Inteligencia Artificial para Videojuegos – Capítulo 2

Introducción

En esta unidad veremos una técnica de Inteligencia Artificial que nos servirá para implementar personajes que controlen sus movimientos para lograr ciertas acciones como moverse entre puntos, interceptar objetivos, evadir obstáculos. Esta técnica se conoce como Steering Behaviors o Comportamientos de Control. Esta fue inventada por Craig Reynolds en 1986 para simular el comportamiento de rebaños (flock). Luego esta técnica gano un lugar en el desarrollo de video juegos y las animaciones en la industria del cine, por ejemplo en el juego Half-Life fueron utilizadas para el comportamiento de las criaturas pseudo aves que se pueden ver al final del juego en Xeon. Esta técnica además es complementaria con FSM, pudiendo mezclarlas y obtener así como resultado personajes más complejos.

Steering Behaviors

¿Porqué?

Comencemos esta unidad de la misma forma que la unidad anterior, viendo que ofrecen hacer por nosotros estos dichosos Steering Behaviors:

- Son simples de implementar y sencillos de comprender.
- Pueden lograr que nuestros personajes en un juego tengan un comportamiento muy natural y orgánico.
- En base a varios comportamientos básicos que veremos más adelante (por ejemplo perseguir, arribar, deambular, etc) podemos lograr comportamientos muy complejos y útiles (por ejemplo podríamos simular como reaccionaria un ejercito de vikingos en un video juego).
- Es fácil idear nuevos comportamientos haciendo mezclas con los comportamientos básicos.
- Pueden mezclarse de forma muy simple con una FSM y así lograr diseñar personajes más complejos para nuestros juegos.
- Pueden ser utilizados para solucionar ciertos problemas complejos (por ejemplo podría ser evadir obstáculos dinámicos) de forma suficiente para muchos juegos, siendo una solución barata computacionalmente hablando y económicamente hablando ya que su implementación es mas simple que otras alternativas.

Agentes autónomos… ¿Cómo se come eso?

Como dijimos antes los Steering Behaviors nos servirán para simular agentes autónomos. Aunque todavía no dijimos que es exactamente un agente autónomo seguramente ya tendrán intuitivamente una idea, que posiblemente sea la idea correcta. De todas formas formalicemos el concepto dando una definición:

> ***Un agente autónomo es un sistema que esta situado y forma parte de un ambiente, obteniendo sensaciones del ambiente y realizando acciones sobre este a lo largo del tiempo en la búsqueda de cumplir sus objetivos.***

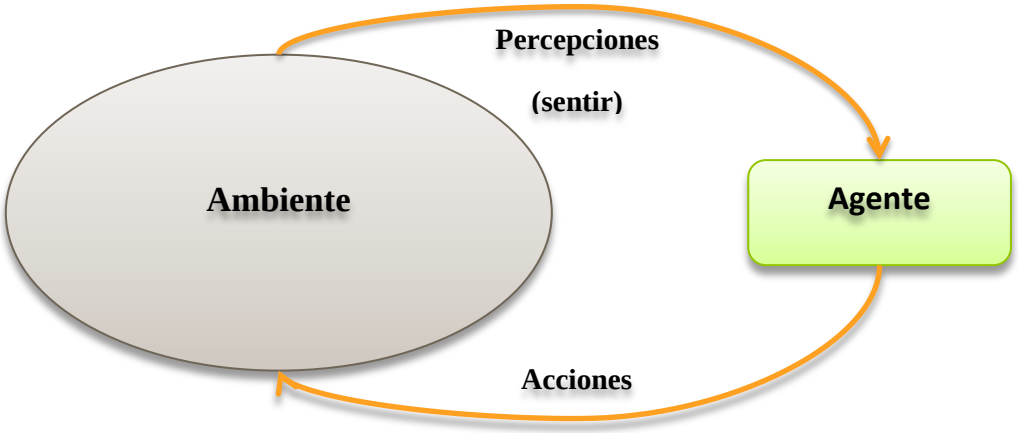

ILUSTRACIÓN 1: ESQUEMA DE UN AGENTE AUTÓNOMO

Es decir que un agente se relaciona de dos formas con el ambiente al cual pertenece, percibe sensaciones de éste y realiza acciones sobre éste modificándolo. Veamos esto en un caso más concreto. Imaginemos que estamos programando un Pacman, donde nuestro agente de interés a estudiar será un fantasmita. El ambiente al cual pertenece el fantasmita será el tablero de juego que ya todos conocemos, donde tenemos caminos que podemos tomar, pastillas pequeñas y grandes, fantasmitas (incluyéndonos a nosotros mismos) y el siempre odiado Pacman. Nuestro fantasmita tendrá ciertos sentidos que le permitirán ver de forma parcial el ambiente en el cual se encuentra, en general estos sentidos permitirán saber si Pacman u otro fantasmita se encuentra a la vista y si Pacman comió una pastilla grande por lo cual es peligroso mientras le dure el efecto. Como el fantasmita es un agente inteligente podrá actuar realizando acciones sobre la información que tiene del ambiente a través de esos sentidos, ejemplos de esas acciones podrían ser vagar por el tablero evitando chocar con otros fantasmitas, si ve a Pacman y este no esta bajo el efecto de una pastilla grande perseguirlo o por el contrario si este esta bajo el efecto de una pastilla grande evadirlo. Notemos que nos referimos al agente por autónomo ya que el fantasmita en este caso reaccionara de forma automática, sin ninguna intervención del programador. Es decir que nuestro fantasmita intentara cumplir sus objetivos de permanecer vivo e intentar comer a Pacman de forma automática.

El movimiento de un agente autónomo puede ser analizado en tres capas:

- **Selección de Acciones:** Esta capa del comportamiento del agente es responsable de elegir sus objetivos y decidir que plan seguir para cumplirlos. Es decir que es la parte que dice “Ve hacia el punto X” y “Haz A, B y luego C”.
- **Control:** Esta capa se encarga de calcular las direcciones necesarias para satisfacer los objetivos y planes que impuso la capa de Selección de Acciones. Los Steering Behaviors son una implementación de esta capa. Ellos producen las fuerzas que dicen de que forma deberá moverse el agente para cumplir lo que requirió la capa de Selección de Acciones.
- **Locomoción:** Es la capa base, esta establece los aspectos mas mecánicos del movimiento del agente. Por ejemplo un caballo o un automóvil realizaran de diferente forma la acción de moverse hacia el norte, por ejemplo el caballo puede girar sobre su propio eje mientras

que el automóvil puede requerir hacer algunas maniobras para girar. Manteniendo separada esta capa de la capa de Control podemos implementar distintos agentes (por ejemplo un caballo o un automóvil) con movimientos muy diferentes que utilicen la misma capa de Control (es decir los mismo Steering Behaviors).

Usemos una analogía para explicar mejor cual es la función de cada una de estas capas. Para eso robémosle (argentinizándola de paso) la genial idea que uso el padre de esta técnica, Craig Reynolds, para explicarla:

> *Imaginémonos que hay un gaucho, montando su corcel, cuidando un rebaño de vacas en una pastura. En un momento el estanciero le informa al gaucho que arree de vuelta al rebaño a un grupo de vacas que se alejó del grupo. Entonces el siempre diligente gaucho cabalga su corcel guiándolo hacia las vacas alejadas, tomando el camino más conveniente y evadiendo los obstáculos que encuentre en su camino. En esta analogía el estanciero representa la capa de Selección de Acciones: él censa del ambiente que un grupo de vacas se alejaron peligrosamente del rebaño, y debido a esto establece el objetivo de arrear las vacas alejadas devuelta al rebaño. El gaucho representa la capa de Control: dividiendo el objetivo establecido en sub-objetivos (cabalgar hacia las vacas, evadir obstáculos y zonas peligrosas, arrear las vacas hacia el rebaño). Cada uno de esos sub-objetivos estaría implementado por uno o una combinación de Steering Behaviors, que controlarían el accionar del par gaucho/caballo. El gaucho controlara como se mueve su corcel usando varias señales (físicas y vocales). En general esas señales expresan conceptos como: ir más despacio, ir más deprisa, girar a la izquierda, girar a la derecha, etc. El caballo representa la capa de Locomoción, recibiendo las señales del gaucho como entrada, éste se desplazara de la forma indicada. Ese desplazamiento será el resultado de la interacción del sentido de visión del corcel, de su sentido del balance y de sus músculos aplicando fuerzas a su estructura ósea.*

En esta unidad veremos como podemos implementar las dos capas inferiores, la de Control y de Locomoción. La primera utilizando Steering Behaviors y la segunda un modelo de vehículo simple. Mas adelante en el curso veremos otras técnicas que pueden utilizarse para implementar la capa de Selección de Acciones.

