

How to Create Autonomously Moving Game Agents

During the late '80s I remember watching a BBC Horizon documentary about state-of-the-art computer graphics and animation. There was lots of exciting stuff covered in that program, but the thing I remember most vividly was an amazing demonstration of the flocking behavior of birds. It was based on very simple rules, yet it looked so spontaneous and natural and was mesmerizing to watch. The programmer who designed the behavior is named Craig Reynolds. He called the flocking birds “boids,” and the simple rules the flocking behavior emerged from he called “steering behaviors.”

Since that time Reynolds has published a number of articles on various types of steering behaviors, all of them fascinating. Most, if not all, of his steering behaviors have direct relevance to games, which is why I’m going to spend a considerable amount of time describing them and showing you how to code and use them.

What Is an Autonomous Agent?

I’ve seen many definitions for what an autonomous agent is, but probably the best is this:

An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

Throughout this chapter I will use the term “autonomous agent” in reference to agents that possess a degree of autonomous *movement*. If an autonomous agent stumbles upon an unexpected situation, like finding a wall in its way, it will have the ability to respond and adjust its motion accordingly. For example, you might design one autonomous agent to behave like a rabbit and one like a fox. If while munching happily on the fresh dewy grass, the rabbit happens to spot the fox, it will autonomously attempt to evade it. At the same time the fox will autonomously pursue the rabbit. Both these events occur without any further intervention from the

programmer; once up and running, autonomous agents simply look after themselves.

This is not to say that an autonomous agent *should* be able to cope with absolutely any situation at all (although that might be one of your goals), but it is often very useful to be able to bestow an *amount* of autonomy. For example, a common problem when writing pathfinding code is how to deal with dynamic obstacles. Dynamic obstacles are those objects in your game world that move around or change position, like other agents, sliding doors, and so forth. Given a suitable environment, incorporating the correct steering behavior into a game character will preclude writing special pathfinding code to handle dynamic obstacles — an autonomous agent will have the ability to deal with them if and when it has to.

The movement of an autonomous agent can be broken down into three layers:

- **Action Selection:** This is the part of the agent's behavior responsible for choosing its goals and deciding what plan to follow. It is the part that says “go here” and “do A, B, and then C.”
- **Steering:** This layer is responsible for calculating the desired trajectories required to satisfy the goals and plans set by the action selection layer. Steering behaviors are the implementation of this layer. They produce a steering force that describes where an agent should move and how fast it should travel to get there.
- **Locomotion:** The bottom layer, locomotion, represents the more mechanical aspects of an agent's movement. It is the *how* of traveling from A to B. For example, if you had implemented the mechanics of a camel, a tank, and a goldfish and then gave a command for them to travel north, they would all use different mechanical processes to create motion even though their intent (to move north) is identical. By separating this layer from the steering layer, it's possible to utilize, with little modification, the same steering behaviors for completely different types of locomotion.

Reynolds makes use of an excellent analogy to describe the roles of each of these layers in his paper “Steering Behaviors for Autonomous Characters.”

“Consider, for example, some cowboys tending a herd of cattle out on the range. A cow wanders away from the herd. The trail boss tells a cowboy to fetch the stray. The cowboy says ‘giddy-up’ to his horse and guides it to the cow, possibly avoiding obstacles along the way. In this example, the trail boss represents action selection: noticing that the state of the world has changed (a cow left the herd) and setting a goal (retrieve the stray). The steering level is represented by the cowboy, who decomposes the goal into a series of simple sub-goals (approach the cow, avoid obstacles, retrieve the cow). A sub-goal corresponds to a steering behavior for the cowboy-and-horse team. Using various control

The Vehicle Model

signals (vocal commands, spurs, reins), the cowboy steers his horse toward the target. In general terms, these signals express concepts like: go faster, go slower, turn right, turn left, and so on. The horse implements the locomotion level. Taking the cowboy's control signals as input, the horse moves in the indicated direction. This motion is the result of a complex interaction of the horse's visual perception, its sense of balance, and its muscles applying torques to the joints of its skeleton. From an engineering point of view, legged locomotion is a very hard problem, but neither the cowboy nor the horse give it a second thought."

Not all is rosy and sweet in the world of autonomous agents though. The implementation of steering behaviors can beset the programmer with a truckload of new problems to deal with. Some behaviors may involve heavy manual tweaking, while others have to be carefully coded to avoid using large portions of CPU time. When combining behaviors, care usually must be taken to avoid the possibility that two or more of them may cancel each other out. There are means and ways around most of these problems though (well, all except for the tweaking — but that's fun anyway), and most often the benefits of steering behaviors far outweigh any disadvantages.

The Vehicle Model

Before I discuss each individual steering behavior I'm going to spend a little time explaining the code and class design for the vehicle model (the locomotion). `MovingEntity` is a base class from which all moving game agents are derived. It encapsulates the data that describes a basic vehicle with point mass. Let me run you through the class declaration:

```
class MovingEntity : public BaseGameEntity
{
protected:
```

The `MovingEntity` class is derived from the `BaseGameEntity` class, which defines an entity with an ID, a type, a position, a bounding radius, and a scale. All game entities from here onward in this book will be derived from `BaseGameEntity`. A `BaseGameEntity` also has an additional Boolean member variable, `m_bTag`, which will be utilized in a variety of ways, some of which will be described very shortly. I'm not going to list the class declaration here, but I recommend you take a quick look at the `BaseGameEntity.h` header sometime during your read through this chapter.

```
SVector2D    m_vVelocity;

//a normalized vector pointing in the direction the entity is heading.
SVector2D    m_vHeading;

//a vector perpendicular to the heading vector
SVector2D    m_vSide;
```

The heading and side vectors define a local coordinate system for the moving entity. In the examples given in this chapter, a vehicle's heading will always be aligned with its velocity (for example, a train has a velocity aligned heading). These values will be used often by the steering behavior algorithms and are updated every frame.

```
double      m_dMass;

//the maximum speed at which this entity may travel.
double      m_dMaxSpeed;

//the maximum force this entity can produce to power itself
//(think rockets and thrust)
double      m_dMaxForce;

//the maximum rate (radians per second) at which this vehicle can rotate
double      m_dMaxTurnRate;

public:

    /* EXTRANEOUS DETAIL OMITTED */
};
```

Although this is enough data to represent a moving object, we still need a way of giving a moving entity access to the various types of steering behaviors. I have chosen to create a class, `Vehicle`, which inherits from `MovingEntity` and owns an instance of the steering behavior class, `SteeringBehaviors`. `SteeringBehaviors` encapsulates all the different steering behaviors I'll be discussing throughout this chapter. More on that in a moment though; first, let's take a look at the `Vehicle` class declaration.

```
class Vehicle : public MovingEntity
{
private:

    //a pointer to the world data enabling a vehicle to access any obstacle
    //path, wall, or agent data
    GameWorld*      m_pWorld;
```

The `GameWorld` class contains all the data and objects pertinent to the environment the agents are situated in, such as walls, obstacles, and so on. I won't list the declaration here to save space, but it might be a good idea to check out `GameWorld.h` in your IDE at some point to get a feel for it.

```
//the steering behavior class
SteeringBehaviors* m_pSteering;
```

A vehicle has access to all available steering behaviors through its own instance of the steering behavior class.

```
public:

    //updates the vehicle's position and orientation
    void      Update(double time_elapsed);
```

The Vehicle Model

```
/* EXTRANEOUS DETAIL OMITTED */
};
```

You can see the class relationships clearly in the simplified UML diagram shown in Figure 3.1.

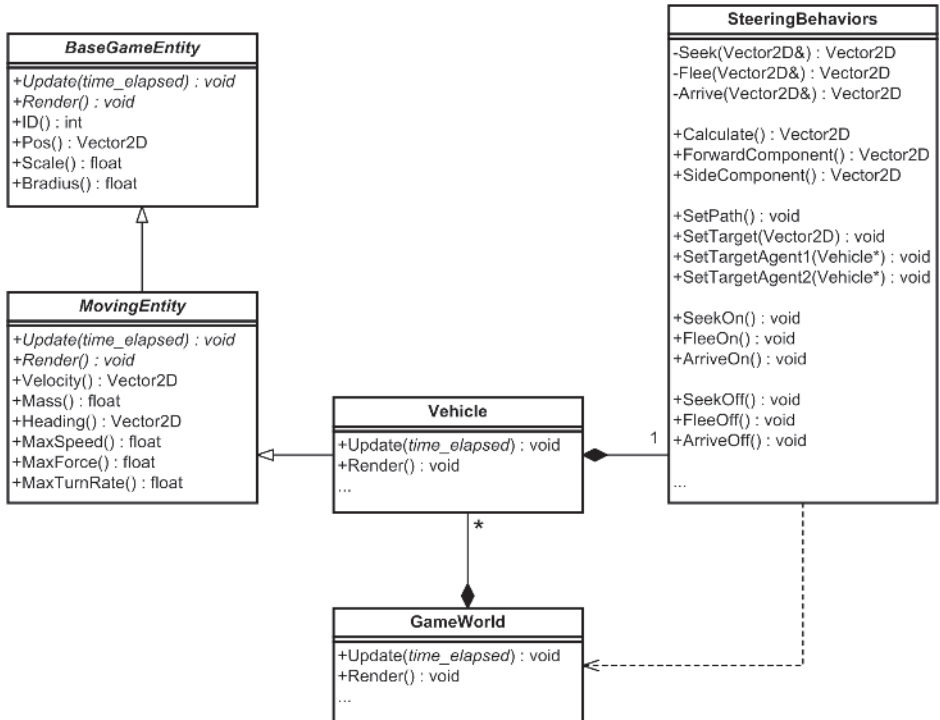


Figure 3.1. The Vehicle and SteeringBehaviors class relationships

Updating the Vehicle Physics

Before we move on to the steering behaviors themselves, I'd just like to walk you through the `Vehicle::Update` method. It's important that you understand every line of code in this function because it's the main work-horse of the `Vehicle` class. (If you do not know Newton's laws of force and motion, I would strongly recommend you read the relevant part of Chapter 1 before continuing.)

```
bool Vehicle::Update(double time_elapsed)
{
    //calculate the combined force from each steering behavior in the
    //vehicle's list
    SVector2D SteeringForce = m_pSteering->Calculate();
```

First the steering force for this simulation step is calculated. The `Calculate` method sums all a vehicle's active steering behaviors and returns the total steering force.

```
//Acceleration = Force/Mass
SVector2D acceleration = SteeringForce / m_dMass;
```

Using Newton's laws of physics, the steering force is converted into an acceleration (see equation 1.93, Chapter 1).

```
//update velocity
m_vVelocity += acceleration * time_elapsed;
```

Using the acceleration, the vehicle's velocity can be updated (see equation 1.81, Chapter 1).

```
//make sure vehicle does not exceed maximum velocity
m_vVelocity.Truncate(m_dMaxSpeed);

//update the position
m_vPos += m_vVelocity * time_elapsed;
```

The vehicle's position can now be updated using the new velocity (see equation 1.77, Chapter 1).

```
//update the heading if the vehicle has a velocity greater than a very small
//value
if (m_vVelocity.LengthSq() > 0.00000001)
{
    m_vHeading = Vec2DNormalize(m_vVelocity);

    m_vSide = m_vHeading.Perp();
}
```

As mentioned earlier, a `MovingEntity` has a local coordinate system that must be kept updated each simulation step. A vehicle's heading should always be aligned with its velocity so this is updated, making it equal to the normalized velocity vector. But — and this is important — *the heading is only calculated if the vehicle's velocity is above a very small threshold value*. This is because if the magnitude of the velocity is zero, the program will crash with a divide by zero error, and if the magnitude is non-zero but very small, the vehicle may (depending on the platform and operating system) start to move erratically a few seconds after it has stopped.

