# Understanding Steering Behaviors: Movement Manager

Steering behaviors are great for creating realistic movement patterns, but they are even greater if you can control, use and combine them easily. In this tutorial, I'll discuss and cover the implementation of a movement manager for [all of our previously discussed behaviors](http://gamedev.tutsplus.com/series/understanding-steering-behaviors/).

***Note:****Although this tutorial is written using AS3 and Flash, you should be able to use the same techniques and concepts in almost any game development environment. You must have a basic understanding of math vectors.*

**Combining Steering Forces**

As [previously discussed](http://gamedev.tutsplus.com/tutorials/implementation/understanding-steering-behaviors-seek/), every steering behavior produces a resulting force (called a "steering force") that is added to the velocity vector. The direction and magnitude of that force will drive the character, making it move according to a pattern ([seek](http://gamedev.tutsplus.com/tutorials/implementation/understanding-steering-behaviors-seek/), [flee](http://gamedev.tutsplus.com/tutorials/implementation/understanding-steering-behaviors-flee-and-arrival/), [wander](http://gamedev.tutsplus.com/tutorials/implementation/understanding-steering-behaviors-wander/), and so on). The general calculation is:

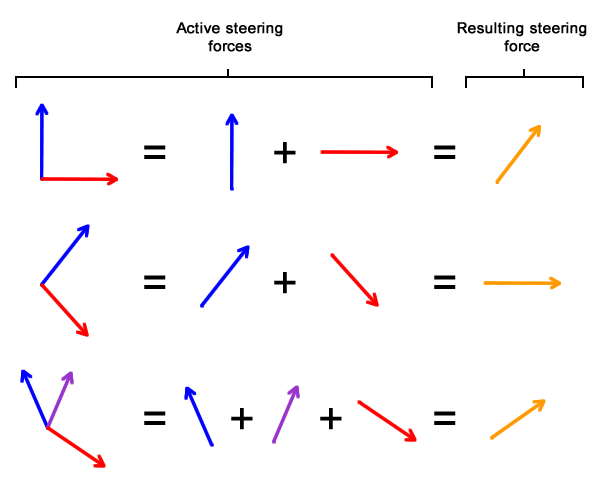
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | steering = seek(); // this can be any behavior  steering = truncate (steering, max\_force)  steering = steering / mass    velocity = truncate (velocity + steering , max\_speed)  position = position + velocity |

Since the steering force is a vector, it can be added to any other vector (just like the velocity). However the real  "magic" lies in the fact that you can add several steering forces together - it's as simple as:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | steering = nothing(); // the null vector, meaning "zero force magnitude"  steering = steering + seek();  steering = steering + flee();  (...)  steering = truncate (steering, max\_force)  steering = steering / mass    velocity = truncate (velocity + steering , max\_speed)  position = position + velocity |

The combined steering forces will result in a vector that represents *all*those forces. In the code snippet above,  the resulting steering force will make the character seek something while *at the same time* it will flee something *else*.

Check below some examples of steering forces combined to produce a single steering force:

  
Steering forces combined.

**Complex Patterns Effortlessly**

The combination of steering forces will produce extremely complex movement patterns effortlessly. Imagine how hard it would be to write code to make a character seek something, but at the same time avoid an specific area, without using vectors and forces?

That would require the calculation of distances, areas, paths, graphs and the like. If things are moving around, all of those calculations must be repeated every now and then, because the environment changes constantly.

With steering behaviors, all forces are *dynamic*. They are meant to be calculated every game update, so they will naturally and seamlessly react to environment changes.

The demo below shows ships that will seek the mouse cursor, but will flee the center of the screen, both at the same time:

  
The ships will seek the mouse cursor (gray), but will flee the center of the screen (orange). Click to show forces.