Diseño en general

Antes que nada hablando de algunas clases bases sobre la que se apoyara todo el código. Comencemos hablando de la clase de la que todo objeto físico de la escena heredara. Esta clase hará uso de dos librerías que ya conocen: SFML para graficarse y Box2D para resolver su física. Sin más vueltas el código de la clase será[1]:

```
class EntidadEscena
{

	//Forma de SFML para mostrar el objeto
	sf::Drawable* m_pGrafico;

	//El cuerpo rigido del avatar
	b2Body*    m_pCuerpo;

	//El fixture del cuerpo rigido, se almacena para limpiar el avatar
	//de la escena
	b2Fixture* m_pAdorno;

public:

	EntidadEscena(const b2BodyDef& CuerpoDef, const b2FixtureDef& AdornoDef,
	              float escala, sf::Image* pTextura);

	~EntidadEscena();

	//Actualiza la entidad
	virtual void Actualizar(float dt);

	//Dibuja el avatar
	void Dibujar(sf::RenderWindow &RW);

	//Aplica una fuerza sobre el cuerpo
	//Parametros:
	//     -El vector de fuerza a aplicar
	//     -El origen de aplicacion de la fuerza en coordenadas del mundo
	void AplicarFuerzaMundo(const sf::Vector2f& fuerza,
	              const sf::Vector2f& origen = sf::Vector2f(0.0f,0.0f));

	//Aplica una fuerza sobre el cuerpo
	//Parametros:
	//     -El vector de fuerza a aplicar
	//     -El origen de aplicacion de la fuerza en coordenadas locales
	void AplicarFuerzaLocal(const sf::Vector2f& fuerza,
	              const sf::Vector2f& origen = sf::Vector2f(0.0f,0.0f));

	/*Se omitieron los métodos de Get y Set y detalles menores*/

};
```

[1] En el código final hay una clase más de la que hereda EntidadEscena, esta clase se llama EntidadBase. Aquí se la omitió porque no es necesario saber de ella para esta explicación.

Luego crearemos una clase llamada Vehiculo la cual hereda de EntidadEscena, y que será la cual tenga los comportamientos de control que harán que el vehículo se mueva de forma autónoma dependiendo de que comportamientos este vehículo tenga activos. Los comportamientos propiamente dichos estarán embebidos en una clase llamada SteeringBehaviors (sorpresa!), que cuando se actualiza (la clase Vehiculo se encarga de actualizar sus comportamientos de control) calculara una fuerza que luego el vehículo aplicara sobre si misma para aplicar los comportamientos. Podemos ver en la Ilustración 2 un grafico donde se resume esto que acabamos de explicar, donde EntidadEscena es la base de todo, Vehiculo hereda de esta y tiene una instancia a un objeto SteeringBehaviors, el cual lo usa para modificar su movimiento y así moverse autónomamente.

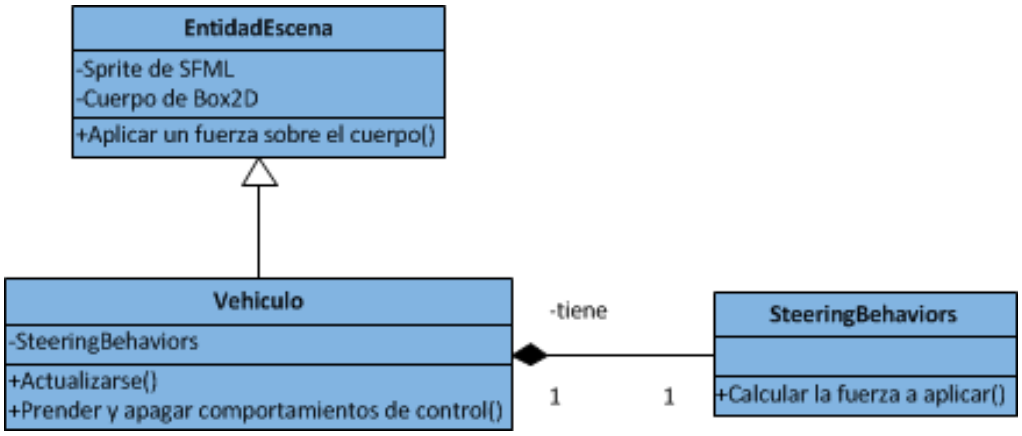

ILUSTRACIÓN 2: DISEÑO DE CLASES

No se asusten porque parecen muchas clases, vamos a ver bien en detalle para que sirve cada una. Además esas son absolutamente todas las clases que vamos a tener, así que a no desesperar y seguir leyendo.

Locomoción

Empecemos implementando la clase que se encargara de la capa de Locomoción del modelo de agente autónomo del que hablamos antes. Esta clase será Vehicula de la cual también ya hablamos. Ésta solo representa una EntidadEscena que puede tener comportamientos de control y ser controlados por estos. Lo que diferenciara a la clase Vehiculo es que tendrá una instancia de la clase SteeringBehaviors y la utilizara para saber que fuerza debe aplicar sobre si mismo para moverse. Veamos la declaración de la clase:

```
class Vehiculo: public EntidadEscena
{
private:

	SteeringBehaviors m_Comportamiento;

public:

	//Fuerza maxima que pueden aplicar los comportamientos sobre el vehiculo
	float m_FuerzaSteeringMax;

	//Velocidad maxima a la cual queremos que se mueva el vehiculo
	float m_VelocidadMax;

	Vehiculo(const b2BodyDef& CuerpoDef, const b2FixtureDef& AdornoDef,
		 float escala, sf::Image* pTextura);

	//Actualiza la entidad
	virtual void Actualizar(float dt);

	float GetFuerzaSteeringMax() const;

	float GetVelocidadMax() const;

	//Podemos obtener le vector que nos dice hacia donde mira el vehiculo
	sf::Vector2f GetDireccion() const;

	//Utilizaremos esta función para obtener el objeto SteeringBehaviors
	//y asi activar y desactivar comportamientos
	SteeringBehaviors& GetSteeringBehaviors();

};
```

Vemos que en la clase se declara una variable llamada m_FuerzaSteeringMax, ésta establece la máxima fuerza que los comportamientos de control pueden aplicar sobre el vehículo. Esto es necesario para controlar que el vehículo no “enloquezca” en algún caso, por ejemplo que alguno o muchos comportamientos den una fuerza muy grande que haga que el vehículo deje de ser estable. También hay una variable llamada m_VelocidadMax, ésta dice cual es la velocidad máxima a la cual se podrá mover el vehículo. Es útil limitar esto para evitar que el vehículo se vuelva él mismo inmanejable.

Para entender como utiliza Vehiculo los SteeringBehaviors debemos dejar bien en claro que se puede hacer con un objeto de este tipo:

- SteeringBehaviors **permite prender y apagar comportamientos de control**. Por ejemplo podemos decirle que prenda el comportamiento Arribar (más adelante lo veremos) a la posición (x, y), y de esta misma forma podemos prender o apagar cualquier combinación de los comportamientos que describiremos más adelante.
- SteeringBehaviors tendrá un método Calcular, el cual **calculara dependiendo los comportamientos que tenga activos la fuerza que Vehiculo debe aplicar sobre si mismo**, así Vehiculo utilizara el método de aplicar fuerza heredado de EntidadEscena para aplicar la fuerza.

A este segundo punto la clase Vehiculo lo implementara en el método Actualizar, así podrá cada frame (o bucle a de actualización de física[2]) de nuestro juego actualizar la fuerza a aplicar. Veamos como implementamos eso:

```
void Vehiculo::Actualizar(float dt)
{
	//Llamamos a actualizar de la clase madre
	EntidadEscena::Actualizar(dt);

	//Le pedimos al objeto SteeringBehaviors que calcule la fuerza
	//resultante de los comportamientos
	sf::Vector2f fuerzaComportamiento = m_Comportamiento.Calcular();

	//Aplicamos la fuerza al vehiculo
	AplicarFuerzaLocal(fuerzaComportamiento);

	//Algunos detalles de como limitar la velocidad del vehiculo
	//y rotar la dirección en la que mira se omitieron aqui

}
```

[2] Recordemos que dijimos en MAVII que la física de un juego, si bien no es raro que se actualiza a la misma velocidad que el juego se dibuja, esto no es siempre así y la física puede estarse actualizando más o menos veces de la que se dibuja.

Control: Steering Behaviors

Ahora comenzaremos a hablar de los comportamientos en particular. Clasificaremos en dos grupos a los comportamientos: comportamientos primitivos y comportamientos complejos. Los primeros serán comportamientos relativamente simples como perseguir o escapar, mientras que los segundos serán comportamientos logrados a través de activar varios comportamientos primitivos al mismo tiempo que darán un resultado particular. Los comportamientos que veremos son:

- **Comportamientos primitivos**
  - *Buscar*
  - *Escapar*
  - *Arribar*
  - *Interceptar*
  - *Evadir*
  - *Deambular*
  - *Evadir Obstáculos*
  - *Evadir Paredes*
  - *Esconderse*
- **Comportamientos complejos**
  - *Un conjunto de comportamientos primitivos activos que en conjunto logran algo*

Estos comportamientos que nombramos no son todos los que existen, sino que son solo algunos. Antes de comenzar a explicarlos recordemos que es lo que estos comportamientos le dan al vehículo. Un comportamiento de control, si se encuentra activo, calculara una fuerza que aplicada al vehículo contribuirá a que el vehículo realice la acción que representa ese comportamiento, por ejemplo el comportamiento de Arribar contribuirá a que el vehículo valla a un determinado punto deteniéndose en éste. Si hay varios comportamientos de control activos al mismo tiempo, se calculara la fuerza de cada uno por separado y luego se las mezclara para obtener la fuerza que se debe aplicar al vehículo. Por mezclar fuerzas, podemos hacernos la idea de que sumaremos todas las fuerzas de los comportamientos activos, pero sobre el final veremos que existen formas mucho más inteligentes de hacerlo. Luego de explicar algunos comportamientos veremos en detalle como la clase SteeringBehaviors realiza esto de activar o desactivar comportamientos, calcular las fuerzas de los comportamientos activos y mezclarlos para devolver la fuerza que se aplicara al vehículo.