The side component of the local coordinate system is easily calculated by calling `SVector2D::Perp`.

```
//treat the screen as a toroid
WrapAround(m_vPos, m_pWorld->cxClient(), m_pWorld->cyClient());
}
```

Finally, the display area is considered to wrap around from top to bottom and from left to right (if you were to imagine it in 3D it would be toroidal — doughnut shaped). Therefore, a check is made to see if the updated

The Steering Behaviors

position of the vehicle has exceeded the screen boundaries. If so, the position is wrapped around accordingly.

That's the boring stuff out of the way — let's move on and have some fun!

The Steering Behaviors

I'm now going to describe each steering behavior individually. Once I've covered all of them I'll explain the `SteeringBehaviors` class that encapsulates them and show you the different methods available for combining them. Toward the end of the chapter I'll demonstrate a few tips and tricks for getting the most out of steering behaviors.

Seek

The **seek** steering behavior returns a force that directs an agent toward a target position. It is very simple to program. The code looks like this (note that `m_pVehicle` points to the `Vehicle` that owns the `SteeringBehaviors` class):

```
Vector2D SteeringBehaviors::Seek(Vector2D TargetPos)
{
    Vector2D DesiredVelocity = Vec2DNormalize(TargetPos - m_pVehicle->Pos())
        * m_pVehicle->MaxSpeed();

    return (DesiredVelocity - m_pVehicle->Velocity());
}
```

First the *desired velocity* is calculated. This is the velocity the agent would need to reach the target position in an ideal world. It represents the vector from the agent to the target, scaled to be the length of the maximum possible speed of the agent.

The steering force returned by this method is the force required, which when added to the agent's current velocity vector gives the desired velocity. To achieve this you simply subtract the agent's current velocity from the desired velocity. See Figure 3.2.

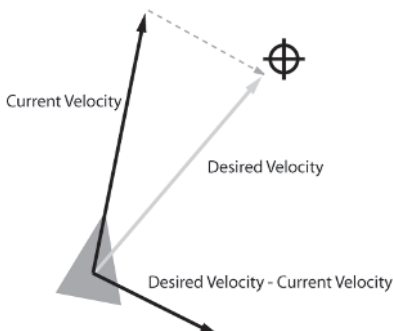


Figure 3.2. Calculating vectors for the seek behavior. The dotted vector shows how the addition of the steering force to the current velocity produces the desired result.

You can observe this behavior in action by running the `Seek.exe` executable. Click with the left mouse button to alter the position of the target. Notice how the agent will overshoot the target and then turn around to approach again. The amount of overshoot is determined by the ratio of `MaxSpeed` to `MaxForce`. You can change the magnitude of these values by pressing the `Ins/Del` and `Home/End` keys.

Seek comes in handy for all sorts of things. As you'll see, many of the other steering behaviors will make use of it.

Flee

Flee is the opposite of **seek**. Instead of producing a steering force to steer the agent toward a target position, **flee** creates a force that steers the agent away. Here's the code:

```
Vector2D SteeringBehaviors::Flee(Vector2D TargetPos)
{
    Vector2D DesiredVelocity = Vec2DNormalize(m_pVehicle->Pos() - TargetPos)
        * m_pVehicle->MaxSpeed();

    return (DesiredVelocity - m_pVehicle->Velocity());
}
```

Note how the only difference is that the `DesiredVelocity` is calculated using a vector pointing in the opposite direction (`m_pVehicle->Pos() - TargetPos` instead of `TargetPos - m_pVehicle->Pos()`).

Flee can be easily adjusted to generate a fleeing force only when a vehicle comes within a certain range of the target. All it takes is a couple of extra lines of code.

```
Vector2D SteeringBehaviors::Flee(Vector2D TargetPos)
{
    //only flee if the target is within 'panic distance'. Work in distance
    //squared space.
    const double PanicDistanceSq = 100.0 * 100.0;
    if (Vec2DDistanceSq(m_pVehicle->Pos(), target) > PanicDistanceSq)
    {
        return Vector2D(0,0);
    }

    Vector2D DesiredVelocity = Vec2DNormalize(m_pVehicle->Pos() - TargetPos)
        * m_pVehicle->MaxSpeed();

    return (DesiredVelocity - m_pVehicle->Velocity());
}
```

Notice how the distance to the target is calculated in distance squared space. As you saw in Chapter 1, this is to save calculating a square root.

Arrive

Seek is useful for getting an agent moving in the right direction, but often you'll want your agents to come to a gentle halt at the target position, and as you've seen, **seek** is not too great at stopping gracefully. **Arrive** is a behavior that steers the agent in such a way it *decelerates* onto the target position.

In addition to the target, this function takes a parameter of the enumerated type `Deceleration`, given by:

```
enum Deceleration{slow = 3, normal = 2, fast = 1};
```

Arrive uses this value to calculate how much time the agent desires to take to reach the target. From this value we can calculate at what speed the agent must travel to reach the target position in the desired amount of time. After that, the calculations proceed just like they did for **seek**.

```
Vector2D SteeringBehaviors::Arrive(Vector2D TargetPos,
                                   Deceleration deceleration)
{
    Vector2D ToTarget = TargetPos - m_pVehicle->Pos();

    //calculate the distance to the target position
    double dist = ToTarget.Length();

    if (dist > 0)
    {
        //because Deceleration is enumerated as an int, this value is required
        //to provide fine tweaking of the deceleration.
        const double DecelerationTweaker = 0.3;

        //calculate the speed required to reach the target given the desired
        //deceleration
        double speed = dist / ((double)deceleration * DecelerationTweaker);

        //make sure the velocity does not exceed the max
        speed = min(speed, m_pVehicle->MaxSpeed());

        //from here proceed just like Seek except we don't need to normalize
        //the ToTarget vector because we have already gone to the trouble
        //of calculating its length: dist.
        Vector2D DesiredVelocity = ToTarget * speed / dist;

        return (DesiredVelocity - m_pVehicle->Velocity());
    }

    return Vector2D(0,0);
}
```

Now that you know what it does, have a look at the demo executable. Notice how when the vehicle is far away from the target the **arrive** behavior acts just the same as **seek**, and how the deceleration only comes into effect when the vehicle gets close to the target.

Pursuit

Pursuit behavior is useful when an agent is required to intercept a moving target. It could keep **seeking** to the current position of the target of course, but this wouldn't really help to create the illusion of intelligence. Imagine you're a child again and playing tag in the schoolyard. When you want to tag someone, you don't just run straight at their current position (which is effectively **seeking** toward them); you predict where they are going to be in the future and run toward that offset, making adjustments as you narrow the gap. See Figure 3.3. This is the sort of behavior we want our agents to demonstrate.

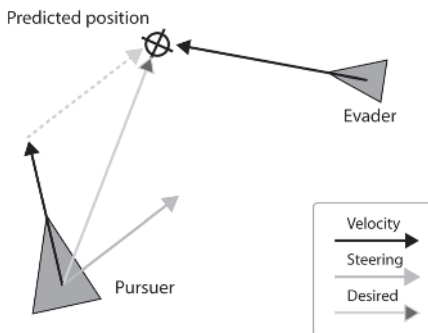


Figure 3.3. Calculating the vectors for the pursuit steering behavior. Once again, the dotted vector shows how the addition of the steering force to the current velocity produces the desired result.

The success of the pursuit function depends on how well the pursuer can predict the evader's trajectory. This can get very complicated, so a compromise must be made to obtain adequate performance without eating up too many clock cycles.

There is one situation the pursuer may face that enables an early out: If the evader is ahead and almost directly facing the agent, the agent should head directly for the evader's current position. This can be calculated quickly using dot products (see Chapter 1). In the example code, the evader's inverted heading must be within 20 degrees (approximately) of the agent's to be considered "facing."

One of the difficulties in creating a good predictor is deciding how far into the future the agent should predict. Clearly, the amount of look-ahead should be proportional to the separation between the pursuer and its evader, and inversely proportional to the pursuer's and evader's speeds. Once this time has been decided, an estimated future position can be calculated for the pursuer to **seek** to. Let's take a look at the code for this behavior:

```
Vector2D SteeringBehaviors::Pursuit(const Vehicle* evader)
{
    //if the evader is ahead and facing the agent then we can just seek
```

The Steering Behaviors

```
//for the evader's current position.
Vector2D ToEvader = evader->Pos() - m_pVehicle->Pos();

double RelativeHeading = m_pVehicle->Heading().Dot(evader->Heading());

if ((ToEvader.Dot(m_pVehicle->Heading()) > 0) &&
    (RelativeHeading < -0.95)) //acos(0.95)=18 degs
{
    return Seek(evader->Pos());
}

//Not considered ahead so we predict where the evader will be.

//the look-ahead time is proportional to the distance between the evader
//and the pursuer; and is inversely proportional to the sum of the
//agents' velocities
double LookAheadTime = ToEvader.Length() /
    (m_pVehicle->MaxSpeed() + evader->Speed());

//now seek to the predicted future position of the evader
return Seek(evader->Pos() + evader->Velocity() * LookAheadTime);
}
```

TIP Some locomotion models may also require that you factor in some time for turning the agent to face the offset. You can do this fairly simply by increasing the `LookAheadTime` by a value proportional to the dot product of the two headings and to the maximum turn rate of the vehicle. Something like:

```
LookAheadTime += TurnAroundTime(m_pVehicle, evader->Pos());
```

Where `TurnAroundTime` is the function:

```
double TurnaroundTime(const Vehicle* pAgent, Vector2D TargetPos)
{
    //determine the normalized vector to the target
    Vector2D toTarget = Vec2DNormalize(TargetPos - pAgent->Pos());

    double dot = pAgent->Heading().Dot(toTarget);

    //change this value to get the desired behavior. The higher the max turn
    //rate of the vehicle, the higher this value should be. If the vehicle is
    //heading in the opposite direction to its target position then a value
    //of 0.5 means that this function will return a time of 1 second for the
    //vehicle to turn around.
    const double coefficient = 0.5;

    //the dot product gives a value of 1 if the target is directly ahead and -1
    //if it is directly behind. Subtracting 1 and multiplying by the negative of
    //the coefficient gives a positive value proportional to the rotational
    //displacement of the vehicle and target.
    return (dot - 1.0) * -coefficient;
}
```

The **pursuit** demo shows a small vehicle being pursued by a larger one. The crosshair indicates the estimated future position of the evader. (The evader is utilizing a small amount of **wander** steering behavior to affect its motion. I'll be covering **wander** in just a moment.)

A pursuer's prey is set by passing the relevant method a pointer to the target in question. To set up a situation similar to the demo for this behavior you'd create two agents, one to pursue and the other to wander, just like this:

```
Vehicle* prey = new Vehicle(/* params omitted */);
prey->Steering()->WanderOn();

Vehicle* predator = new Vehicle(/* params omitted */);
predator->Steering()->PursuitOn(prex);
```

Got that? Okay, let's move on to **pursuit's** opposite: **evade**.

Evade

Evade is almost the same as **pursuit** except that this time the evader **flees** from the estimated future position.

```
Vector2D SteeringBehaviors::Evade(const Vehicle* pursuer)
{
    /* Not necessary to include the check for facing direction this time */

    Vector2D ToPursuer = pursuer->Pos() - m_pVehicle->Pos();

    //the look-ahead time is proportional to the distance between the pursuer
    //and the evader; and is inversely proportional to the sum of the
    //agents' velocities
    double LookAheadTime = ToPursuer.Length() /
        (m_pVehicle->MaxSpeed() + pursuer->Speed());

    //now flee away from predicted future position of the pursuer
    return Flee(pursuer->Pos() + pursuer->Velocity() * LookAheadTime);
}
```

Note that it is not necessary to include the check for facing direction this time.

