Wander
100%

Figure 28: Examples for force distributions using PriorityMind

Using the PriorityMind class is advantageous in situations where several vehicles are traveling within a very tight area. Choosing the influence values in a skillful way ensures that the vehicles do not overlap and obstacles are not ignored.

Simulations

Introduction

The basis for all simulations is the Simulation class. It is the basis for creating complex systems. First versions of the Steering Behavior applets had the simulation code tightly integrated into the applet code. This was not very flexible and lead to problems when enhancements were considered that were not compatible with earlier versions. By moving the common features into its own class hierarchy it is much easier to enhance the system without having to modify previous versions.

Simulation class

The base class for the hierarchy of simulations is the Simulation class. It defines the common interface for all derived classes. To make the whole system more flexible the base class has two arrays with so called pre- and post-simulations.

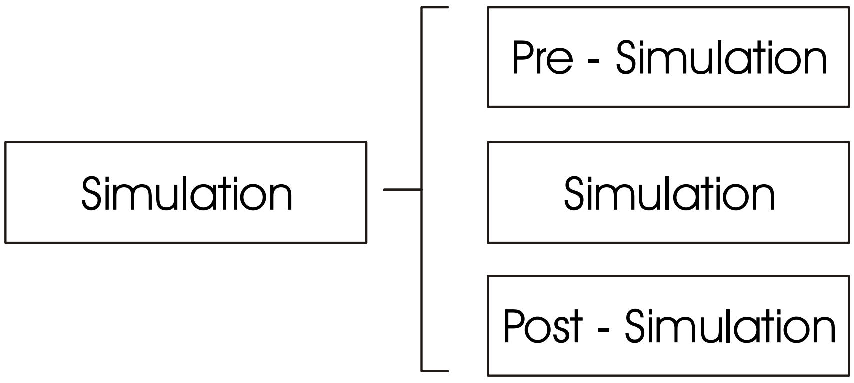

Figure 29: Division of the simulation class

A pre-simulation is called before the actual simulation step. This can be used for any kind of simulations that need to initialize data before behaviors can access it. One example is the Neighborhood class that has to initialize its distance matrix before it can be used.
A post-simulation is used for any clean up operations after the main simulation step has been performed. An example is the VehicleInfo class. It is used for displaying the influence of the single behaviors on the combined behavior and therefore needs its information after the main simulation step.

Neighborhood

Overview

The Neighborhood class is one of the most important classes used in the simulation. It is responsible for the correct working of behaviors like Alignment, Cohesion and all other behaviors used for the avoidance of other objects. The main functionality is to provide functionality to make spatial queries within the simulation environment. Searches can be made for vehicles as well as for obstacles within a certain radius.

Inner workings

The Neighborhood class uses two different way to provide spatial queries for vehicles and obstacles. Each method has its advantages and disadvantages but both fulfill the requirement of providing results as fast as possible.

The query for vehicles is based upon a distance matrix. This matrix holds the information about the distance between each vehicle. When handling a query the row of the specified vehicle is selected and the distances are compared with the threshold value. The distance is based on the center point of each object. To speed up the construction and updates of the distance matrix, several techniques have been used.

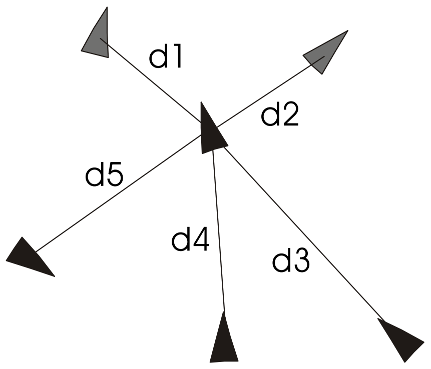

Figure 30: Distance calculations for Neighborhood

The distance between two vehicles in two-dimensional space is calculated using the formula $d = \sqrt{x^2 + y^2}$. The square root is the slowest part of this calculation. To avoid this part of the calculation all queries use the quadratic distance between the vehicles. Therefore the distance is calculated as $d^2 = x^2 + y^2 = \left(\sqrt{x^2 + y^2}\right)^2$. There is no difference in the results of the spatial queries using the new formula. The advantage lies in saving a call to the square root function. Even when the square root is calculated using the mathematical coprocessor there is still a slight speed up when omitting the call. It is important to square the distance threshold as well, otherwise the comparisons would not be within the same coordinate system.

	Fahrzeug 1	Fahrzeug 2	…	Fahrzeug x
Fahrzeug 1	X	d1-2	…	d1-x
Fahrzeug 2	d1-2	X	…	d2-x
…	…	…	X	…
Fahrzeug x	d1-x	d2-x	…	X

Figure 31: Symmetrical nature of the distance matrix

Another way of optimizing is in taking advantage of the symmetrical nature of the distance matrix. Since the vehicles in the rows are the same as the vehicles in the columns is the distance between vehicle A and vehicle B present in row A and column B and in row B and column A as well. By taking advantage of this feature the number of calculations can be reduced by 50%. Only the upper right half of the matrix is calculated, the other entries result from this automatically.

The disadvantage of the distance matrix is its size. It is based on the number of vehicles in a scene. The matrix consists of $n = Vehicles^2$ elements. For each iteration of the simulation loop $n = \frac{Vehicles^2}{2}$ distances have to be calculated. Therefore the calculations will get slower based on the number of vehicles in a simulation.

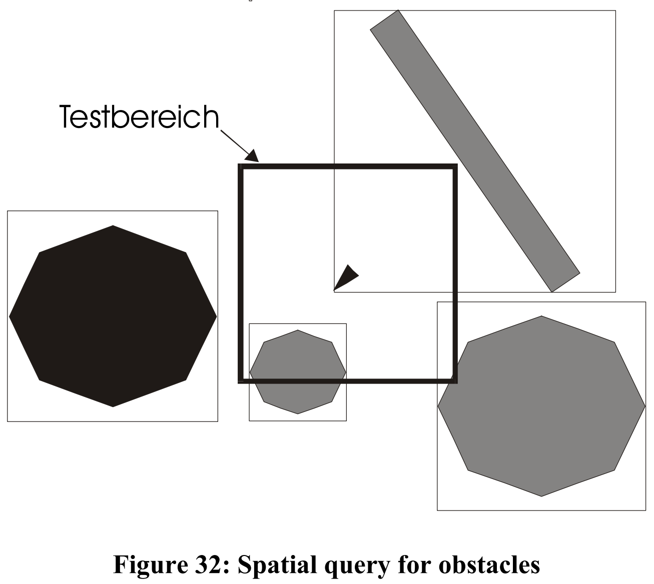

Figure 32: Spatial query for obstacles

The spatial query for obstacles uses a different system to find the objects within a certain distance. The distance does not describe a circle around the vehicle but defines half the width and height of a rectangle centred on the vehicle. Each obstacle has an encompassing rectangle defined. An object is considered to be within the search distance if its encompassing rectangle intersects the testing rectangle or totally encompasses it. Since everything is defined using rectangles by checking the x and y positions of the rectangles. An obstacle can be discarded if the y positions of the encompassing rectangle are either both smaller than the minimum y values of the rectangle around the vehicle or both larger than the maximum y values. The same is true for the x positions. In all other cases the rectangle are intersecting or one overlaps the other.

A problem with this method is that every single obstacle in a scene has to be tested. Additionally it is not very exact since the enclosing rectangles are only a rough approximation of the actual shape. Even with these disadvantages is the method very fast and simple to use.