Steering Behaviors: Primitivos

Algunos de los comportamientos tienen implementaciones que no son para nada triviales. Pero no desesperemos ya que no pediremos que implementen todos por si mismos, sino que les daremos las implementaciones para que puedan usarlas y si lo desean revisar en más profundidad algunos detalles de implementación. Lo importante de cada comportamiento es saber que es lo que hace, saber conceptualmente como lo hace y saber como se lo puede llegar a implementar.

Buscar

Este comportamiento es uno de los más importantes. Lo que hace es que el vehículo se mueva a un punto que se le pasa por parámetro. Veamos primero la implementación y luego la explicaremos:

```
sf::Vector2f SteeringBehaviors::Buscar(const sf::Vector2f& PosObjetivo)
{
	sf::Vector2f VelocidadDeseada =
			Normalizar(PosObjetivo - m_pVehiculoDueño->GetPosicion());
	VelocidadDeseada *= m_pVehiculoDueño->GetVelocidadMax();

	return (VelocidadDeseada - m_pVehiculoDueño->GetVelocidadLineal());
}
```

En el método el parámetro PosObjetivo es la posición a la cual queremos que nuestro vehículo se desplace. La variable m_pVehiculoDueño es un puntero al objeto de tipo Vehiculo, al cual pertenece el comportamiento de control que estamos implementando, es decir el vehículo al que se quiere aplicar el comportamiento. Notemos que m_pVehiculoDueño es una propiedad de la clase SteeringBehaviors, que es exactamente lo que indica el prefijo “m_” del nombre; desde ahora en más toda variable que tenga como prefijo “m_” significara que es una propiedad de una clase. VelocidadDeseada calcula cual será la dirección en la cual se deberá mover el vehículo para llegar a PosObjetivo. Notemos que la llamamos dirección porque normalizamos le vector por lo que este tiene norma uno. Luego en la siguiente línea escalamos VelocidadDeseada por la máxima velocidad que tiene asignado el vehículo. Hasta este punto tenemos la velocidad deseada para llegar al punto al cual queremos desplazarnos, pero no tuvimos en cuanta que el vehículo seguramente ya se esta moviendo, tal vez incluso en esa misma dirección que indica VelocidadDeseada! ya que los comportamientos de control contribuyen a que el vehículo realice una acción pero no requieren que el vehículo este en reposo. Para solucionar este pequeño inconveniente calculamos la diferencia entre el vector de VelocidadDeseada y la velocidad actual del vehículo, esto nos dará un vector desde la punta de la velocidad actual del vehículo hasta la punta de VelocidadDeseada. Recordemos que por la suma de vectores si sumamos el resultado de esta última operación a la velocidad actual del vehículo hará que la velocidad actual sea igual a la VelocidadDeseada. En la Ilustración 3 podemos ver estos cálculos de forma grafica. Es importante que veamos que si bien la variable se llama VelocidadDeseada, no significa que el método Buscar devuelva una velocidad, sino que devuelve una fuerza. Pero la llamamos velocidad deseada porque luego de que apliquemos la fuerza resultante que devuelve Buscar al vehículo se hará una contribución a que la velocidad del vehículo comience a ser similar a VelocidadDeseada.

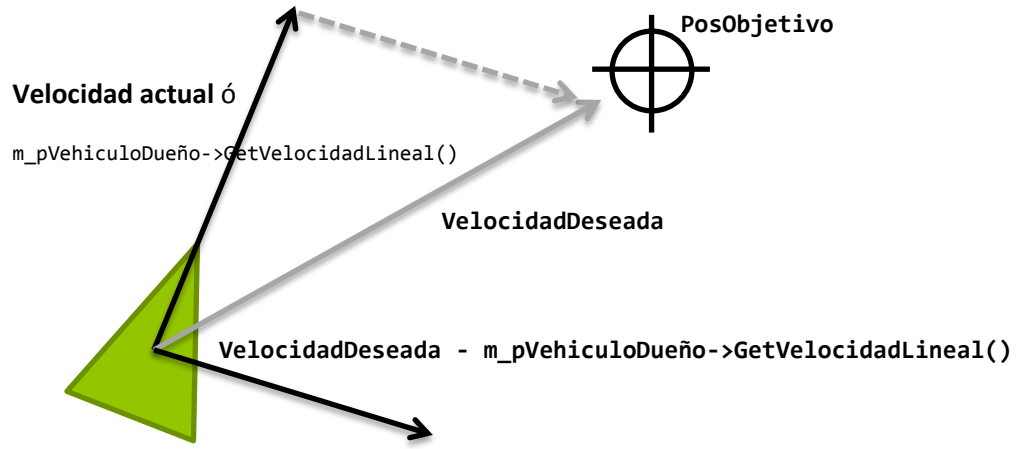

ILUSTRACIÓN 3: CALCULO DE COMPORTAMIENTO BUSCAR. EL TRIANGULO VERDE ES LA REPRESENTACIÓN DEL VEHÍCULO. CADA FLECHA INDICA UN VECTOR QUE SE CORRESPONDE CON ALGUNA VARIABLE DEL CÓDIGO.

Escapar

Este comportamiento es exactamente el opuesto al de Buscar. En vez de devolver una fuerza que aproxime el vehículo a un punto objetivo, hace que el vehículo se aleje (escapando) de esa posición. Veamos la implementación que es muy similar a la de Buscar:

```
sf::Vector2f SteeringBehaviors::Escapar(const sf::Vector2f& PosPeligro)
{
	//Solo escapa si esta en la distancia de panico del origen del peligro
	const double DistanciaDePanico = 100.0 * 100.0;
	if(Norma(m_pVehiculoDueño->GetPosicion()-PosPeligro) > DistanciaDePanico)
		return sf::Vector2f(0.0f, 0.0f);

	sf::Vector2f VelocidadDeseada =
		Normalizar(m_pVehiculoDueño->GetPosicion() - PosPeligro);
	VelocidadDeseada *= m_pVehiculoDueño->GetVelocidadMax();

	return (VelocidadDeseada - m_pVehiculoDueño->GetVelocidadLineal());
}
```

La implementación difiere en dos cosas de Buscar. Primero verifica que la distancia a la posición del peligro, de la cual queremos huir, no sea mayor a un cierto valor pre-seteado (en el código se eligió 10000 de forma arbitraria, podría ser también algún valor calculado). Si la distancia en mayor a esta devuelve fuerza cero, es decir que no huye. En cambio si la distancia es menor calcula la fuerza de huida, el calculo es similar al de Buscar solo que la variable VelocidadDeseada se calcula apuntando en la dirección opuesta, ya que en la resta los elementos están intercambiados de lugar. En la Ilustración 4 podemos ver estos cálculos de forma grafica.

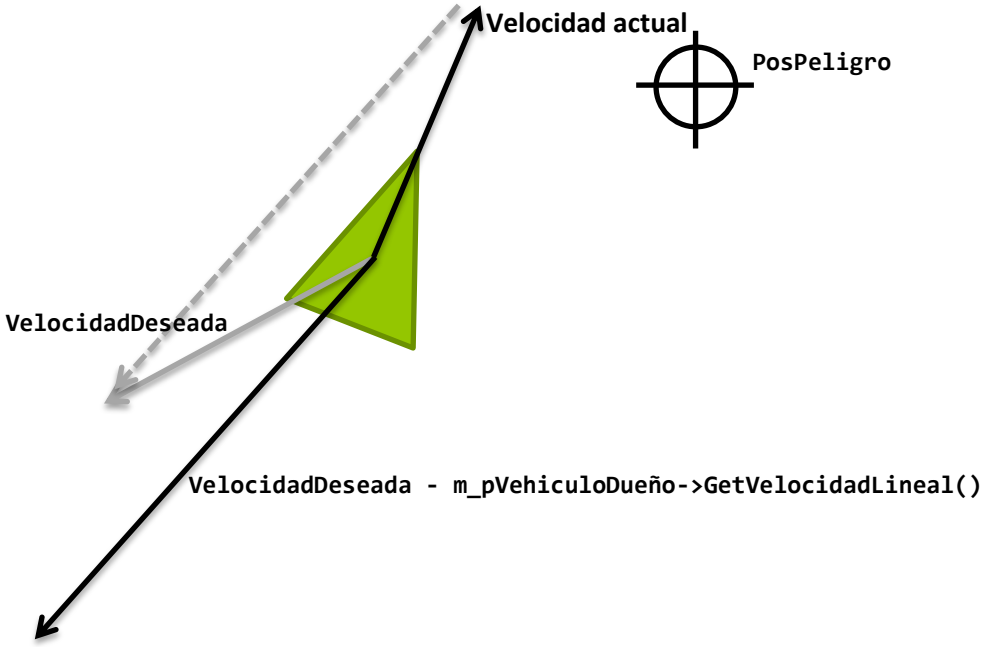

ILUSTRACIÓN 4: CALCULO DE COMPORTAMIENTO ESCAPAR

De esta forma en un video podríamos usar los dos comportamientos que ya vimos para simular el comportamiento de un conejo y zorro. El zorro si encuentra al conejo a la vista activaría su comportamiento de Buscar hacia la posición del conejo. Mientras que el conejo al alertarse de la presencia del zorro activaría su comportamiento de Escapar de la posición del zorro. De esta forma podemos lograr que estas dos entidades se comporten de forma orgánica de forma fácil. Incluso es muy sencillo integrar estos comportamientos a una FSM, la cual podría activar y desactivar los comportamientos según sea oportuno.