Wander

You'll often find **wander** a useful ingredient when creating an agent's behavior. It's designed to produce a steering force that will give the impression of a random walk through the agent's environment.

A naive approach is to calculate a random steering force each time step, but this produces jittery behavior with no ability to achieve long persistent turns. (Actually, a rather nifty sort of random function, Perlin noise, can be used to produce smooth turning but this isn't very CPU friendly. It's still something for you to look into though if you get bored on a rainy day — Perlin noise has many applications.)

Reynolds' solution is to project a circle in front of the vehicle and steer toward a target that is constrained to move along the perimeter. Each time step, a small random displacement is added to this target, and over time it moves backward and forward along the circumference of the circle,

The Steering Behaviors

creating a lovely jitter-free alternating motion. This method can be used to produce a whole range of random motion, from very smooth undulating turns to wild *Strictly Ballroom* type whirls and pirouettes depending on the size of the circle, its distance from the vehicle, and the amount of random displacement each frame. As they say, a picture is worth a thousand words, so it's probably a good idea for you to examine Figure 3.4 to get a better understanding.

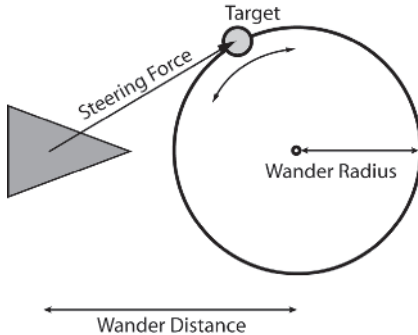


Figure 3.4

Let me take you through the code step by step. First there are three member variables wander makes use of:

```
double m_dWanderRadius;
```

This is the radius of the constraining circle.

```
double m_dWanderDistance;
```

This is the distance the wander circle is projected in front of the agent.

```
double m_dWanderJitter;
```

Finally, `m_dWanderJitter` is the maximum amount of random displacement that can be added to the target each second. Now for the method itself:

```
SVector2D SteeringBehaviors::Wander()
{
    //first, add a small random vector to the target's position (RandomClamped
    //returns a value between -1 and 1)
    m_vWanderTarget += SVector2D(RandomClamped() * m_dWanderJitter,
                                   RandomClamped() * m_dWanderJitter);
```

`m_vWanderTarget` is a point constrained to the parameter of a circle of radius `m_dWanderRadius`, centered on the vehicle (`m_vWanderTarget`'s initial position is set in the constructor of `SteeringBehaviors`). Each time step, a small random displacement is added to the wander target's position. See Figure 3.5A.

```
//reproject this new vector back onto a unit circle
m_vWanderTarget.Normalize();
```

```
//increase the length of the vector to the same as the radius
//of the wander circle
m_vWanderTarget *= m_dWanderRadius;
```

The next step is to reproject this new target back onto the wander circle. This is achieved by normalizing the vector and multiplying it by the radius of the wander circle. See Figure 3.5B.

```
//move the target into a position WanderDist in front of the agent
SVector2D targetLocal = m_vWanderTarget + SVector2D(m_dWanderDistance, 0);

//project the target into world space
SVector2D targetWorld = PointToWorldSpace(targetLocal,
                                          m_pVehicle->Heading(),
                                          m_pVehicle->Side(),
                                          m_pVehicle->Pos());

//and steer toward it
return targetWorld - m_pVehicle->Pos();
}
```

Finally, the new target is moved in front of the vehicle by an amount equal to `m_dWanderDistance` and projected into world space. The steering force is then calculated as the vector to this position. See Figure 3.5C.

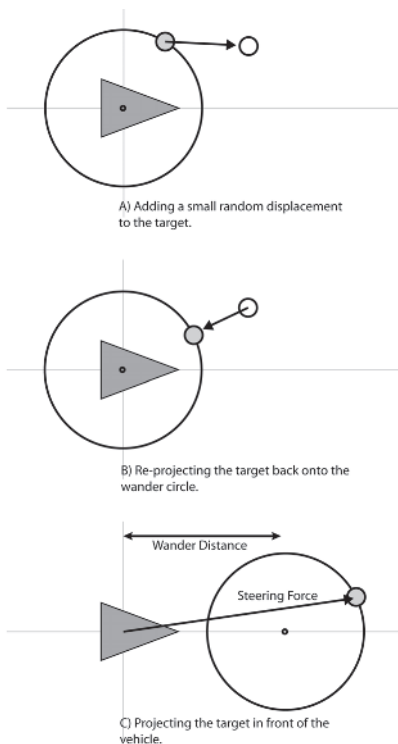


Figure 3.5. Steps toward calculating the wander behavior

The Steering Behaviors

If you have a computer at hand I recommend you check out the demo for this behavior. The green circle is the constraining “wander circle” and the red dot the target. The demo allows you to adjust the size of the wander circle, the amount of jitter, and the wander distance so you can observe the effect they have on the behavior. Notice the relationship between the wander distance and the variation in angle of the steering force returned from the method. When the wander circle is far away from the vehicle, the method produces small variations in angle, thereby limiting the vehicle to small turns. As the circle is moved closer to the vehicle, the amount it can turn becomes less and less restricted.

3D TIP If you require your agents to wander in three dimensions (like a spaceship patrolling its territory), all you have to do is constrain the wander target to a *sphere instead of a circle*.

Obstacle Avoidance

Obstacle avoidance is a behavior that steers a vehicle to avoid obstacles lying in its path. An obstacle is any object that can be approximated by a circle (or sphere, if you are working in 3D). This is achieved by steering the vehicle so as to keep a rectangular area — a detection box, extending forward from the vehicle — free of collisions. The detection box’s width is equal to the bounding radius of the vehicle, and its length is proportional to the vehicle’s current speed — the faster it goes, the longer the detection box.

I think before I describe this process any further it would be a good idea to show you a diagram. Figure 3.6 shows a vehicle, some obstacles, and the detection box used in the calculations.

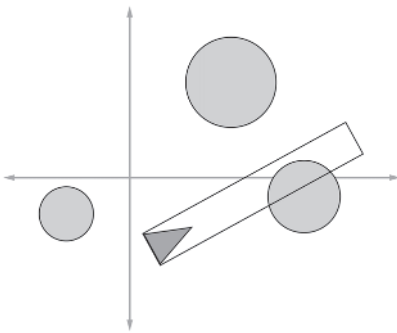


Figure 3.6. Setup for the obstacle avoidance steering behavior

Finding the Closest Intersection Point

The process of checking for intersections with obstacles is quite complicated, so let's take this step by step.

- A) The vehicle should only consider those obstacles within range of its detection box. Initially, the **obstacle avoidance** algorithm iterates through all the obstacles in the game world and tags those that are within this range for further consideration.
- B) The algorithm then transforms all the tagged obstacles into the vehicle's *local space* (for an explanation of local space, see Chapter 1). This makes life much easier as after transformation any objects with a negative local x -coordinate can be dismissed.
- C) The algorithm now has to check to see if any obstacles overlap the detection box. Local coordinates are useful here as all you need to do is expand the bounding radius of an obstacle by half the width of the detection box (the vehicle's bounding radius) and then check to see if its local y value is smaller than this value. If it isn't, then it won't intersect the detection box and can subsequently be discarded from further consideration.

Figure 3.7 should help clear up these first three steps for you. The letters on the obstacles in the diagram correspond to the descriptions.

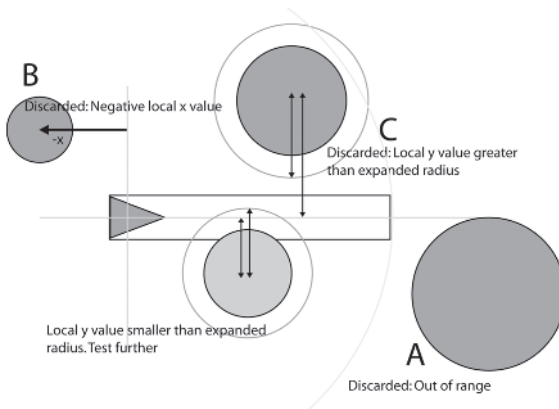


Figure 3.7. Steps A, B, and C

- D) At this point there are only those obstacles remaining that intersect the detection box. It's now necessary to find the intersection point closest to the vehicle. Once more, local space comes to the rescue. Step C expanded an object's bounding radius. Using this, a simple line/circle intersection test can be used to find where the expanded circle intersects the x -axis. There will be two intersection points, as shown in Figure 3.8. (We don't have to worry about the case where there is one

The Steering Behaviors

intersection tangent to the circle — the vehicle will appear to just glance off the obstacle.) Note that it is possible to have an obstacle in front of the vehicle, but it will have an intersection point to the rear of the vehicle. This is shown in the figure by obstacle A. The algorithm discards these cases and only considers intersection points laying on the positive x -axis.

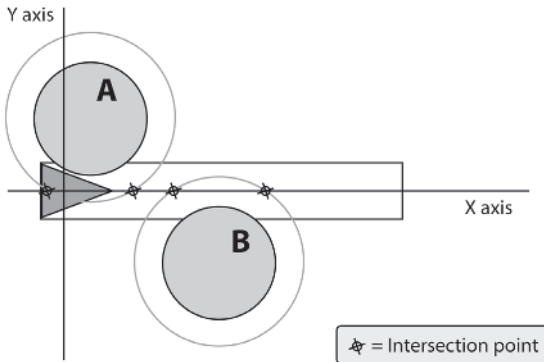


Figure 3.8. Intersection points

The algorithm tests all the remaining obstacles to find the one with the closest (positive) intersection point.

Before I show you how the steering force is calculated, let me list the part of the obstacle avoidance algorithm code that implements steps A to D.

```
Vector2D
SteeringBehaviors::ObstacleAvoidance(const std::vector<BaseGameEntity*>
&obstacles)
{
    //the detection box length is proportional to the agent's velocity
    m_dBoxLength = Prm.MinDetectionBoxLength +
        (m_pVehicle->Speed()/m_pVehicle->MaxSpeed()) *
        Prm.MinDetectionBoxLength;
```

All the parameters used by the project are read from an initialization file called Params.ini and stored in the singleton class ParamLoader. All the data in this class is public and is easily accessible through the #definition of Prm (#define Prm (*ParamLoader::Instance())). If further clarification is needed, see the ParamLoader.h file.

```
//tag all obstacles within range of the box for processing
m_pVehicle->World()->TagObstaclesWithinViewRange(m_pVehicle, m_dBoxLength);

//this will keep track of the closest intersecting obstacle (CIB)
BaseGameEntity* ClosestIntersectingObstacle = NULL;
```

```

//this will be used to track the distance to the CIB
double DistToClosestIP = MaxDouble;

//this will record the transformed local coordinates of the CIB
Vector2D LocalPosOfClosestObstacle;

std::vector<BaseGameEntity*>::const_iterator curOb = obstacles.begin();

while(curOb != obstacles.end())
{
    //if the obstacle has been tagged within range proceed
    if ((*curOb)->IsTagged())
    {
        //calculate this obstacle's position in local space
        Vector2D LocalPos = PointToLocalSpace((*curOb)->Pos(),
                                              m_pVehicle->Heading(),
                                              m_pVehicle->Side(),
                                              m_pVehicle->Pos());

        //if the local position has a negative x value then it must lay
        //behind the agent. (in which case it can be ignored)
        if (LocalPos.x >= 0)
        {
            //if the distance from the x axis to the object's position is less
            //than its radius + half the width of the detection box then there
            //is a potential intersection.
            double ExpandedRadius = (*curOb)->BRadius() + m_pVehicle->BRadius();

            if (fabs(LocalPos.y) < ExpandedRadius)
            {
                //now to do a line/circle intersection test. The center of the
                //circle is represented by (cX, cY). The intersection points are
                //given by the formula  $x = cX \pm \sqrt{r^2 - cY^2}$  for  $y=0$ .
                //We only need to look at the smallest positive value of x because
                //that will be the closest point of intersection.
                double cX = LocalPos.x;
                double cY = LocalPos.y;

                //we only need to calculate the sqrt part of the above equation once
                double SqrtPart = sqrt(ExpandedRadius*ExpandedRadius - cY*cY);

                double ip = A - SqrtPart;

                if (ip <= 0)
                {
                    ip = A + SqrtPart;
                }

                //test to see if this is the closest so far. If it is, keep a
                //record of the obstacle and its local coordinates
                if (ip < DistToClosestIP)
                {
                    DistToClosestIP = ip;

                    ClosestIntersectingObstacle = *curOb;
                }
            }
        }
    }
}

```

The Steering Behaviors

```

        LocalPosOfClosestObstacle = LocalPos;
    }
}
}
}

++curOb;
}

```

Calculating the Steering Force

Determining the steering force is easy. It's calculated in two parts: a *lateral force* and a *braking force*. See Figure 3.9.