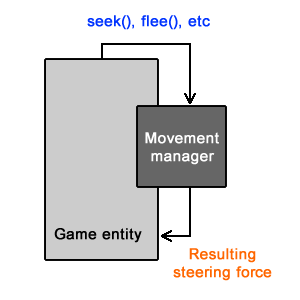
**Movement Manager**

In order to use several steering behaviors at the same time in a simple and easy way, a *movement manager* comes in handy. The idea is to create a "black box" that can be plugged into any existing entity, making it able to perform those behaviors.

The manager has a reference to the entity that it is plugged into (the "host"). The manager will provide the host with a set of methods, such as seek() and flee(). Every time such methods are invoked, the manager updates its internal properties to produce  a steering force vector.

After the manager processes all invocations,  it will add the resulting steering force to the host's velocity vector. That will change the host's velocity vector magnitude and direction according to the active behaviors.

The figure below demonstrates the architecture:

  
Movement manager: plugin architecture.

**Making Things Generic**

The manager has a set of methods, each one representing a distinct behavior. Every behavior must be supplied with different pieces of external information in order to work.

The [seek](http://gamedev.tutsplus.com/tutorials/implementation/understanding-steering-behaviors-seek/) behavior, for instance, needs a point in the space that is used to calculate the steering force towards that place; [pursue](http://gamedev.tutsplus.com/tutorials/implementation/understanding-steering-behaviors-pursuit-and-evade/) needs several pieces of information from its target, such as current position and velocity. A point in the space can be expressed as an instance of Point or Vector2D, both pretty common classes in any framework.

The target used in the pursue behavior, however, can be anything. In order to make the movement manager generic enough, it needs to receive a target that, independently of its type, is able to answer a few "questions", such as "*What is your current velocity?*". Using [some principles of object-oriented programming](http://gamedev.tutsplus.com/tutorials/implementation/quick-tip-the-oop-principle-of-abstraction/), it can be achieved with *interfaces*.

Assuming the interface IBoid describes an entity able to be handled by the movement manager, any class in the game can use steering behaviors, as long as it implements IBoid. That interface has the following structure:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public interface IBoid  {      function getVelocity() :Vector3D;      function getMaxVelocity() :Number;      function getPosition() :Vector3D;      function getMass() :Number;  } |

**Movement Manager Structure**

Now that the manager can interact with all game entities in a generic way, its basic structure can be created. The manager is composed of two properties (the resulting steering force and the host reference) and a set of public methods, one for each behavior:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class SteeringManager  {      public var steering :Vector3D;      public var host :IBoid;        // The constructor      public function SteeringManager(host :IBoid) {          this.host   = host;          this.steering   = new Vector3D(0, 0);      }        // The public API (one method for each behavior)      public function seek(target :Vector3D, slowingRadius :Number = 20) :void {}      public function flee(target :Vector3D) :void {}      public function wander() :void {}      public function evade(target :IBoid) :void {}      public function pursuit(target :IBoid) :void {}        // The update method.      // Should be called after all behaviors have been invoked      public function update() :void {}        // Reset the internal steering force.      public function reset() :void {}        // The internal API      private function doSeek(target :Vector3D, slowingRadius :Number = 0) :Vector3D {}      private function doFlee(target :Vector3D) :Vector3D {}      private function doWander() :Vector3D {}      private function doEvade(target :IBoid) :Vector3D {}      private function doPursuit(target :IBoid) :Vector3D {}  } |

When the manager is instantiated, it must receive a reference to the host it is plugged into. It will allow the manager to change the host velocity vector according to the active behaviors.

Every behavior is represented by two methods, a public and a private one. Using seek as an example:

|  |  |
| --- | --- |
| 1  2 | public function seek(target :Vector3D, slowingRadius :Number = 20) :void {}  private function doSeek(target :Vector3D, slowingRadius :Number = 0) :Vector3D {} |

The public seek() will be invoked to tell the manager to apply that specific behavior. The method has no return value and its parameters are related to the behavior itself, such as a point in the space. Under the hood the private method doSeek() will be invoked and its return value, the calculated steering force for that specific behavior, will be added to the manager's steering property.