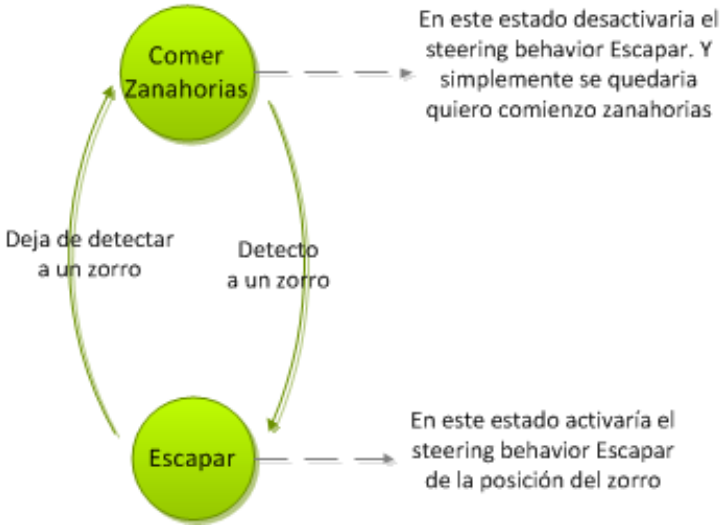

ILUSTRACIÓN 5: EJEMPLO DE UNA FSM USANDO STEERING BEHAVIORS

Arribar

Imaginemos que estamos programando las unidades de un juego de estrategia en tiempo real (por ejemplo algo similar al Starcraft), si seleccionamos una unidad y le decimos que se mueva a una determinada posición podríamos pensar que podríamos usar el comportamiento Buscar para esto, pero en general queremos que cuando la unidad llegue a la posición que le indicamos se detenga en esta pero el comportamiento Buscar se pasaría del punto ya que no pasaría "desacelerando" sino que pasaría "por el punto a toda marcha". Para lograr este comportamiento implementaremos un nuevo comportamiento que llamaremos Arribar. Él comportamiento tiene mucha similaridad con el de Buscar, pero en vez de intentar que el vehículo avance a la posición a la máxima velocidad que éste puede alcanzar, calculara la velocidad con la cual acercarse en función de la distancia a la que se encuentra el vehículo. De esta forma si el vehículo esta muy lejos se acercara a mucha velocidad, tienen como limite la velocidad máxima del vehículo, pero a medida que se acerca acortando la distancia al punto la velocidad ira disminuyendo hasta que en el punto la velocidad cera cero. Además el comportamiento recibirá un nuevo parámetro que le dirá que tan rápidamente ir desacelerando, este parámetro dirá el tipo de desaceleración y podrá tener los valores:

```
        enum Desaceleracion{ Lento  = 3,// desacelera lentamente
                             Normal = 2,// desacelera normalmente
                             Rapido = 1 // desacelera rapidamente
                            };
```

Veamos la implementación del comportamiento:

```
sf::Vector2f SteeringBehaviors::Arribar(const sf::Vector2f& PosObjetivo,
                                        Desaceleracion Tipo)
{
       sf::Vector2f HaciaObjetivo =
                     PosObjetivo - m_pVehiculoDueño->GetPosicion();

       float dist = Norma(HaciaObjetivo); //distancia al objetivo
       if( dist > 0.0f )//si todavia no estamos sobre el objetivo
       {
              //es necesaria para utilizar la desaceleracion
              const float CoefDesaceleracion = 0.3;
              //calculamos la velocidad para llegar al
              float velocidad = dist / ((float)Tipo * CoefDesaceleracion);
              //nos aseguramos que la velocidad no excede el limite del vehiculo
              velocidad =
                     std::min(velocidad, m_pVehiculoDueño->GetVelocidadMax());

              // (HaciaObjetivo/dist) es solo HaciaObjetivo normalizado
              sf::Vector2f VelocidadDeseada = (HaciaObjetivo/dist) * velocidad ;

              return
                 (VelocidadDeseada - m_pVehiculoDueño->GetVelocidadLineal());
       }
       return sf::Vector2f(0.0f, 0.0f);
}
```

La clave del comportamiento esta en ver que si la distancia al objetivo es muy grande entonces el calculo std::min(velocidad, m_pVehiculoDueño->GetVelocidadMax()) devolverá m_pVehiculoDueño->GetVelocidadMax(), siendo el comportamiento idéntico al de Buscar. Por otro lado si la distancia al objetivo no es tan grande luego el calculo devolverá velocidad, que es menor que la velocidad máxima del vehículo y que cada vez que el vehículo se acerque más al objetivo será menor. Ese comportamiento de la variable velocidad de achicarse cuando el vehículo se acerca al objetivo es debido a que se calcula como dist / ((float)Tipo * CoefDesaceleracion), es decir que la velocidad es directamente proporcional a la distancia entre el vehículo y el objetivo, si la distancia se acerca la velocidad se achicara, si la distancia se aleja la velocidad se agrandara. La división por ((float)Tipo * CoefDesaceleracion) solo tiene como propósito poder controlar con el parámetro Tipo que tan rápido desacelera el vehículo.

Interceptar

Estamos llegando a comportamientos más interesantes. Supongamos que siguiendo el ejemplo que planteamos antes del conejo y el zorro, si usamos el comportamiento de Buscar en el zorro para que persiga al conejo este lo seguirá, pero no lo hará de una forma muy inteligente! Imaginémonos que estamos cazando patos, algo al estilo del clásico Duck Hunt de NES pero llevado a los tiempos modernos, si queremos dispararle a un pato no le dispararemos a la posición que se encuentra en este preciso momento, sino que viendo que el pato se mueve a una determinada velocidad apuntaremos mas adelante en la dirección en la que se mueve le pato, de

esta forma considerando el tiempo que demorar llegar la bala hasta al posición del pato, éste en ese momento se encontrara en la posición de la bala. De no hacer este cálculo jamás derribaríamos un pato a menos que este se encuentre quieto (levitando en el cielo… cosa muy poco probable a menos que ese pato sea magneto). La misma lógica se aplica al zorro, si el corre siempre a la posición en la que se encuentra el conejo, le seria muy difícil alcanzarlo además de que el personaje se vería tonto (y no seria coincidencia, seria tonto). En cambio si el zorro pudiera predecir en que posición se encontraría el conejo si éste sigue corriendo en la misma dirección, y correría hacia ese punto tendría muchas mas posibilidades de atraparlo.

Claramente que tan bien el persecutor realice su tarea depende principalmente de que tan bien haga la predicción de donde se encontrar su objetivo en el futuro. Se puede tener muchas cosas en cuenta para realizar una predicción más o menos precisa, pero siempre tenemos que recordar que mientras más complejicemos el código seguramente más tiempo computacional requerirá calcularlo.

Lo más difícil de calcular en este comportamiento será cuanto tiempo en el futuro predecir la posición del objetivo. Es decir si el zorro y el conejo se encuentran separados a un metro, no tendría sentido que el zorro calculara en que posición se encontraría el conejo en una hora, y luego se dirija hacia esa posición, sino que como se encuentran cerca el zorro debería predecir la posición del conejo en a un tiempo relativamente corto para poder atraparlo. Por otro lado también hay que tener en cuenta que tan rápidos son los personajes que estamos simulando, ya que para un conejo y un zorro un metro es poca distancia, pero por ejemplo para un caracol es mucha distancia, por ejemplo en este caso tal vez predecir la posición del otro caracol en una hora seria lo correcto. De ese análisis podemos ver que el tiempo correcto en el que calcular la posición de la presa parece estar relacionado directamente con la distancia entre el persecutor y la presa, es decir mientras mas cerca predecir a menos tiempo y mientras más lejos predecir a mas tiempo. Además el tiempo correcto también esta relacionado con la velocidad del persecutor y de la presa, en este caso inversamente proporcional, si el persecutor y la presa se mueven muy rápido entonces debemos realizar la predicción a poco tiempo, mientras que si el persecutor y la presa son muy lento deberemos realizar la predicción a más tiempo. Por todo esto que analizamos vemos que al parecer una buena alternativa para saber a cuanto tiempo predecir la posición del objetivo puede calcularse como[3]:

$$dt_{\text{predicción}} = \frac{distancia_{persecutor\ y\ presa}}{velocidad_{persecutor} + velocidad_{presa}}$$

Además debemos darnos cuenta de un caso particular. Supongamos que el zorro y el conejo están corriendo hacia el mismo lugar (digamos que seria un conejo muy valiente), de forma que en el futuro se chocaran. En ese caso no debemos predecir en que posición estará el conejo en el futuro, sino que corriendo directamente hacia él (es decir con el comportamiento de Buscar) lo atraparemos. Para saber cuando el persecutor y la presa están corriendo en la misma dirección

[3] Si nuestro modelo de Locomoción (Vehículo) tarda tiempo en girar, por ejemplo un tanque, podríamos tenerlo en cuenta en el cálculo del tiempo de predicción.

deberemos definir cuando consideramos que tienen la misma dirección, por ejemplo podríamos decir que cuando la dirección del persecutor y de la presa se encuentran entre cierto rango de ángulos entonces no hacer el calculo de predicción, sino que directamente llamar al comportamiento Buscar. Podemos ver este caso en la Ilustración 6.