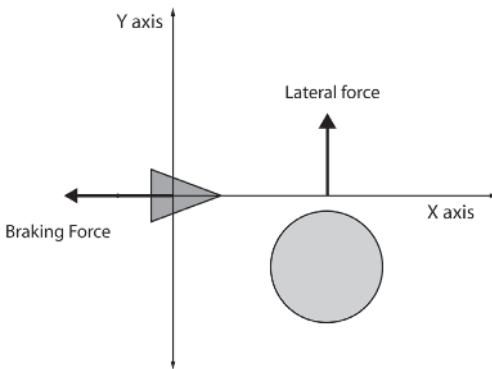


Figure 3.9. Calculating the steering force

There are a number of ways to calculate the lateral force but the one I prefer is to subtract the y value of the obstacle's local position from its radius. This results in a lateral steering force away from the obstacle that diminishes with the obstacle's distance from the x -axis. This force is scaled in proportion to the vehicle's distance from the obstacle (because the closer the vehicle is to an obstacle the quicker it should react).

The next component of the steering force is the braking force. This is a force acting backward, along the horizontal axis as shown in the figure, and is also scaled in proportion to the vehicle's distance from the obstacle.

The steering force is finally transformed into world space, resulting in the value returned from the method. The code is as follows

```

//if we have found an intersecting obstacle, calculate a steering
//force away from it
Vector2D SteeringForce;

if (ClosestIntersectingObstacle)
{
    //the closer the agent is to an object, the stronger the steering force

```

```

//should be
double multiplier = 1.0 + (m_dDBoxLength - LocalPosOfClosestObstacle.x) /
    m_dDBoxLength;

//calculate the lateral force
SteeringForce.y = (ClosestIntersectingObstacle->BRadius() -
    LocalPosOfClosestObstacle.y) * multiplier;

//apply a braking force proportional to the obstacle's distance from
//the vehicle.
const double BrakingWeight = 0.2;

SteeringForce.x = (ClosestIntersectingObstacle->BRadius() -
    LocalPosOfClosestObstacle.x) *
    BrakingWeight;
}

//finally, convert the steering vector from local to world space
return VectorToWorldSpace(SteeringForce,
    m_pVehicle->Heading(),
    m_pVehicle->Side());
}

```

3D TIP When implementing obstacle avoidance in three dimensions, use spheres to approximate the obstacles and a cylinder in place of the detection box. The math to check against a sphere is not that much different than that to check against a circle. Once the obstacles have been converted into local space, steps A and B are the same as you have already seen, and step C just involves checking against another axis.

Wall Avoidance

A wall is a line segment (in 3D, a polygon) with a normal pointing in the direction it is facing. **Wall avoidance** steers to avoid potential collisions with a wall. It does this by projecting three “feelers” out in front of the vehicle and testing to see if any of them intersect with any walls in the game world. See Figure 3.10. (The little “stub” halfway along the wall indicates the direction of the wall normal.) This is similar to how cats and rodents use their whiskers to navigate their environment in the dark.

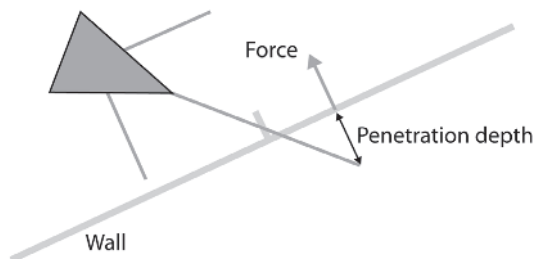


Figure 3.10. Wall avoidance

The Steering Behaviors

When the closest intersecting wall has been found (if there is one of course), a steering force is calculated. This is deduced by calculating how far the feeler tip has penetrated through the wall and then by creating a force of that magnitude in the direction of the wall normal.

```
Vector2D SteeringBehaviors::WallAvoidance(const std::vector<Wall2D>& walls)
{
    //the feelers are contained in a std::vector, m_Feelers
    CreateFeelers();

    double DistToThisIP    = 0.0;
    double DistToClosestIP = MaxDouble;

    //this will hold an index into the vector of walls
    int ClosestWall = -1;

    Vector2D SteeringForce,
              point,          //used for storing temporary info
              ClosestPoint;   //holds the closest intersection point

    //examine each feeler in turn
    for (int flr=0; flr<m_Feelers.size(); ++flr)
    {
        //run through each wall checking for any intersection points
        for (int w=0; w<walls.size(); ++w)
        {
            if (LineIntersection2D(m_pVehicle->Pos(),
                                   m_Feelers[flr],
                                   walls[w].From(),
                                   walls[w].To(),
                                   DistToThisIP,
                                   point))
            {
                //is this the closest found so far? If so keep a record
                if (DistToThisIP < DistToClosestIP)
                {
                    DistToClosestIP = DistToThisIP;

                    ClosestWall = w;

                    ClosestPoint = point;
                }
            }
        }
    } //next wall

    //if an intersection point has been detected, calculate a force
    //that will direct the agent away
    if (ClosestWall >=0)
    {
        //calculate by what distance the projected position of the agent
        //will overshoot the wall
        Vector2D OverShoot = m_Feelers[flr] - ClosestPoint;

        //create a force in the direction of the wall normal, with a
```

```

    //magnitude of the overshoot
    SteeringForce = walls[ClosestWall].Normal() * OverShoot.Length();
}

} //next feeler

return SteeringForce;
}

```

I have found the three feeler approach to give good results, but it's possible to achieve reasonable performance with just one feeler that continuously scans left and right in front of the vehicle. It all depends on how many processor cycles you have to play with and how accurate you require the behavior to be.

NOTE If you are the impatient sort and have already looked at the source code, you may have noticed that the final update function in the source is a little more complicated than the basic update function listed earlier. This is because many of the techniques I will be describing toward the end of this chapter involve adding to, or even changing, this function. All the steering behaviors listed over the next few pages, however, just use this basic skeleton.

Interpose

Interpose returns a steering force that moves a vehicle to the midpoint of the imaginary line connecting two other agents (or points in space, or of an agent and a point). A bodyguard taking a bullet for his employer or a soccer player intercepting a pass are examples of this type of behavior.

Like **pursuit**, the vehicle must estimate where the two agents are going to be located at a time T in the future. It can then steer toward that position. But how do we know what the best value of T is to use? The answer is, we don't, so we make a calculated guess instead.

The first step in calculating this force is to determine the midpoint of a line connecting the positions of the agents at the current time step. The distance from this point is computed and the value divided by the vehicle's maximum speed to give the time required to travel the distance. This is our T value. See Figure 3.11, top.

Using T , the agents' positions are extrapolated into the future. The midpoint of these predicted positions is determined and finally the vehicle uses the **arrive** behavior to steer toward that point. See Figure 3.11, bottom.

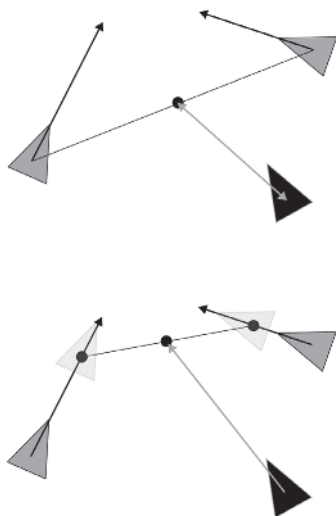


Figure 3.11. Predicting the interpose point

The Steering Behaviors

Here's the listing:

```
Vector2D SteeringBehaviors::Interpose(const Vehicle* AgentA,
                                     const Vehicle* AgentB)
{
    //first we need to figure out where the two agents are going to be at
    //time T in the future. This is approximated by determining the time
    //taken to reach the midway point at the current time at max speed.
    Vector2D MidPoint = (AgentA->Pos() + AgentB->Pos()) / 2.0;

    double TimeToReachMidPoint = Vec2DDistance(m_pVehicle->Pos(), MidPoint) /
                                   m_pVehicle->MaxSpeed();

    //now we have T, we assume that agent A and agent B will continue on a
    //straight trajectory and extrapolate to get their future positions
    Vector2D APos = AgentA->Pos() + AgentA->Velocity() * TimeToReachMidPoint;
    Vector2D BPos = AgentB->Pos() + AgentB->Velocity() * TimeToReachMidPoint;

    //calculate the midpoint of these predicted positions
    MidPoint = (APos + BPos) / 2.0;

    //then steer to arrive at it
    return Arrive(MidPoint, fast);
}
```

Note that **arrive** is called with fast deceleration, allowing the vehicle to reach the target position as quickly as possible.

The demo for this behavior shows a red vehicle attempting to interpose itself between two blue wandering vehicles.

Hide

Hide attempts to position a vehicle so that an obstacle is always between itself and the agent — the hunter — it's trying to hide from. You can use this behavior not only for situations where you require an NPC to hide from the player — like find cover when fired at — but also in situations where you would like an NPC to sneak up on a player. For example, you can create an NPC capable of stalking a player through a gloomy forest, darting from tree to tree. Creepy!

The method I prefer to effect this behavior is as follows:

Step One. For each of the obstacles, a hiding spot is determined. See Figure 3.12.

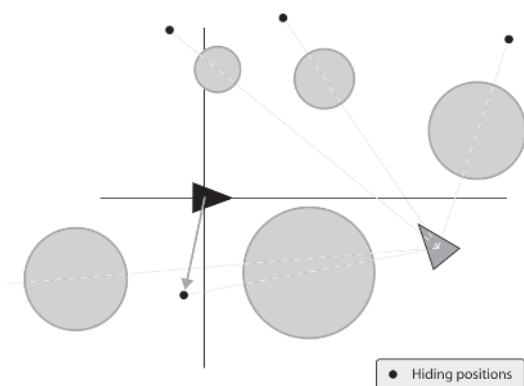


Figure 3.12. Potential hiding spots

These are calculated by the method `GetHidingPosition`, which looks like this:

```
SVector2D SteeringBehaviors::GetHidingPosition(const SVector2D& posOb,
                                                const double   radiusOb,
                                                const SVector2D& posTarget)
{
    //calculate how far away the agent is to be from the chosen obstacle's
    //bounding radius
    const double DistanceFromBoundary = 30.0;

    double DistAway = radiusOb + DistanceFromBoundary;

    //calculate the heading toward the object from the target
    SVector2D ToOb = Vec2DNormalize(posOb - posTarget);

    //scale it to size and add to the obstacle's position to get
    //the hiding spot.
    return (ToOb * DistAway) + posOb;
}
```

Given the position of a target and the position and radius of an obstacle, this method calculates a position `DistanceFromBoundary` away from the object's bounding radius and directly opposite the target. It does this by scaling the normalized "to obstacle" vector by the required distance away from the center of the obstacle and then adding the result to the obstacle's position. The black dots in Figure 3.12 show the hiding spots returned by this method for that example.