The following code demonstrates the implementation of seek:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // The publish method.  // Receives a target to seek and a slowingRadius (used to perform arrive).  public function seek(target :Vector3D, slowingRadius :Number = 20) :void {      steering.incrementBy(doSeek(target, slowingRadius));  }    // The real implementation of seek (with arrival code included)  private function doSeek(target :Vector3D, slowingRadius :Number = 0) :Vector3D {      var force :Vector3D;      var distance :Number;        desired = target.subtract(host.getPosition());        distance = desired.length;      desired.normalize();        if (distance <= slowingRadius) {          desired.scaleBy(host.getMaxVelocity() \* distance/slowingRadius);      } else {          desired.scaleBy(host.getMaxVelocity());      }        force = desired.subtract(host.getVelocity());        return force;  } |

All other behavior methods are implemented in a very similar way. The pursuit() method, for instance, will look like this:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16 | public function pursuit(target :IBoid) :void {      steering.incrementBy(doPursuit(target));  }    private function doPursuit(target :IBoid) :Vector3D {      distance = target.getPosition().subtract(host.getPosition());        var updatesNeeded :Number = distance.length / host.getMaxVelocity();        var tv :Vector3D = target.getVelocity().clone();      tv.scaleBy(updatesNeeded);        targetFuturePosition = target.getPosition().clone().add(tv);        return doSeek(targetFuturePosition);  } |

Using the code from [previous tutorials](http://gamedev.tutsplus.com/series/understanding-steering-behaviors/), all you have to do is to adapt them in the form of behavior() and doBehavior(), so they can be added to the movement manager.

**Applying and Updating Steering Forces**

Every time a behavior's method is invoked, the resulting force it produces is added to the manager's steering property. As a consequence that property will accumulate all steering forces.

When all behaviors have been invoked, the manager must apply the current steering force to the hosts velocity, so it will move according to the active behaviors. It is performed in the update() method of the movement manager:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | public function update():void {      var velocity :Vector3D = host.getVelocity();      var position :Vector3D = host.getPosition();        truncate(steering, MAX\_FORCE);      steering.scaleBy(1 / host.getMass());        velocity.incrementBy(steering);      truncate(velocity, host.getMaxVelocity());        position.incrementBy(velocity);  } |

The method above must be invoked by the host (or any other game entity) after all behaviors have been invoked, otherwise the host will never change its velocity vector to match the active behaviors.

**Usage**

Let's assume a class named Prey should move using steering behavior, but at the moment it has no steering code nor the movement manager. Its structure will look like this:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class Prey  {      public var position  :Vector3D;      public var velocity  :Vector3D;      public var mass      :Number;        public function Prey(posX :Number, posY :Number, totalMass :Number) {          position    = new Vector3D(posX, posY);          velocity    = new Vector3D(-1, -2);          mass        = totalMass;            x = position.x;          y = position.y;      }        public function update():void {          velocity.normalize();          velocity.scaleBy(MAX\_VELOCITY);          velocity.scaleBy(1 / mass);            truncate(velocity, MAX\_VELOCITY);          position = position.add(velocity);            x = position.x;          y = position.y;      }  } |