ILUSTRACIÓN 6: ANÁLISIS DE VEHÍCULOS ENFRENTADOS. LAS LÍNEAS DE PUNTOS NEGRAS REPRESENTAN LAS DIRECIONES DE LOS VEHÍCULOS

Veamos la implementación del comportamiento:

```
sf::Vector2f SteeringBehaviors::Interceptar(const Vehiculo& Acechado)
{
	assert(&Acechado != NULL);//Verificamos por que sea un objeto valido

	sf::Vector2f haciaAcechado =
		Acechado.GetPosicion() - m_pVehiculoDueño->GetPosicion();
	float angDif =
		Dot(m_pVehiculoDueño->GetDireccion(), Acechado.GetDireccion());

	angDif = (180/PI)*acos(angDif);
	//Si estan mirandose de frente no predecimos sino que solo avanzamos
hacia donde esta ahora
	if( Dot(haciaAcechado, m_pVehiculoDueño->GetDireccion() ) > 0.0f &&
		angDif > (180-TOLERANCIA_MIRANDOSE_GRADOS) &&
		angDif < (180+TOLERANCIA_MIRANDOSE_GRADOS) )
		return Buscar(Acechado.GetPosicion());//Buscar directamente

	//calculo de dt_prediccion = ditancia / sumaVelocidad
	float dtPrediccion = Norma(haciaAcechado) /
   (m_pVehiculoDueño->GetVelocidadMax()+Norma(Acechado.GetVelocidadLineal()));

	//en funcion de dt_prediccion calculamos en donde estara la presa
	sf::Vector2f posAcechadoPrediccion =
	Acechado.GetPosicion() + Acechado.GetVelocidadLineal()*dtPrediccion;

	return Buscar(posAcechadoPrediccion);//Buscar hacia la posicion predicha
}
```

Veamos antes que nada que en el caso de Interceptar no pasaremos como parámetro una posición, sino que le pasaremos un objeto de tipo Vehiculo ya que necesitamos obtener la velocidad a la que se encuentra moviéndose el personaje y en la dirección en la que se mueve, a diferencia de Buscar, Escapar y Arribar que solo necesitábamos una posición. Podemos ver en Ilustración 7 el cálculo del comportamiento.

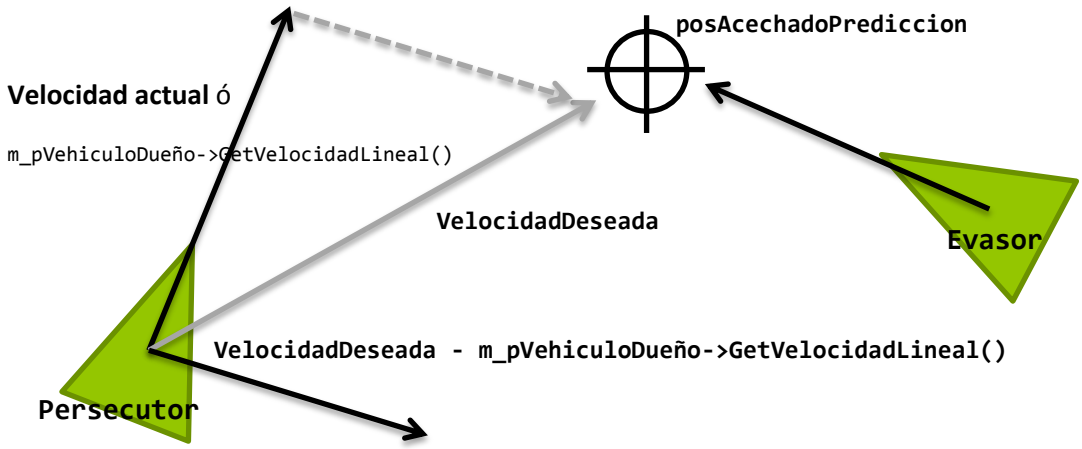

ILUSTRACIÓN 7: CALCULO DE COMPORTAMIENTO INTERCEPTAR

Evadir

El comportamiento de evadir es exactamente el opuesto a interceptar. El evasor predice a donde estará el persecutor en un momento del futuro y escapa de esa posición. La implementación es:

```
sf::Vector2f SteeringBehaviors::Evadir(const Vehiculo& Acechador)
{
	assert(&Acechador != NULL);//Verificamos por que sea un objeto valido

	sf::Vector2f haciaAsechador =
		Acechador.GetPosicion() - m_pVehiculoDueño->GetPosicion();

	float dtPrediccion = Norma(haciaAsechador) /
  (m_pVehiculoDueño->GetVelocidadMax()+Norma(Acechador.GetVelocidadLineal()));

	sf::Vector2f posAsechadorPrediccion =
	Acechador.GetPosicion() + Acechador.GetVelocidadLineal()*dtPrediccion;

	return Escapar(posAsechadorPrediccion);
}
```

Como vemos la implementación es casi idéntica a Interceptar, solo que en este caso no hace falta verificar el caso de que el evasor y el persecutor estén enfrentados y en vez de utilizar el comportamiento Buscar sobre la posición predicha utilizamos el comportamiento Escapar.

Deambular

Un comportamiento muy útil en un video juego es uno que haga que el vehículo vague por el escenario de forma orgánica, por orgánica nos referimos a que no cambie de dirección de forma "nerviosa" sino que lo haga de forma suave. Eso es exactamente lo que hace este comportamiento. Por ejemplo podríamos usarlo en el ejemplo del zorro y el conejo, activando el comportamiento de Deambular en el zorro mientras no tenga el conejo a la vista, de esta forma el zorro se movería por el escenario de forma natural hasta que al ver al conejo cambiaria a un comportamiento de Buscar o de Interceptar para atrapar al conejo.

Si bien para implementar un comportamiento que deambule podríamos simplemente devolver una fuerza aleatoria, esto provocaría lo que ya dijimos que queríamos evitar, que el vehículo cambie de dirección de forma abrupta. Para evitar ese cambio abrupto utilizaremos un método relativamente barato computacionalmente hablando y que además da un buen resultado. Él mismo consiste en utilizar un círculo de proyección que estará adelante del vehículo, y donde la posición a la cual se moverá el vehículo siempre estará en el perímetro de ese círculo. Cada ejecución de Deambular le agrega un desplazamiento aleatorio al objetivo, el cual luego se lo proyectara de tal forma que el objetivo siempre quede sobre el perímetro del círculo. Veamos en la Ilustración 8, antes de implementarlo, los pasos que realizaremos:

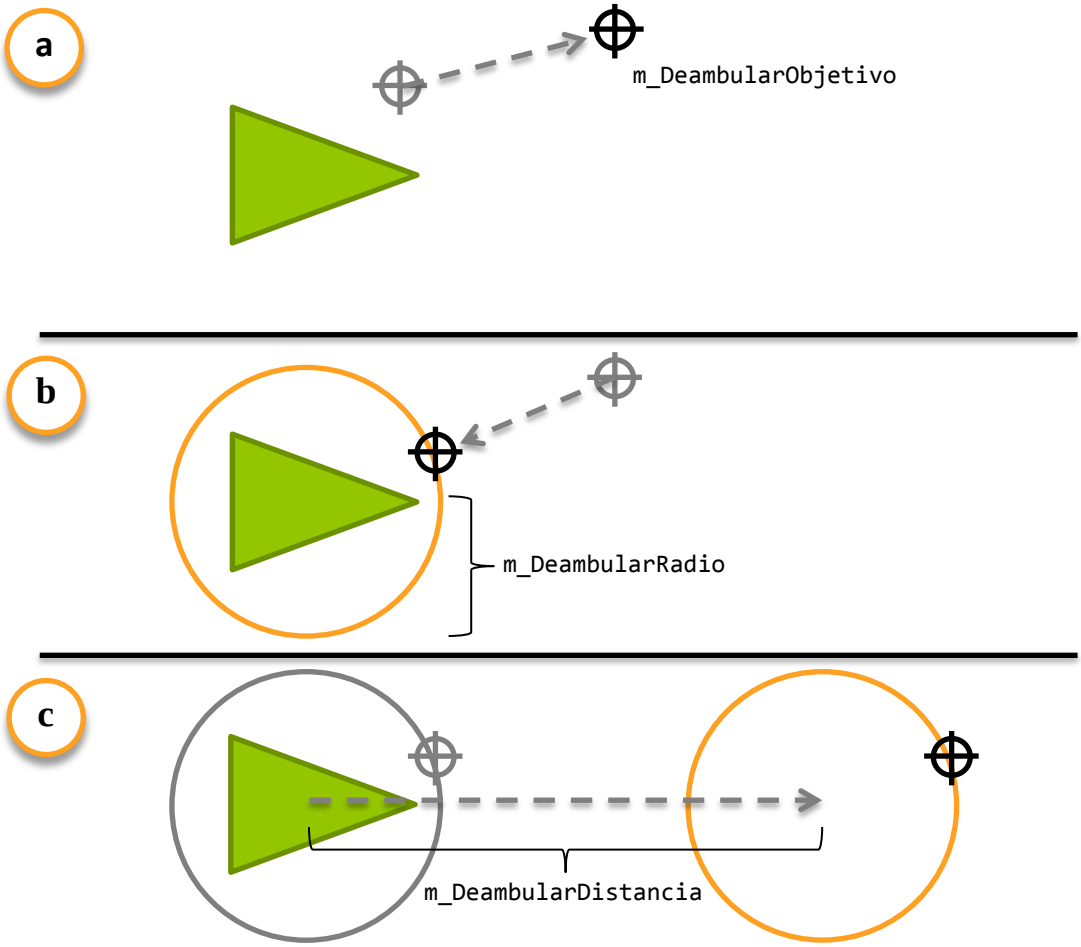

ILUSTRACIÓN 8: ETAPAS DEL CÁLCULO DE DEAMBULAR

En el grafico podemos ver que la mira negra representa la posición objetivo a desplazarse, mientras que la mira en gris representa el valor antiguo de la posición objetivo a desplazarse antes de que se produzca un cambio en ésta. Por esto podemos ver por ejemplo que en la etapa b, la mira pasa de estar en donde estaba (en color negro) en la etapa a, a donde queda luego de realizar la acción que corresponde a la etapa b. Una línea interrumpida en color gris representa un desplazamiento, que puede ser de la posición objetivo si es que sale de esta o del círculo de proyección. El método requerirá que definamos los valores de ciertas variables antes de poder ejecutarlo:

- m_DeambularRuido: El valor máximo de desplazamiento que tendrá m_DeambularObjetivo en cada ejecución.
- m_DeambularRadio: El radio del círculo en el cual proyectaremos el objetivo.
- m_DeambularDistancia: La distancia adelante del vehículo que desplazaremos el círculo de proyección.

Los valores que asignemos a estos parámetros modificaran como cambiara de dirección el personaje mientras deambule. Al final analizaremos como afectaran. Veamos viendo como implementamos las etapas que vimos en el grafico:

a. Desplazamos la posición objetivo un valor aleatorio. `RandUnitario`[4] `es una función propia que devuelve un valor flotante aleatorio entre -1 y 1.`

```
//movemos el objetivo de deambular
m_DeambularObjetivo += sf::Vector2f(RandUnitario()*m_DeambularRuido,
                                    RandUnitario()*m_DeambularRuido);
```

b. Proyectamos la posición objetivo sobre el perímetro del círculo que esta alrededor del vehículo.

```
//proyectamos el objetivo sobre el circulo al rededor del vehiculo
m_DeambularObjetivo = m_DeambularObjetivo/Norma(m_DeambularObjetivo);
m_DeambularObjetivo *= m_DeambularRadio;
```

c. Desplazamos el circulo (en realidad lo que movemos es solo el objetivo) hacia adelante del vehículo.

```
//movemos el circulo(y el punto) adelante de vehiculo
sf::Vector2f m_DeambularLocal = sf::Vector2f(m_DeambularDistancia, 0.0f) +
                                m_DeambularObjetivo;
```

Una vez realizados estos tres pasos ya tenemos nuestro punto hacia donde queremos desplazarnos, solo haría falta devolver la distancia entre ese punto y la posición del vehículo como hemos hecho en otros comportamientos y listo… Pero un momento! Queda un detallecito más. Si nos ponemos a pensar cuando movimos el círculo hacia adelante en realidad lo único que hicimos fue desplazarlo en la dirección del eje x, además supusimos que la posición $(0,0)$ era el centro del vehículo. Eso se debe a que estábamos trabajando en el espacio local del vehículo[5], es decir donde la posición $(0,0)$ es el centro del vehículo, y el eje x es positivo en la dirección en la que mira el vehículo. Pero no desesperemos esto no es ningún problema, sino que es una gran comodidad, si no hubiésemos trabajado de esa forma los cálculos de los pasos a, b y c hubiesen sido muchísimo mas complicados, lo que debemos hacer es convertir el objetivo, m_DeambularLocal, al espacio global, donde la posición $(0,0)$ se transformara a la posición donde esta el vehículo en la escena (Mundo), el vector x se transformara al vector que apunta en la dirección en la que mira el vehículo, y así el resto de los valores. Para eso solo necesitamos usar una matriz de transformación que nos transforme del espacio local al vehículo hasta el espacio de la escena o mundo. No teman! Estamos usando Box2D y podemos usar su funcionalidad para obtener la transformación que queremos de forma muy simple en tan solo una línea, e incluso usar la transformación también usando un método en una línea más (pufff menos mal). En fin el código que realizara esta pequeña magia será:

```
//transformamos el punto desde espacio local al espacio del mundo
b2Transform T = m_pVehiculoDueño->GetTransformacion();
b2Vec2 ObjetivoMundo = b2Mul(T, b2Vec2(m_DeambularLocal.x,m_DeambularLocal.y));
```

Ta Da! Ahora la variable ObjetivoMundo tiene la posición a la que queremos desplazarnos en el espacio del mundo. Ahora si solo nos queda devolver la resta entre ese punto y la posición del

[4] La definición completa de RandUnitario es: `return (rand()/(float)RAND_MAX)*2.0f-1.0f`

[5] Recordaran algo de matrices y espacios de Manipulación de Objetos en 2D

vehículo, para que esta fuerza de Deambular nos lleve a ese punto. Recapitulando todo lo que hicimos fue:

```
sf::Vector2f SteeringBehaviors::Deambular()
{
	//movemos el objetivo de deambular
	m_DeambularObjetivo += sf::Vector2f(RandUnitario()*m_DeambularRuido,
					RandUnitario()*m_DeambularRuido);

	//proyectamos el objetivo sobre el circulo al rededor del vehiculo
	m_DeambularObjetivo = m_DeambularObjetivo/Norma(m_DeambularObjetivo);
	m_DeambularObjetivo *= m_DeambularRadio;

	//movemos el circulo(y el punto) adelante de vehiculo
	sf::Vector2f m_DeambularLocal =
		sf::Vector2f(m_DeambularDistancia, 0.0f) + m_DeambularObjetivo;

	//transformamos el punto desde espacio local al espacio del mundo
	b2Transform T = m_pVehiculoDueño->GetTransformacion();
	b2Vec2 ObjetivoMundo =
		b2Mul(T, b2Vec2(m_DeambularLocal.x,m_DeambularLocal.y));

	return
 sf::Vector2f(ObjetivoMundo.x,ObjetivoMundo.y)-m_pVehiculoDueño->GetPosicion();
}
```

La implementación de este comportamiento no fue tan fácil como los anteriores, pero cuando vean a sus personajes deambulando por el escenario de forma orgánica se van a sentir satisfechos.

En cuanto a los parámetros del método, podemos observar que si la distancia del circulo m_DeambularDistancia es muy grande los cambios en el ángulo del vehículo serán chicos. Además si el nivel de desplazamiento aleatorio m_DeambularRuido es grande los cambios de posición del objetivo en el perímetro variaran más, y si es chico variaran menos. En cuanto al radio del circulo, m_DeambularRadio, hará que los giros del vehículo sean más o menos pronunciados.

Evadir Obstáculos

Este comportamiento le permite al vehículo evadir obstáculos que estén en el escenario. Los obstáculos que el vehículo es capaz de evadir son obstáculos circulares. Obviamente en la escena estos podrían graficarse como cualquier cosa, ya sea un árbol, una roca, un personaje que se mueve por el escenario, etc pero la figura de colisión que se utilizara para evitar colisionar son círculos. Que este comportamiento este activo no quiere decir que no pueda darse una colisión, sino que lo que quiere decir es que cuando el objeto se mueva cerca de ese obstáculo, si esta moviéndose en una dirección que producirá un choque, el vehículo intentara evitar chocar girando su dirección y frenándose. Es decir que la fuerza que se generara tendrá dos componentes, una de giro y otra de freno como podemos ver en la Ilustración 9.