Step Two. The distance to each of these spots is determined. The vehicle then uses the **arrive** behavior to steer toward the closest. If no appropriate obstacles can be found, the vehicle **evades** the target.

The Steering Behaviors

Here's how it's done in code:

```
SVector2D SteeringBehaviors::Hide(const Vehicle* target,
                                  vector<BaseGameEntity*>& obstacles)
{
    double DistToClosest = MaxDouble
    SVector2D BestHidingSpot;

    std::vector<BaseGameEntity*>::iterator curOb = obstacles.begin();
    while(curOb != obstacles.end())
    {
        //calculate the position of the hiding spot for this obstacle
        SVector2D HidingSpot = GetHidingPosition((*curOb)->Pos(),
                                                (*curOb)->BRadius(),
                                                target->Pos());

        //work in distance-squared space to find the closest hiding
        //spot to the agent
        double dist = Vec2DDistanceSq(HidingSpot, m_pVehicle->Pos());

        if (dist < DistToClosest)
        {
            DistToClosest = dist;

            BestHidingSpot = HidingSpot;
        }

        ++curOb;
    } //end while

    //if no suitable obstacles found then evade the target
    if (DistToClosest == MaxDouble)
    {
        return Evade(target);
    }

    //else use Arrive on the hiding spot
    return Arrive(BestHidingSpot, fast);
}
```

The demo executable shows two vehicles hiding from a slower, wandering vehicle.

There are a few modifications you can make to this algorithm:

1. You can allow the vehicle to hide only if the target is within its field of view. This tends to produce unsatisfactory performance though, because the vehicle starts to act like a child hiding from monsters beneath the bed sheets. I'm sure you remember the feeling — the “if you can't see it, then it can't see you” effect. It might work when you're a kid, but this sort of behavior just makes the vehicle look dumb. This can be countered slightly though by adding in a time effect so that the vehicle will hide if the target is visible *or* if it has

- seen the target within the last n seconds. This gives it a sort of memory and produces reasonable-looking behavior.
2. The same as above, but this time the vehicle only tries to hide if the vehicle can see the target *and* the target can see the vehicle.
 3. It might be desirable to produce a force that steers a vehicle so that it always favors hiding positions that are to the side or rear of the pursuer. This can be achieved easily using your friend the dot product to bias the distances returned from `GetHidingPosition`.
 4. At the beginning of any of the methods a check can be made to test if the target is within a “threat distance” before proceeding with any further calculations. If the target is not a threat, then the method can return immediately with a zero vector.

Path Following

Path following creates a steering force that moves a vehicle along a series of waypoints forming a path. Sometimes paths have a start and end point, and other times they loop back around on themselves forming a never-ending, closed path. See Figure 3.13.

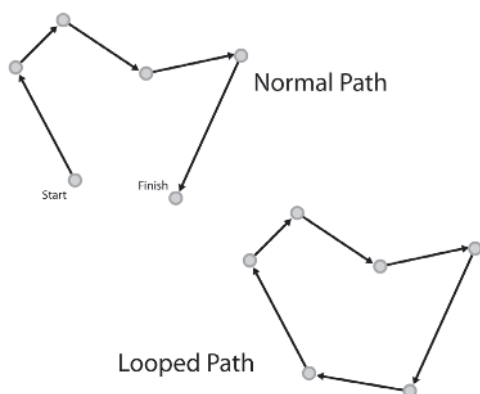


Figure 3.13. Different types of paths

You’ll find countless uses for using paths in your game. You can use them to create agents that patrol important areas of a map, to enable units to traverse difficult terrain, or to help racing cars navigate around a racetrack. They are useful in most situations where an agent must visit a series of checkpoints.

The paths the vehicles described in this chapter follow are described by a `std::list` of `Vector2Ds`. In addition, the vehicle also needs to know what the current waypoint is and whether it is a closed path or not to enable it to take the appropriate action when it reaches the final waypoint. If it is a closed path, it should head back to the first waypoint in the list and start all

The Steering Behaviors

over again. If it's an open path, the vehicle should just decelerate to a stop (**arrive**) over the final waypoint.

Path is a class that looks after all these details. I'm not going to list it here but you may like to examine it in your IDE. You can find it in the file Path.h.

The simplest way of following a path is to set the current waypoint to the first in the list, steer toward that using **seek** until the vehicle comes within a target distance of it, then grab the next waypoint and **seek** to that, and so on, until the current waypoint is the last waypoint in the list. When this happens the vehicle should either **arrive** at the current waypoint, or, if the path is a closed loop, the current waypoint should be set to the first in the list again, and the vehicle just keeps on **seeking**. Here's the code for **path following**:

```
SVector2D SteeringBehaviors::FollowPath()
{
    //move to next target if close enough to current target (working in
    //distance squared space)
    if(Vec2DDistanceSq(m_pPath->CurrentWaypoint(), m_pVehicle->Pos()) <
        m_WaypointSeekDistSq)
    {
        m_pPath->SetNextWaypoint();
    }

    if (!m_pPath->Finished())
    {
        return Seek(m_pPath->CurrentWaypoint());
    }

    else
    {
        return Arrive(m_pPath->CurrentWaypoint(), normal);
    }
}
```

You have to be very careful when implementing **path following**. The behavior is very sensitive to the max steering force/max speed ratio and also the variable `m_WaypointSeekDistSq`. The demo executable for this behavior allows you to alter these values to see what effect they have. As you will discover, it's easy to create behavior that is sloppy. How tight you need the **path following** to be depends entirely on your game environment. If you have a game with lots of gloomy tight corridors, then you're (probably) going to need stricter **path following** than a game set in the Sahara.

Offset Pursuit

Offset pursuit calculates the steering force required to keep a vehicle positioned at a specified offset from a target vehicle. This is particularly useful for creating formations. When you watch an air display, such as the British Red Arrows, many of the spectacular maneuvers require that the aircraft

remain in the same relative positions to the lead aircraft. See Figure 3.14. This is the sort of behavior we want to emulate.

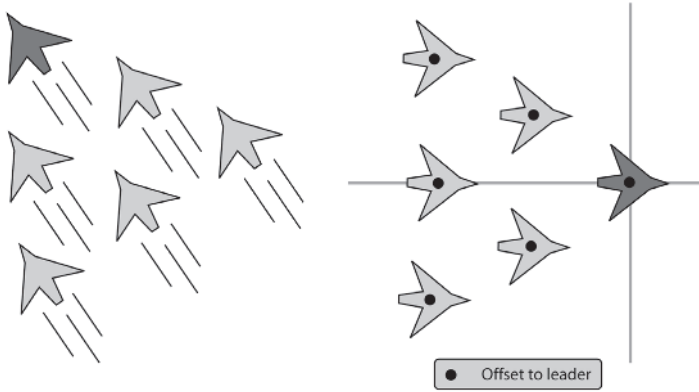


Figure 3.14. Offset pursuit. The leader is shown in dark gray.

The offset is always defined in “leader” space, so the first thing to do when calculating this steering force is to determine the offset’s position in world space. After that the function proceeds similar to pursuit: A future position for the offset is predicted and the vehicle arrives at that position.

```
SVector2D SteeringBehaviors::OffsetPursuit(const Vehicle* leader,
                                          const SVector2D offset)
{
    //calculate the offset's position in world space
    SVector2D WorldOffsetPos = PointToWorldSpace(offset,
                                                leader->Heading(),
                                                leader->Side(),
                                                leader->Pos());

    SVector2D ToOffset = WorldOffsetPos - m_pVehicle->Pos();

    //the look-ahead time is proportional to the distance between the leader
    //and the pursuer; and is inversely proportional to the sum of both
    //agents' velocities
    double LookAheadTime = ToOffset.Length() /
        (m_pVehicle->MaxSpeed() + leader->Speed());

    //now arrive at the predicted future position of the offset
    return Arrive(WorldOffsetPos + leader->Velocity() * LookAheadTime, fast);
}
```

Arrive is used instead of **seek** as it gives far smoother motion and isn’t so reliant on the max speed and max force settings of the vehicles. **Seek** can give some rather bizarre results at times — orderly formations can turn into what looks like a swarm of bees attacking the formation leader!

Group Behaviors

Offset pursuit is useful for all kinds of situations. Here are a few:

- Marking an opponent in a sports simulation
- Docking with a spaceship
- Shadowing an aircraft
- Implementing battle formations

The demo executable for **offset pursuit** shows three smaller vehicles attempting to remain at offsets to the larger lead vehicle. The lead vehicle is using **arrive** to follow the crosshair (click the left mouse button to position the crosshair).

Group Behaviors

Group behaviors are steering behaviors that take into consideration some or all of the other vehicles in the game world. The flocking behavior I described at the beginning of this chapter is a good example of a group behavior. In fact, flocking is a combination of three group behaviors — **cohesion**, **separation**, and **alignment** — all working together. We'll take a look at these specific behaviors in detail shortly, but first let me show you how a group is defined.

To determine the steering force for a group behavior, a vehicle will consider all other vehicles within a circular area of predefined size — known as the *neighborhood radius* — centered on the vehicle. Figure 3.15 should help clarify. The white vehicle is the steering agent and the gray circle shows the extent of its neighborhood. Consequently, all the vehicles shown in black are considered to be its neighbors and the vehicles shown in gray are not.

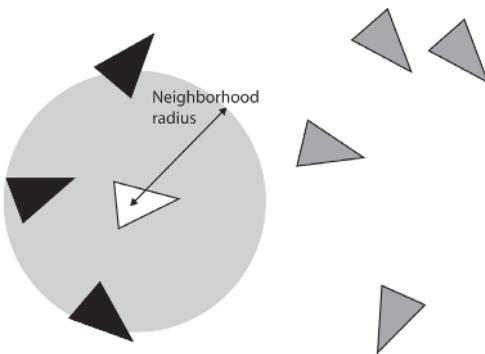


Figure 3.15. The neighborhood radius

Before a steering force can be calculated, a vehicle's neighbors must be determined and either stored in a container or tagged ready for processing. In the demo code for this chapter, the neighboring vehicles are tagged

using the `BaseGameEntity::Tag` method. This is done by the `TagNeighbors` function template. Here's the code:

```
template <class T, class conT>
void TagNeighbors(const T* entity, conT& ContainerOfEntities, double radius)
{
    //iterate through all entities checking for range
    for (typename conT::iterator curEntity = ContainerOfEntities.begin();
         curEntity != ContainerOfEntities.end();
         ++curEntity)
    {
        //first clear any current tag
        (*curEntity)->UnTag();

        Vector2D to = (*curEntity)->Pos() - entity->Pos();

        //the bounding radius of the other is taken into account by adding it
        //to the range
        double range = radius + (*curEntity)->BRadius();

        //if entity within range, tag for further consideration. (working in
        //distance-squared space to avoid sqrts)
        if ( ((*curEntity) != entity) && (to.LengthSq() < range*range))
        {
            (*curEntity)->Tag();
        }
    }
    //next entity
}
```

Most of the group behaviors utilize a similar neighborhood radius, so we can save a little time by calling this method only once prior to a call to any of the group behaviors.

```
if (On(separation) || On(alignment) || On(cohesion))
{
    TagNeighbors(m_pVehicle, m_pVehicle->World()->Agents(), ViewDistance);
}
```

TIP You can pep up the realism slightly for group behaviors by adding a field-of-view constraint to your agent. For example you can restrict the vehicles included in the neighboring region by only tagging those that are within, say, 270 degrees of the heading of the steering agent. You can implement this easily by testing against the dot product of the steering agent's heading and the vector to the potential neighbor.