Using that structure, the class instances can move using [Euler integration](http://en.wikipedia.org/wiki/Euler_method), just like the very first demo of the [seek tutorial](http://gamedev.tutsplus.com/tutorials/implementation/understanding-steering-behaviors-seek/). In order to make it able to use the manager, it needs a property referencing the movement manager and it must implement the IBoid interface:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | public class Prey implements IBoid  {      public var position  :Vector3D;      public var velocity  :Vector3D;      public var mass      :Number;      public var steering  :SteeringManager;        public function Prey(posX :Number, posY :Number, totalMass :Number) {          position    = new Vector3D(posX, posY);          velocity    = new Vector3D(-1, -2);          mass        = totalMass;          steering    = new SteeringManager(this);            x = position.x;          y = position.y;      }        public function update():void {          velocity.normalize();          velocity.scaleBy(MAX\_VELOCITY);          velocity.scaleBy(1 / mass);            truncate(velocity, MAX\_VELOCITY);          position = position.add(velocity);            x = position.x;          y = position.y;      }        // Below are the methods the interface IBoid requires.        public function getVelocity() :Vector3D {          return velocity;      }        public function getMaxVelocity() :Number {          return 3;      }        public function getPosition() :Vector3D {          return position;      }        public function getMass() :Number {          return mass;      }  } |

The update() method must be changed accordingly so the manager can be updated as well:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14 | public function update():void {      // Make the prey wander around...      steering.wander();        // Update the manager so it will change the prey velocity vector.      // The manager will perform the Euler intergration as well, changing      // the "position" vector.      steering.update();        // After the manager has updated its internal structures, all we must      // do is update our position according to the "position" vector.      x = position.x;      y = position.y;  } |

All behaviors can be used at the same time, as long as all method calls are made before the manager's update() invocation, which applies the accumulated steering force to the velocity vector of the host.

The code below demonstrates another version of the Prey's update() method, but this time it will seek a position in the map and evade another character (both at the same time):

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18 | public function update():void {      var destination :Vector3D = getDestination(); // the place to seek      var hunter :IBoid = getHunter(); // get the entity who is hunting us        // Seek the destination and evade the hunter (at the same time!)      steering.seek(destination);      steering.evade(hunter);        // Update the manager so it will change the prey velocity vector.      // The manager will perform the Euler intergration as well, changing      // the "position" vector.      steering.update();        // After the manager has updated its internal structures, all we must      // do is update our position according to the "position" vector.      x = position.x;      y = position.y;  } |

Advertisement

**Demo**

The demo below demonstrates a complex movement pattern where several behaviors are combined. There are two types of characters in the scene: the **Hunter** and the **Prey**.

The hunter will *pursue* a prey if it gets close enough; it will pursue for as long as the stamina supply lasts; when it runs out of stamina, the pursuit is interrupted and the hunter will *wander* until it recovers its stamina levels.

Here is the Hunter's update() method:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public function update():void {      if (resting && stamina++ >= MAX\_STAMINA) {          resting = false;      }        if (prey != null && !resting) {          steering.pursuit(prey);          stamina -= 2;            if (stamina <= 0) {              prey = null;              resting = true;          }      } else {          steering.wander();          prey = getClosestPrey(position);      }        steering.update();        x = position.x;      y = position.y;  } |

The prey will *wander* indefinitely. If the hunter gets too close, it will *evade*. If the mouse cursor is near and there is no hunter around, the prey will *seek* the mouse cursor.

Here is the Prey's update() method:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21 | public function update():void {      var distance :Number = Vector3D.distance(position, Game.mouse);        hunter = getHunterWithinRange(position);        if (hunter != null) {          steering.evade(hunter);      }        if (distance <= 300 && hunter == null) {          steering.seek(Game.mouse, 30);        } else if(hunter == null){          steering.wander();      }        steering.update();        x = position.x;      y = position.y;  } |

The final result (gray is wander, green is seek, orange is pursue, red is evade):

  
The hunting. Click to show forces.

**Conclusion**

A movement manager is very useful for controlling several steering behaviors at the same time. The combination of such behaviors can produce very complex movement patterns, allowing a game entity to seek one thing at the same time it evades another.

I hope you liked the management system discussed and implement in this tutorial and use it in your games. Thank you for reading! Don't forget to keep up to date by following us on [Twitter](https://twitter.com/envatogame), [Facebook](https://www.facebook.com/gamedevtuts), or [Google+](https://plus.google.com/113664840006452864249/posts).