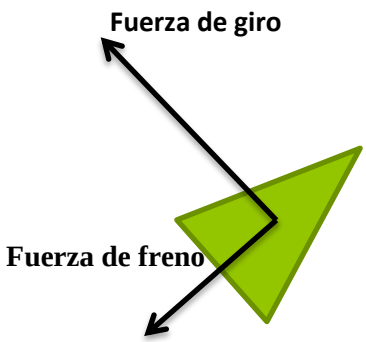

ILUSTRACIÓN 9: ESQUEMA DE FUERZAS DE COMPORTAMIENTO EVADIROBSTACULOS

Como la implementación de este comportamiento tiene algunos detalles complejos como: colisiones entre línea y círculo y trabajo en espacio local y transformación a espacio global. No se vera la implementación en el apunte… claro que se invita a algún valiente a ver la implementación en el código.

Evadir Paredes

Este comportamiento, con un fin similar al anterior, le permite al vehículo evitar colisionar con las paredes que estén en el escenario. Es decir que en vez de evitar colisionar con obstáculos circulares como en el comportamiento anterior, evitaremos colisionar con un segmento que representara una pared. La idea de este comportamiento es usar unos segmentos invisibles que salen por delante del vehículo y probar si estos colisionan con alguna pared, si lo hacen se aplica una fuerza en la dirección de la normal de la pared al vehículo… la cual esperamos que frene al vehículo y lo aleje de la pared. La idea de esos sensores es muy similar a como puede moverse un gato en la oscuridad con sus bigotes, si siente con ellos que tiene algo adelante probara en otra dirección, si no siente que sus bigotes tocan algo entonces puede moverse hacia adelante.

ILUSTRACIÓN 10: ESQUEMA DE EVADIRPAREDES. LA LÍNEA AZUL OSCURA REPRESENTA UNA PARED. LOS SEGMENTOS NEGROS QUE SALEN DEL VEHÍCULO SON SENSORES QUE DETECTAN SI ADELANTE DEL VEHÍCULO HAY UNA COLISIÓN. EN UN CIRCULO ROJO SE MARCO UNA COLISIÓN CON LA PARED

Al igual que el comportamiento EvadirObstaculos no mostraremos la implementación de este comportamiento en este apunte, pero su implementación puede verse en el código que se provee.

La clase Steering Behaviors

Llego el momento de ver como esta implementada la clase SteeringBehaviors que provee la interfaz para activar y desactivar comportamientos así como la de calcular la fuerza resultante de los comportamientos activos, mezclando las fuerzas. Analicemos primero las propiedades y métodos privados de la clase:

```
class SteeringBehaviors
{
private:

	enum { NadaBit             = 0x00000,
	       BuscarBit           = 0x00002,
	       EscaparBit          = 0x00004,
	       ArribarBit          = 0x00008,
	       InterceptarBit      = 0x00010,
	       EvadirBit           = 0x00020,
	       DeambularBit        = 0x00040,
	       EvadirObstaculosBit = 0x00080,
	       EvadirParedesBit    = 0x00100
	       };

	enum Desaceleracion{ Lento  = 3,
	                     Normal = 2,
	                     Rapido = 1
	                    };

	int m_ComportamientosActivos;

	Vehiculo* m_pVehiculoDueño;

	sf::Vector2f m_FuerzaComportamiento;

	sf::Vector2f Buscar(const sf::Vector2f& PosObjetivo);

	sf::Vector2f Escapar(const sf::Vector2f& PosPeligro);

	sf::Vector2f Arribar(const sf::Vector2f& PosObjetivo,
	                     Desaceleracion Tipo);

	sf::Vector2f Interceptar(const Vehiculo& Acechado);

	sf::Vector2f Evadir(const Vehiculo& Acechador);

	sf::Vector2f Deambular();

	sf::Vector2f EvadirObstaculos(std::vector<EntidadEscena*> &Obstaculos);
….
```

La propiedad m_pVehiculoDueño tiene un puntero al objeto vehículo que contiene estos comportamientos, todo vehículo tendrá su objeto SteeringBehaviors y a su vez el objeto

`SteeringBehaviors` debe saber cual es su vehículo dueño (esto se pasara en el constructor de la clase `SteeringBehaviors`). De esta forma podemos acceder a propiedades del vehículo como lo hacemos desde los comportamientos, como por ejemplo obtener la velocidad a la cual se esta moviendo el vehículo, obtener su posición, etc. Los métodos Buscar, Escapar, Arribar, Interceptar, Evadir, Deambular, EvadirObstaculos y EvadirParedes son las funciones que ya hemos visto, las cuales calculan la fuerza resultante de ese comportamiento. Son métodos privados ya que solo los usara la clase para obtener las fuerzas de los comportamientos, pero jamás el usuario ejecutara directamente uno de estas funciones, en vez de eso el usuario de la clase activara o desactivara un comportamiento haciendo uso de una interfaz que veremos a continuación. La parte más interesante de esta sección privada de la clase es la forma en la que se activan y se desactivan los comportamientos. Podríamos tener una variable bool por cada comportamiento, que nos diga si ese comportamiento esta activo o desactivo (por ej. si es true estaría activo y si es false estaría desactivo), pero porque usar eso si podemos hacerlo mas eficiente y más elegante. En vez de tener esa cantidad de variables, una por cada comportamiento, solo tendremos una sola variable de tipo entero: m_ComportamientosActivos. Un valor de tipo entero esta compuesto por 32 bits, los cuales pueden tener dos valores que son 0 o 1. Entonces podríamos almacenar en cada uno de estos bits si un comportamiento esta activo o desactivo, si el bit esta en 1 luego el comportamiento correspondiente a ese bit estará activo, mientras que si el bit esta en 0 luego el comportamiento correspondiente a ese bit estará desactivo. Entonces veamos un ejemplo, nosotros relacionamos el comportamiento Buscar con el primer bit del valor, entonces imaginemos que el estado actual de la variable m_ComportamientosActivos es el siguiente:

$$0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,1\,0\,1\;binario$$

Luego podemos saber si el comportamiento Buscar esta activo fijándonos en el primer bit, es decir el que esta más a la derecha:

$$0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,1\,0\,1\;binario$$

El primer bit es 1, por lo tanto el estado Buscar esta activo

Por otro lado si el valor de m_ComportamientosActivos hubiese sido:

$$0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,1\,0\,0\;binario$$

Entonces podríamos observar que:

$$0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,1\,0\,0\;binario$$

El primer bit es 0, por lo tanto el estado Buscar esta desactivo

Vemos entonces que es posible almacenar el estado de activado o desactivado de todos los estados en esa solo variable entera (tipo int) de forma muy elegante y eficiente. Lo único que queda es asociar a cada bit un comportamiento, de esa forma sabremos que el primer bit corresponde al comportamiento Buscar, el segundo bit corresponde al comportamiento Escapar, etc. El enum del comienzo de la clase hace exactamente eso:

```
enum { NadaBit             = 0x00000,
       BuscarBit           = 0x00002,
       EscaparBit          = 0x00004,
       ArribarBit          = 0x00008,
       InterceptarBit      = 0x00010,
       EvadirBit            = 0x00020,
       DeambularBit        = 0x00040,
       EvadirObstaculosBit = 0x00080,
       EvadirParedesBit    = 0x00100
       };
```

Lo que hace es asignar a cada comportamiento un valor numérico que tiene en 1 solo el bit que corresponde a ese comportamiento. Por ejemplo ArribarBit es un valor que solo tiene en 1 el bit que le asignamos a el comportamiento Arribar.

¿Como activamos y desactivamos comportamientos de un vehículo?

Para activar o desactivar comportamientos, solo hace falta llamar algún método de la clase SteeringBehaviors. Estos métodos son:

```
        void BuscarOn();
        void BuscarOff();

        void EscaparOn();
        void EscaparOff();

        void ArribarOn();
        void ArribarOff();

        void InterceptarOn();
        void InterceptarOff();

        void EvadirOn();
        void EvadirOff();

        void DeambularOn();
        void DeambularOff();

        void EvadirObstaculosOn();
        void EvadirObstaculosOff();

        void EvadirParedesOn();
        void EvadirParedesOff();

        void EsconderseOn();
        void EsconderseOff();
```

Por ejemplo si tenemos un objeto de tipo Vehiculo podriamos activarle un comportamiento de control de la siguiente forma:

```
Vehiculo AutoFantastico(<varios parametros aburridos>);

//Activamos los comportamientos de Deambular y de EvadirObstaculos
AutoFantastico.GetSteeringBehaviors().DeambularOn();
AutoFantastico.GetSteeringBehaviors().EvadirObstaculosOn();

//Desactivamos el comportamiento de Interceptar
AutoFantastico.GetSteeringBehaviors().InterceptarOff();
```

¿Cómo se mezclan las fuerzas de los comportamientos?