It's even possible to adjust an agent's FOV dynamically and make it into a feature of the AI. For example, in a war game a soldier's FOV may be detrimentally affected by its fatigue, thereby affecting its ability to perceive its environment. I don't think this idea has been used in a commercial game but it's certainly food for thought.

Now that you know how a group is defined let's take a look at some of the behaviors that operate on them.

Separation

Separation creates a force that steers a vehicle away from those in its neighborhood region. When applied to a number of vehicles, they will spread out, trying to maximize their distance from every other vehicle. See Figure 3.16, top.

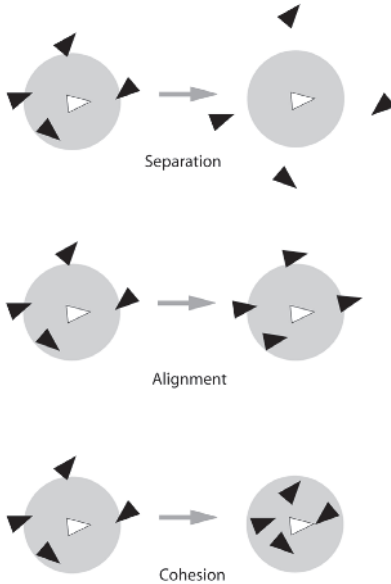


Figure 3.16. The separation, alignment, and cohesion group behaviors

This is an easy behavior to implement. Prior to calling **separation**, all the agents situated within a vehicle's neighborhood are tagged. **Separation** then iterates through the tagged vehicles, examining each one. The vector to each vehicle under consideration is normalized, divided by the distance to the neighbor, and added to the steering force.

```
Vector2D SteeringBehaviors::Separation(const std::vector<Vehicle*>& neighbors)
{
    Vector2D SteeringForce;

    for (int a=0; a<neighbors.size(); ++a)
    {
        //make sure this agent isn't included in the calculations and that
        //the agent being examined is close enough.
        if((neighbors[a] != m_pVehicle) && neighbors[a]->IsTagged())
        {
            Vector2D ToAgent = m_pVehicle->Pos() - neighbors[a]->Pos();

            //scale the force inversely proportional to the agent's distance
            //from its neighbor.
            SteeringForce += Vec2DNormalize(ToAgent)/ToAgent.Length();
        }
    }
}
```

```

    }
}

return SteeringForce;
}

```

Alignment

Alignment attempts to keep a vehicle's heading aligned with its neighbors. See Figure 3.16, middle. The force is calculated by first iterating through all the neighbors and averaging their heading vectors. This value is the desired heading, so we just subtract the vehicle's heading to get the steering force.

```

Vector2D SteeringBehaviors::Alignment(const std::vector<Vehicle*>& neighbors)
{
    //used to record the average heading of the neighbors
    Vector2D AverageHeading;

    //used to count the number of vehicles in the neighborhood
    int    NeighborCount = 0

    //iterate through all the tagged vehicles and sum their heading vectors
    for (int a=0; a<neighbors.size(); ++a)
    {
        //make sure *this* agent isn't included in the calculations and that
        //the agent being examined is close enough
        if((neighbors[a] != m_pVehicle) && neighbors[a]->IsTagged)
        {
            AverageHeading += neighbors[a]->Heading();

            ++NeighborCount;
        }
    }

    //if the neighborhood contained one or more vehicles, average their
    //heading vectors.
    if (NeighborCount > 0)
    {
        AverageHeading /= (double)NeighborCount;

        AverageHeading -= m_pVehicle->Heading();
    }

    return AverageHeading;
}

```

Cars moving along roads demonstrate **alignment** type behavior. They also demonstrate **separation** as they try to keep a minimum distance from each other.

Cohesion

Cohesion produces a steering force that moves a vehicle toward the center of mass of its neighbors. See Figure 3.16, bottom. A sheep running after its flock is demonstrating cohesive behavior. Use this force to keep a group of vehicles together.

This method proceeds similarly to the last, except this time we calculate the average of the position vectors of the neighbors. This gives us the center of mass of the neighbors — the place the vehicle wants to get to — so it **seeks** to that position.

```
Vector2D SteeringBehaviors::Cohesion(const std::vector<Vehicle*>& neighbors)
{
    //first find the center of mass of all the agents
    Vector2D CenterOfMass, SteeringForce;

    int NeighborCount = 0;

    //iterate through the neighbors and sum up all the position vectors
    for (int a=0; a<neighbors.size(); ++a)
    {
        //make sure *this* agent isn't included in the calculations and that
        //the agent being examined is a neighbor
        if((neighbors[a] != m_pVehicle) && neighbors[a]->IsTagged())
        {
            CenterOfMass += neighbors[a]->Pos();

            ++NeighborCount;
        }
    }

    if (NeighborCount > 0)
    {
        //the center of mass is the average of the sum of positions
        CenterOfMass /= (double)NeighborCount;

        //now seek toward that position
        SteeringForce = Seek(CenterOfMass);
    }

    return SteeringForce;
}
```

You might be a little disappointed that I haven't included demos for separation, cohesion, and alignment. Well, there's a reason for that: Like Itchy and Scratchy, they are not particularly interesting on their own; they are much better appreciated when they are *combined*, which takes us nicely into **flocking**.

Flocking

Flocking is the behavior I mentioned at the beginning of this chapter — the one I saw on the BBC documentary. It's a beautiful demonstration of what has become known as *emergent behavior*. Emergent behavior is behavior that looks complex and/or purposeful to the observer but is actually derived spontaneously from fairly simple rules. The lower-level entities following the rules have no idea of the bigger picture; they are only aware of themselves and maybe a few of their neighbors.

One good example of emergence is an experiment undertaken by Chris Melhuish and Owen Holland at the University of the West of England. Melhuish and Holland are interested in *stigmergy*, the field of science partly occupied with emergent behavior in social insects like ants, termites, and bees. They became interested in the way ants gather their dead, eggs, and other material into piles, and specifically the ant *Leptothorax*, because it lives among the cracks in rocks and operates, for all intents and purposes, in 2D... just like a wheeled robot. When observing *Leptothorax* in the laboratory, bustling about in their simulated crack — two sheets of glass — they noticed the ants had a tendency to push small granules of rock material together into clusters and wondered if they could design robots capable of doing the same.

After a little sweat and toil they managed to create robots operating on very simple rules, capable of gathering randomly scattered Frisbees into clusters. The robots had no knowledge of each other and didn't know what a cluster, or even a Frisbee, was. They couldn't even *see* the Frisbees. They could only push Frisbees using a U-shaped arm situated in front of them.

So how does the clustering behavior happen? Well, when the robots are switched on, they wander about until they bump into a Frisbee. A single Frisbee doesn't change a robot's behavior. However, when a Frisbee-pushing robot bumps into a second Frisbee, it immediately leaves the two Frisbees where they are, backs up a little, rotates by a random amount, and then wanders off again. Using just these simple rules and a little time, a few robots will push all the Frisbees into a few large clusters. Just like the ants.

Anyway, let me abandon all this talk of Frisbees and get back to the flocking. Flocking, as originally described by Reynolds, is a combination of the three previously described group behaviors: **separation**, **alignment**, and **cohesion**. This works okay but, because of the limited view distance of a vehicle, it's possible for an agent to become isolated from its flock. If this happens, it will just sit still and do nothing. To prevent this from happening, I prefer to add in the **wander** behavior too. This way, all the agents keep moving all the time.

Tweaking the magnitudes of each of the contributing behaviors will give you different effects such as shoals of fish, loose swirling flocks of birds,

Combining Steering Behaviors

or bustling close-knit herds of sheep. I've even managed to produce dense flocks of hundreds of tiny particles that are reminiscent of jellyfish. As this behavior is better seen than described, I recommend you open up the demo executable and play around for a while. Beware though — flocking is addictive! (Maybe that's why some animals like to do it so much...) You can adjust the influence of each behavior with the "A/Z," "S/X," and "D/C" keys. In addition you can view the neighbors of one of the agents by pressing the "G" key.



INTERESTING FACT Steering behaviors are often used to create special effects for films. The first film to use the flocking behavior was *Batman Returns*, where you can see flocks of bats and herds of penguins. The most recent films to use steering behaviors are *The Lord of the Rings* trilogy, directed by Peter Jackson. The movement of the orc armies in those films is created using steering behaviors via a piece of software called *Massive*.

Now that you've seen the benefits, let's take a look at exactly how steering behaviors can be combined.

Combining Steering Behaviors


Often you will be using a *combination* of steering behaviors to get the behavior you desire. Very rarely will you only use one behavior in isolation. For example, you might like to implement an FPS bot that will run from A to B (**path following**) while avoiding any other bots (**separation**) and walls (**wall avoidance**) that may try to impede its progress (see Chapter 7, "Raven: An Overview"). Or you might want the sheep you've implemented as a food resource in your RTS game to flock together (**flocking**) while simultaneously wandering around the environment (**wander**), avoiding trees (**obstacle avoidance**), and scattering (**evade**) whenever a human or dog comes near.

All the steering behaviors described in this chapter are methods of one class: `SteeringBehaviors`. A `Vehicle` owns an instance of this class and activates/deactivates the various behaviors by switching them on and off using accessor methods. For example, to set up one of the sheep for the situation described in the previous paragraph, you may do something like this (assuming a dog-like agent has already been created):

```
Vehicle* Sheep = new Vehicle();

Sheep->Steering()->SeparationOn();
Sheep->Steering()->AlignmentOn();
Sheep->Steering()->CohesionOn();
Sheep->Steering()->ObstacleAvoidanceOn();
Sheep->Steering()->WanderOn();
Sheep->Steering()->EvadeOn(Dog);
```

And from now on the sheep will look after itself! (You may have to shear it in the summer though.)

 **NOTE** Because of the number of demos I've created for this chapter, the `SteeringBehaviors` class is enormous and contains much more code than would ever get used in a single project. Very rarely will you use more than a handful of behaviors for each game you design. Therefore, whenever I use steering behaviors in later chapters, I will use a cut-down version of the `SteeringBehaviors` class, custom made for the task at hand. I suggest you do the same. (Another approach is to define a separate class for each behavior and add them to a `std::container` as you need them.)

Inside the `Vehicle::Update` method you will see this line:

```
SVector2D SteeringForce = m_pSteering->Calculate();
```

This call determines the resultant force from all the active behaviors. This is not simply a sum of all the steering forces though. Don't forget that the vehicle is constrained by a maximum steering force, so this sum must be truncated in some way to make sure its magnitude never exceeds the limit. There are a number of ways you can do this. It's impossible to say if one method is better than another because it depends on what behaviors you need to work with and what CPU resources you have to spare. They all have their pros and cons. I strongly recommend you experiment for yourself.

Weighted Truncated Sum

The simplest approach is to multiply each steering behavior with a weight, sum them all together, and then truncate the result to the maximum allowable steering force. Like this:

```
SVector2D Calculate()
{
    SVector2D SteeringForce;

    SteeringForce += Wander()           * dWanderAmount;
    SteeringForce += ObstacleAvoidance() * dObstacleAvoidanceAmount;
    SteeringForce += Separation()       * dSeparationAmount;

    return SteeringForce.Truncate(MAX_STEERING_FORCE);
}
```

This can work fine, but the trade-off is that it comes with a few problems. The first problem is that because every active behavior is calculated every time step, this is a very costly method to process. Additionally, the behavior weights can be very difficult to tweak. (Did I say difficult? Sorry, I mean *very* difficult! ☺) The biggest problem, however, happens with conflicting forces — a common scenario is where a vehicle is backed up against a wall by several other vehicles. In this example, the separating forces from the neighboring vehicles can be greater than the repulsive force from the wall and the vehicle can end up being pushed through the wall boundary. This is almost certainly not going to be favorable. Sure you can make the weights for the wall avoidance huge, but then your vehicle may behave strangely next time it finds itself alone and next to a wall. Like I

mentioned, tweaking the parameters for a weighted sum can be quite a juggling act!