Para mezclar las fuerzas se puede usar varios métodos:

a. ***Suma Truncada***

   Este es la forma mas obvia para mezclar las fuerzas y consiste simplemente en sumar todas las fuerzas de los comportamientos que se encuentren activos y truncar el valor a la máxima fuerza que puede producir el vehículo. La implementación es:

```
sf::Vector2f SteeringBehaviors::CalcularSumaTruncada()
{
	//reseteamos la fuerza
	m_FuerzaComportamiento = sf::Vector2f(0.0f, 0.0f);

	//Sumamos las fuerzas de cada comportamiento activo
	if( isBuscarOn() )
		m_FuerzaComportamiento +=
				Buscar(m_PosObjetivo);

	if( isEscaparOn() )
		m_FuerzaComportamiento +=
				Escapar(m_PosObjetivo);

	if( isArribarOn() )
		m_FuerzaComportamiento +=
				Arribar(m_PosObjetivo, Desaceleracion::Lento);

	if( isInterceptarOn() )
		m_FuerzaComportamiento +=
				Interceptar(*m_pPresa);

	if( isDeambularOn() )
		m_FuerzaComportamiento +=
				Deambular();

	if( isEvadirObstaculosOn() )
		m_FuerzaComportamiento +=
				EvadirObstaculos(*m_pObstaculos);

	if( isEvadirParedesOn() )
		m_FuerzaComportamiento +=
				EvadirParedes(*m_pParedes);

	return
    Truncar(m_FuerzaComportamiento, m_pVehiculoDueño->GetFuerzaSteeringMax());
}
```

Las funciones con el prefijo “is” y un nombre de comportamiento nos devuelves verdadero si ese comportamiento esta activo, y si están desactivas devuelven false. La función Truncar hace que si la fuerza calculada excede la máxima fuerza del vehículo entonces escala la fuerza para que no exceda la fuerza máxima. Es una buena primera implementación, pero tiene sus problemas. En primer lugar no es muy eficiente, si tenemos muchos comportamientos activos lo mas probable es que suceda que al sumar todas las contribuciones a la fuerza de todos los comportamientos nos excedamos de la fuerza máxima y por ende esta se trunque al valor máximo. Eso significa que todas las fuerzas que se calcularon contribuirán de forma más o menos similar al valor. Pero esto no es siempre algo ideal, imaginemos que tenemos un vehículo que tiene activos los comportamientos de Buscar, Escapar y EvadirObstaculos, además éste esta a punto de colisionar contra un obstáculo. Podemos suponer que la fuerza del comportamiento EvadirObstaculos será muy grande ya que la colisión entre el vehículo y el obstáculo es

inminente, lo mas probable será un valor muy cercano a la fuerza máxima del vehículo o incluso mayor que esta (que luego seria truncada). Podríamos pensar que lo idea seria que evitar la colisionar fuera prioridad, entonces como el valor de EvadirObstaculos esta cerca del máximo lo mejor seria no calcular los comportamientos de Buscar y Escapar en este momento, ya que si lo hacemos luego truncaría la fuerza la fuerza a aplicar en el vehículo dándole igual importancia a la fuerza de EvadirObstaculos como a las de Buscar y Escapar. Por esto que explicamos este método de mezclar las fuerzas no es para nada ideal.

b. Suma Truncada Priorizada

Este método para mezclar las fuerzas es el que se usa por defecto. El método intenta solucionar el problema que explicamos en el método anterior. Es decir que agrega las fuerzas de los comportamientos que se consideran más importantes y si se excede en magnitud omite los comportamientos considerados menos importantes, por eso llamamos al método priorizado. Para realizar esto necesitaremos una función auxiliar que utilizaremos para agregar fuerzas al total a medida que calculamos las contribuciones de cada comportamiento. Esta función tendrá el siguiente comportamiento:

- Si la fuerza resultante de agregar la nueva fuerza no excede el máximo de fuerza impuesto por el vehículo, luego se agrega la fuerza de forma normal y la función devuelve el valor true indicando que se puede continuar acumulando fuerzas.
- Si la fuerza previa a sumarle la nueva fuerza excede el máximo de fuerza impuesto por el vehículo, luego no se agrega ninguna fuerza y la función devuelve el valor false indicando que ya no se puede continuar acumulando fuerzas.
- Si la fuerza previa a sumarle la nueva es menor a la máxima impuesta por el vehículo, pero sumándole la nueva fuerza excede éste limite, luego agrego la nueva fuerza escalada de forma tal que llegue a la fuerza máxima pero no la exceda y la función devuelve el valor true indicando que se puede continuar acumulando fuerzas.

A principio puede parecer que usa una lógica un tanto compleja, pero si nos ponemos a pensar es muy razonable lo que hace: si puede agregar la fuerza completamente la agrega, si no puede agregar una porción y si nos excedimos del máximo avisa que ya no se pueden seguir agregando mas contribuciones de otros comportamientos ya que llegamos a la fuerza máxima. La implementación de esta función es:

```
bool SteeringBehaviors::AgregarFuerza(const sf::Vector2f &fuerzaAgregar)
{
	//magnitud fuerza actual
	float magFActual  = Norma(m_FuerzaComportamiento);
	//magnitud fuerza agregar
	float magFAgregar = Norma(fuerzaAgregar);

	//Si la magnitud actual ya excede el máximo ni considero agregar fuerza
	if( magFActual >= m_pVehiculoDueño->GetFuerzaSteeringMax() )
		return false;

	//Si agregando la fuerza no excedo el máximo la agrego
	if( (magFActual+magFAgregar) < m_pVehiculoDueño->GetFuerzaSteeringMax() )
		m_FuerzaComportamiento += fuerzaAgregar;
	else
	{
		//Si me excedo del maximo solo agrego una porción de fuerza
		float magDif =
			m_pVehiculoDueño->GetFuerzaSteeringMax() - magFActual;
		m_FuerzaComportamiento += (fuerzaAgregar/magFAgregar)*magDif;
	}
	return true;
}
```

Ahora usando esta (mágica) función como arma podemos resolver el problema que habíamos planteado. Iremos agregando las fuerzas de los comportamientos que estén activos usando esta función, si esta devuelve true podemos seguir evaluando otros comportamientos, si en cambio devuelve false deberemos cortar la evaluación de nuevos comportamientos y devolver la fuerza que tenemos hasta ese momento ya que esa será la definitiva. Esta lógica no solo que soluciona el problema que planteamos sino que también es mucho mas eficiente computacionalmente hablando, ya que si nos excedemos en un comportamiento ya no hace falta seguir evaluando el resto de los comportamientos activos, ahorrándonos muchos ciclos del microprocesador. Entonces la implementación del método para mezclar las fuerzas será:

```
sf::Vector2f SteeringBehaviors::CalcularSumaTruncadaPriorizada()
{
	//reseteamos la fuerza
	m_FuerzaComportamiento = sf::Vector2f(0.0f, 0.0f);

	//Sumamos las fuerzas de cada comportamiento activo
	if( isEvadirParedesOn() )
	     if(!AgregarFuerza(
			EvadirParedes(*m_pParedes)*m_PesoEvadirParedes))
				return m_FuerzaComportamiento;

	if( isEvadirObstaculosOn() )
		if(!AgregarFuerza(
			EvadirObstaculos(*m_pObstaculos)*m_PesoEvadirObstaculos))
				return m_FuerzaComportamiento;

	if( isDeambularOn() )
		if(!AgregarFuerza(
			Deambular()*m_PesoDeambular))
				return m_FuerzaComportamiento;

	if( isBuscarOn() )
		if(!AgregarFuerza(
			Buscar(m_PosObjetivo)*m_PesoBuscar))
				return m_FuerzaComportamiento;

	if( isEscaparOn() )
		if(!AgregarFuerza(
			Escapar(m_PosObjetivo)*m_PesoEscapar))
				return m_FuerzaComportamiento;

	if( isArribarOn() )
		if(!AgregarFuerza(
		     Arribar(m_PosObjetivo, Desaceleracion::Lento)*m_PesoArribar))
				return m_FuerzaComportamiento;

	if( isInterceptarOn() )
		if(!AgregarFuerza(
			Interceptar(*m_pPresa)*m_PesoInterceptar))
				return m_FuerzaComportamiento;

	if( isEvadirOn() )
		if( !AgregarFuerza(
			Evadir(*m_pAcechador)*m_PesoEvadir))
				return m_FuerzaComportamiento;

	return m_FuerzaComportamiento;
}
```

Pero debemos tener cuidado, no es opcional el orden en el cual evaluamos los comportamientos con este método, sino que eso determina que tanto priorizamos ese comportamiento. Es decir en el código primero evaluamos el comportamiento de EvadirParedes, es decir que es al que más importancia le estamos asignando ya que si este devuelve un valor muy grande y excede el máximo no se agregaran los otros comportamientos y el vehículo solo reaccionara usando este comportamiento. De la

misma forma el segundo comportamiento que mas priorizamos es EvadirObstaculos, y así sucesivamente hasta llegar al comportamiento de Evadir que es al que menos importancia le asignamos. Pero comprendamos que cuando decimos que un comportamiento le damos menos importancia no nos referimos a que este no pueda tener una gran contribución a la fuerza, sino que nos referimos a que este se hace en ultima instancia y solo si los comportamientos mas importantes no saturaron la fuerza ya.

Bibliografía

"Programming Game AI by Example", Mat Buckland. Wordware Publishing, 2005.

"Steering Behaviors For Autonomous Characters"*, Craig W. Reynolds.*

"AI for Game Developers", David M. Bourg, Glenn Seeman. O'Reilly,2004.

"Artificial Intelligence: A Modern Approach"*, Stuart Russell, Peter Norvig.*

"The C++ Programming Language", Bjarne Stroustrup. Addison Wesley.