Weighted Truncated Running Sum with Prioritization

What a mouthful! This is the method used to determine the steering forces for all the examples used in this book, chosen mainly because it gives a good compromise between speed and accuracy. This method involves calculating a *prioritized* weighted running total that is truncated after the addition of each force to make sure the magnitude of the steering force does not exceed the maximum available.

The steering behaviors are prioritized since some behaviors can be considered much more important than others. Let's say a vehicle is using the behaviors **separation**, **alignment**, **cohesion**, **wall avoidance**, and **obstacle avoidance**. The **wall avoidance** and **obstacle avoidance** behaviors should be given priority over the others as the vehicle should try not to intersect a wall or an obstacle — it's more important for a vehicle to avoid a wall than it is for it to align itself with another vehicle. If what it takes to avoid a wall is higgledy-piggledy alignment, then that's probably going to be okay and certainly preferable to colliding with a wall. It's also more important for vehicles to maintain some separation from each other than it is for them to align. But it's probably less important for vehicles to maintain separation than avoid walls. See where I'm going with this? Each behavior is prioritized and processed in order. The behaviors with the highest priority are processed first, the ones with the lowest, last.

In addition to the prioritization, this method iterates through every active behavior, summing up the forces (with weighting) as it goes. Immediately after the calculation of each new behavior, the resultant force, together with the running total, is dispatched to a method called `AccumulateForce`. This function first determines how much of the maximum available steering force is remaining, and then one of the following happens:

- If there is a surplus remaining, the new force is added to the running total.
- If there is no surplus remaining, the method returns `false`. When this happens, `Calculate` returns the current value of `m_vSteeringForce` immediately and without considering any further active behaviors.
- If there is still some steering force available, but the magnitude remaining is less than the magnitude of the new force, the new force is truncated to the remaining magnitude before it is added.

Here is a snippet of code from the `SteeringBehaviors::Calculate` method to help you better understand what I'm talking about.

```
SVector2D SteeringBehaviors::Calculate()
{
    //reset the force.
```

```

m_vSteeringForce.Zero();

SVector2D force;
if (On(wall_avoidance))
{
    force = WallAvoidance(m_pVehicle->World()->Walls()) *
        m_dMultWallAvoidance;

    if (!AccumulateForce(m_vSteeringForce, force)) return m_vSteeringForce;
}

if (On(obstacle_avoidance))
{
    force = ObstacleAvoidance(m_pVehicle->World()->Obstacles()) *
        m_dMultObstacleAvoidance;

    if (!AccumulateForce(m_vSteeringForce, force)) return m_vSteeringForce;
}

if (On(separation))
{
    force = Separation(m_pVehicle->World()->Agents()) *
        m_dMultSeparation;

    if (!AccumulateForce(m_vSteeringForce, force)) return m_vSteeringForce;
}

/* EXTRANEIOUS STEERING FORCES OMITTED */
return m_vSteeringForce;
}

```

This doesn't show all the steering forces, just a few so you can get the general idea. To see the list of all behaviors and the order of their prioritization, check out the `SteeringBehaviors::Calculate` method in your IDE. The `AccumulateForce` method may also be better explained in code. Take your time looking over this method and make sure you understand what it's doing.

```

bool SteeringBehaviors::AccumulateForce(Vector2D &RunningTot,
                                       Vector2D ForceToAdd)
{
    //calculate how much steering force the vehicle has used so far
    double MagnitudeSoFar = RunningTot.Length();

    //calculate how much steering force remains to be used by this vehicle
    double MagnitudeRemaining = m_pVehicle->MaxForce() - MagnitudeSoFar;

    //return false if there is no more force left to use
    if (MagnitudeRemaining <= 0.0) return false;

    //calculate the magnitude of the force we want to add
    double MagnitudeToAdd = ForceToAdd.Length();

    //if the magnitude of the sum of ForceToAdd and the running total

```

Combining Steering Behaviors

```

//does not exceed the maximum force available to this vehicle, just
//add together. Otherwise add as much of the ForceToAdd vector as
//possible without going over the max.
if (MagnitudeToAdd < MagnitudeRemaining)
{
    RunningTot += ForceToAdd;
}

else
{
    //add it to the steering force
    RunningTot += (Vec2DNormalize(ForceToAdd) * MagnitudeRemaining);
}

return true;
}

```

Prioritized Dithering

In his paper, Reynolds suggests a method of force combination he calls *prioritized dithering*. When used, this method checks to see if the first priority behavior is going to be evaluated this simulation step, dependent on a pre-set probability. If it is and the result is non-zero, the method returns the calculated force and no other active behaviors are considered. If the result is zero or if that behavior has been skipped over due to its probability of being evaluated, the next priority behavior is considered and so on, for all the active behaviors. This is a little snippet of code to help you understand the concept:

```

SVector2D SteeringBehaviors::CalculateDithered()
{
    //reset the steering force
    m_vSteeringForce.Zero();

    //the behavior probabilities
    const double prWallAvoidance      = 0.9;
    const double prObstacleAvoidance = 0.9;
    const double prSeparation         = 0.8;
    const double prAlignment          = 0.5;
    const double prCohesion           = 0.5;
    const double prWander              = 0.8;

    if (On(wall_avoidance) && RandFloat() > prWallAvoidance)
    {
        m_vSteeringForce = WallAvoidance(m_pVehicle->World()->Walls()) *
            m_dWeightWallAvoidance / prWallAvoidance;

        if (!m_vSteeringForce.IsZero())
        {
            m_vSteeringForce.Truncate(m_pVehicle->MaxForce());

            return m_vSteeringForce;
        }
    }
}

```

```

    }
}

if (On(obstacle_avoidance) && RandFloat() > prObstacleAvoidance)
{
    m_vSteeringForce += ObstacleAvoidance(m_pVehicle->World()->Obstacles()) *
        m_dWeightObstacleAvoidance / prObstacleAvoidance;

    if (!m_vSteeringForce.IsZero())
    {
        m_vSteeringForce.Truncate(m_pVehicle->MaxForce());

        return m_vSteeringForce;
    }
}

if (On(separation) && RandFloat() > prSeparation)
{
    m_vSteeringForce += Separation(m_pVehicle->World()->Agents()) *
        m_dWeightSeparation / prSeparation;

    if (!m_vSteeringForce.IsZero())
    {
        m_vSteeringForce.Truncate(m_pVehicle->MaxForce());

        return m_vSteeringForce;
    }
}

/* ETC ETC */

```

This method requires far less CPU time than the others, but at the cost of accuracy. Additionally, you will have to tweak the probabilities a fair bit before you get the behavior just as you want it. Nevertheless, if you are low on resources and it's not imperative your agent's movements are precise, this method is certainly worth experimenting with. You can see the effect of each of the three summing methods I've described by running the demo Big Shoal/Big Shoal.exe. This demonstration shows a shoal of 300 small vehicles (think fish) being wary of a single larger wandering vehicle (think shark). You can switch between the various summing methods to observe how they affect the frame rate and accuracy of the behaviors. You can also add walls or obstacles to the environment to see how the agents handle those using the different summing methods.

Ensuring Zero Overlap

Often when combining behaviors, the vehicles will occasionally overlap one another. The **separation** steering force alone is not enough to prevent this from happening. Most of the time this is okay — a little overlap will go unnoticed by the player — but sometimes it's necessary to ensure that whatever happens, vehicles cannot pass through one another's bounding

Ensuring Zero Overlap

radii. This can be prevented with the use of a *non-penetration constraint*. This is a function that tests for overlap. If there is any, the vehicles are moved apart in a direction away from the point of contact (and without regard to their mass, velocity, or any other physical constraints). See Figure 3.17.

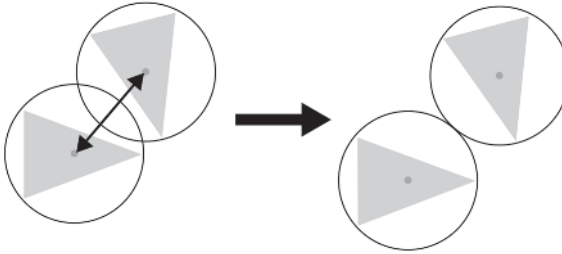


Figure 3.17. The non-penetration constraint in action

The constraint is implemented as a function template and can be used for any objects derived from a `BaseGameEntity`. You can find the code in the `EntityFunctionTemplates.h` header and it looks like this:


```
template <class T, class conT>
void EnforceNonPenetrationConstraint(const T& entity,
                                     const conT& ContainerOfEntities)
{
    //iterate through all entities checking for any overlap of bounding radii
    for (typename conT::const_iterator curEntity = ContainerOfEntities.begin();
         curEntity != ContainerOfEntities.end();
         ++curEntity)
    {
        //make sure we don't check against the individual
        if (*curEntity == entity) continue;

        //calculate the distance between the positions of the entities
        Vector2D ToEntity = entity->Pos() - (*curEntity)->Pos();

        double DistFromEachOther = ToEntity.Length();

        //if this distance is smaller than the sum of their radii then this
        //entity must be moved away in the direction parallel to the
        //ToEntity vector
        double AmountOfOverlap = (*curEntity)->BRadius() + entity->BRadius() -
                                   DistFromEachOther;
        if (AmountOfOverlap >= 0)
        {
            //move the entity a distance away equivalent to the amount of overlap.
            entity->SetPos(entity->Pos() + (ToEntity/DistFromEachOther) *
                           AmountOfOverlap);
        }
    }
    //next entity
}
```

You can watch the non-penetration constraint in action by running the craftily named `Non Penetration Constraint.exe` demo. Try altering the amount of separation to see what effect it has on the vehicles.

 **NOTE** For large numbers of densely packed vehicles such as you would see in big congested flocks, the non-penetration constraint will fail occasionally and there will be some overlap. Fortunately, this is not usually a problem as the overlap is difficult to see with the human eye.

Coping with Lots of Vehicles: Spatial Partitioning

When you have many interacting vehicles, it becomes increasingly inefficient to tag neighboring entities by comparing each one with every other one. In algorithm theory, something called *Big O* notation is used to express the relationship of time taken to the number of objects being processed. The all-pairs method we have been using to search for neighboring vehicles can be said to work in $O(n^2)$ time. This means that as the number of vehicles grows, the time taken to compare them increases in proportion to the square of their number. You can easily see how the time taken will escalate rapidly. If processing one object takes 10 seconds, then processing 10 objects will take 100 seconds. Not good, if you want a flock of several hundred birds!

Large speed improvements can be made by partitioning the world space. There are many different techniques to choose from. You've probably heard of many of them — BSP trees, quad-trees, oct-trees, etc. — and may even have used them, in which case you'll be familiar with their advantages. The method I use here is called *cell-space partitioning*, sometimes called bin-space partitioning (that's *not* short for binary space partitioning by the way; in this case “bin” really means bin). With this method, 2D space is divided up into a number of cells (or bins). Each cell contains a list of pointers to all the entities it contains. This is updated (in an entity's update method) every time an entity changes position. If an entity moves into a new cell, it is removed from its old cell's list and added to the current one.

This way, instead of having to test every vehicle against every other, we can just determine which cells lie within a vehicle's neighborhood and test against the vehicles contained in those cells. Here is how it's done step by step:

1. First of all, an entity's bounding radius is approximated with a box. See Figure 3.18.
2. The cells that intersect with this box are tested to see if they contain any entities.

Coping with Lots of Vehicles: Spatial Partitioning

3. All the entities contained within the cells from step 2 are examined to see if they are positioned within the neighborhood radius. If they are, they are added to the neighborhood list.

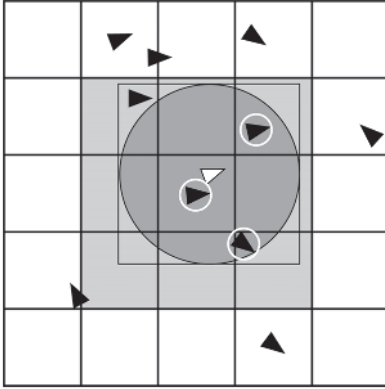


Figure 3.18. Cell-space partitioning. The circled vehicles are those within the white vehicle's neighborhood region.



3D NOTE If you are working in 3D, simply make the cells cubes and use a sphere as the neighborhood region.

If entities maintain a minimum separation distance from each other, then the number of entities each cell can contain is finite and cell space partitioning will operate in $O(n)$ time. This means the time taken to process the algorithm is directly proportional to the number of objects it's operating on. If the number of objects is doubled, the time taken is only doubled and not squared as with $O(n^2)$ algorithms. This implies the advantage you gain using space partitioning over the standard all-pairs technique is dependent on how many agents you have moving around. For small numbers, say less than fifty, there is no real advantage; but for large numbers, cell-space partitioning can be much faster. Even if the entities do not maintain a minimum separation distance and there is occasional overlap, on average the algorithm will perform much better than $O(n^2)$.

I have implemented cell-space partitioning as a class template: `CellSpacePartition`. This class uses another class template, `Cell`, to define the cell structure.

```
template <class entity>
struct Cell
{
    //all the entities inhabiting this cell
    std::list<entity>    Members;

    //the cell's bounding box (it's inverted because the Windows' default
    //coordinate system has a y-axis that increases as it descends)
    InvertedAABBox2D    BBox;
```

```

    Cell(Vector2D topleft,
         Vector2D botright):BBox(InvertedAABBox2D(topleft, botright))
    {}
};

```

A Cell is a very simple structure. It contains an instance of a bounding box class, which defines its extents, and a list of pointers to all those entities that are situated within this bounding area.

The CellSpacePartition class definition is as follows:

```

template <class entity>
class CellSpacePartition
{
private:

    //the required number of cells in the space
    std::vector<Cell<entity> >      m_Cells;

    //this is used to store any valid neighbors when an agent searches
    //its neighboring space
    std::vector<entity>              m_Neighbors;

    //this iterator will be used by the methods next and begin to traverse
    //through the above vector of neighbors
    std::vector<entity>::iterator    m_curNeighbor;

    //the width and height of the world space the entities inhabit
    double   m_dSpaceWidth;
    double   m_dSpaceHeight;

    //the number of cells the space is going to be divided into
    int       m_iNumCellsX;
    int       m_iNumCellsY;

    double    m_dCellSizeX;
    double    m_dCellSizeY;

    //given a position in the game space, this method determines the
    //relevant cell's index
    inline int PositionToIndex(const Vector2D& pos)const;

public:

    CellSpacePartition(double width,          //width of the environment
                       double height,        //height ...
                       int   cellsX,         //number of cells horizontally
                       int   cellsY,         //number of cells vertically
                       int   MaxEntities);   //maximum number of entities to add

    //adds entities to the class by allocating them to the appropriate cell
    inline void AddEntity(const entity& ent);

    //update an entity's cell by calling this from your entity's Update method

```

Coping with Lots of Vehicles: Spatial Partitioning

```

inline void UpdateEntity(const entity& ent, Vector2D OldPos);

//this method calculates all a target's neighbors and stores them in
//the neighbor vector. After you have called this method use the begin,
//next, and end methods to iterate through the vector.
inline void CalculateNeighbors(Vector2D TargetPos, double QueryRadius);

//returns a reference to the entity at the front of the neighbor vector
inline entity& begin();

//this returns the next entity in the neighbor vector
inline entity& next();

//returns true if the end of the vector is found (a zero value marks the end)
inline bool end();

//empties the cells of entities
void EmptyCells();
};

```

The class initializes `m_Neighbors` to have a maximum size equal to the total number of entities in the world. The iterator methods `begin`, `next`, and `end` and the `CalculateNeighbors` method manually keep track of valid elements inside this vector. This is to prevent the slowdown associated with the memory allocation and deallocation costs of repeatedly calling `std::vector::clear()` and `std::vector::push_back()` many times a second. Instead, previous values are simply overwritten and a zero value is used to mark the end of the vector.

Here is the listing for the `CalculateNeighbors` method. Notice how it follows the steps described earlier to determine a vehicle's neighbors.

```

template<class entity>
void CellSpacePartition<entity>::CalculateNeighbors(Vector2D TargetPos,
                                                    double QueryRadius)
{
    //create an iterator and set it to the beginning of the neighbor list
    std::list<entity>::iterator curNbor = m_Neighbors.begin();

    //create the query box that is the bounding box of the target's query
    //area
    InvertedAABBBox2D QueryBox(TargetPos - Vector2D(QueryRadius, QueryRadius),
                               TargetPos + Vector2D(QueryRadius, QueryRadius));

    //iterate through each cell and test to see if its bounding box overlaps
    //with the query box. If it does and it also contains entities then
    //make further proximity tests.
    std::vector<Cell<entity>> >::iterator curCell;
    for (curCell=m_Cells.begin(); curCell!=m_Cells.end(); ++curCell)
    {
        //test to see if this cell contains members and if it overlaps the
        //query box
        if (curCell->pBBBox->isOverlappedWith(QueryBox) &&
            !curCell->Members.empty())
        {

```

```
//add any entities found within query radius to the neighbor list
std::list<entity>::iterator it = curCell->Members.begin();
for (it; it!=curCell->Members.end(); ++it)
{
    if (Vec2DDistanceSq((*it)->Pos(), TargetPos) <
        QueryRadius*QueryRadius)
    {
        *curNbor++ = *it;
    }
}
}
} //next cell

//mark the end of the list with a zero.
*curNbor = 0;
}
```

You can find the full implementation of this class in `Common/misc/CellSpacePartition.h`. I have added cell space partitioning to the demo `Big_Shoal.exe`. It's now called `Another_Big_Shoal.exe`. You can toggle the partitioning on and off and see the difference it makes to the frame rate. There is also an option to view how the space is divided (default is 7 x 7 cells) and to see the query box and neighborhood radius of one of the agents.

TIP When applying the steering force to some vehicle types it can be useful to resolve the steering vector into forward and side components. For a car, for example, this would be analogous to creating the throttle and steering forces, respectively. To this end, you will find the methods `ForwardComponent` and `SideComponent` in the 2D `SteeringBehaviors` class used in this chapter's accompanying project file.

Smoothing

When playing with the demos, you may have noticed that sometimes a vehicle can twitch or jitter slightly when it finds itself in a situation with conflicting responses from different behaviors. For example, if you run one of the Big Shoal demos and switch the obstacles and walls on, you will see that sometimes when the “shark” agent approaches a wall or an obstacle, its nose shudders or trembles a little. This is because in one update step the **obstacle avoidance** behavior returns a steering force away from the obstacle but in the next update step there is no threat from the obstacle, so one of the agent’s other active behaviors may return a steering force pulling its heading back toward the obstruction, and so on, creating unwanted oscillations in the vehicle’s heading. Figure 3.19 shows how these oscillations can be started with just two conflicting behaviors: **obstacle avoidance** and **seek**.

Smoothing

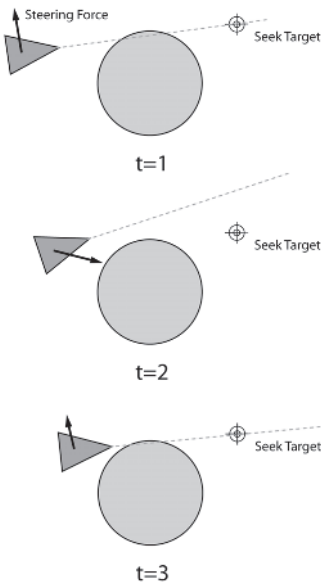


Figure 3.19. Conflicting behaviors can produce “judder.”

This shakiness is usually not too noticeable. Occasionally though, there will be times when it will be preferable for the shaking not to occur. So how do you stop it? Well, as the vehicle’s velocity is always aligned with its heading, stopping the shaking is not trivial. To negotiate the scenario given in Figure 3.19 successfully and smoothly, the vehicle needs to be able to foresee the conflict ahead of time and change behavior accordingly. Although this can be done, the solution can require a lot of calculation and additional baggage. A simple alternative suggested by Robin Green of Sony is to decouple the heading from the velocity vector and to average its value over several update steps. While this solution isn’t perfect, it produces adequate results at low cost (relative to any other solution I know about). To facilitate this, another member variable is added to the `Vehicle` class: `m_vSmoothedHeading`. This vector records the average of a vehicle’s heading vector and is updated each simulation step (in `Vehicle::Update`), using a call to an instance of a `Smoother` — a class that samples a value over a range and returns the average. This is what the call looks like:

```
if (SmoothingIsOn())
{
    m_vSmoothedHeading = m_pHeadingSmoother->Update(Heading());
}
```

This smoothed heading vector is used by the world transform function in the render call to transform a vehicle’s vertices to the screen. The number of update steps the `Smoother` uses to calculate the average is set in

params.ini and is assigned to the variable NumSamplesForSmoothing. When adjusting this value, you should try to keep it as low as possible to avoid unnecessary calculations and memory use. Using *very* high values produces weird behavior. Try using a value of 100 for NumSamplesForSmoothing and you'll see what I mean. It reminds me of a quote from *The Hitchhiker's Guide to the Galaxy*:

"You know," said Arthur with a slight cough, "if this is Southend, there's something very odd about it..."

"You mean the way the sea stays steady and the buildings keep washing up and down?" said Ford. "Yes, I thought that was odd too."

You can see the difference smoothing makes if you run the Another_Big_Shoal with Smoothing executable.

Practice Makes Perfect

In his paper "Steering Behaviors for Autonomous Characters," Reynolds describes a behavior called *leader following*. Leader following is a behavior that creates a steering force to keep multiple vehicles moving in single file behind a leader vehicle. If you've ever watched goslings follow their mother you'll know what I mean. To create this sort of behavior the followers must **arrive** at an offset position behind the vehicle in front while using **separation** to remain apart from one another. See Figure 3.20.

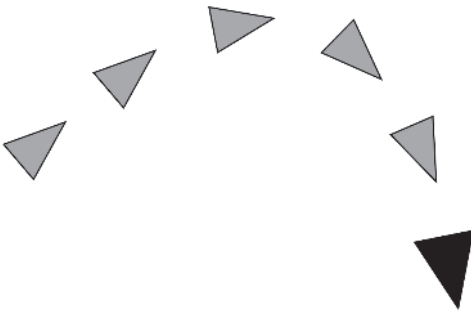


Figure 3.20. Leader following

Leader following can be improved further by creating a behavior that steers a vehicle laterally away from the direction of the leader if it finds itself in the leader's path.

Create a group of 20 vehicles that behave like a flock of sheep. Now add a *user-controlled vehicle* you can steer using the keyboard. Program your sheep so they believe the vehicle is a dog. Can you get the flock's behavior to look realistic?