Universidad de las Ciencias Informáticas

Facultad 5 Entornos Virtuales

Comportamientos de autos en pistas de carreras basados en Steering Behaviors para el videojuego "Rápido y Curioso".

Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

Autores: Dalian Sánchez Cruz.

Alexei Herrera Pérez.

Tutor: MsC. Yuniesky Coca Bergolla.

Ciudad de la Habana
Junio - 2008

Declaración de autoría

Declaramos ser autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Dalian Sánchez Cruz	Alexei Herrera Pérez	Yuniesky Coca Bergolla
_____	_____	_____
Firma del Autor	Firma del Autor	Firma del Tutor

"La Ciencia es una tentativa en el sentido de lograr que la caótica diversidad de nuestras experiencias sensoriales corresponda a un sistema de pensamiento lógicamente ordenado".

Albert Einstein.

Agradecimientos

La ciencia es la verdadera escuela moral; ella enseña al hombre el amor y el respeto a la verdad, sin el cual toda esperanza es quimérica. A lo largo de mi vida he comprendido que tener fe en el mejoramiento humano y en la virtud es el único modo de triunfar y alcanzar las metas alcanzadas. Hoy me siento emocionado y agradezco:

A Fidel y a Raúl por creer y luchar por un mundo mejor, sin ellos este sueño no hubiese sido posible,

A la UCI por recibirme como un niño y despedirme hoy como todo un profesional,

A sus profesores por ser ejemplo de profesionalismo y constancia, a Idelkys, Zoraida y muchos otros,

A mis compañeros de estudio, a los integrantes del proyecto "Juego Consola" que fueron un tercero, cuarto y hasta un quinto integrante de este trabajo,

A mis amigos, a aquellos que están e incluso a los que no,

A ti Ine por ser fuente de cariño y acompañarme en estos últimos tiempos como estudiante,

A mi compañero de tesis Alexei por ser compañero de batalla científica,

A nuestro tutor por su dedicación y profesionalismo,

A mi familia que tanto apoyo me brindó, a mi hermano y abuelo del alma,

En especial a mi madre por ser más que eso, por ser guía, confidente, padre, madre, amiga; por ser ángel de la guarda y cuidar de este loco que te ama con todo el corazón.

"La armonía de la vida, el equilibrio necesario para tener paz con uno mismo se debe a la purificación que seamos capases de lograr cada día, ese intento de ser mejores que nos eleva la conciencia y el alma"

A todos, gracias

Dalian

Nada hay tan importante para nosotros los seres humanos, como las relaciones con los demás, "el ser está hecho de relaciones y sólo éstas le importan". La mayor parte de nuestras necesidades requieren de la participación de otras personas y por eso las relaciones humanas son tan complejas.

Si todos quisiéramos ser amados, comprendidos y valorados…..entonces

¿por qué parecemos indiferentes y tacaños a la hora de demostrar el afecto, el aprecio, el agradecimiento y la admiración que nos producen quienes nos rodean?

Despojándome de todo tabú y sabiendo lo difícil que se torna para la conciencia traducir el lenguaje del corazón cuando este tiene tanto que agradecer, la eximo de toda responsabilidad y dejo a este libre en su misión.

Madre, a ti que has sido guardián de todos mis sueños, ángel de mi guarda, veladora fiel e incansable de mi presente y mi futuro, te pido disculpas porque no existen palabras en el lenguaje de los mortales capaces de expresar todo mi agradecimiento.

Padre, por ser mi ejemplo a seguir, por saber transmitirme tu sabiduría y por apoyarme en todo momento. Gracias, gracias de todo corazón.

Al Comandante Fidel por construir una Universidad de futuro y a Raúl por ser fiel continuador de sus conquistas.

A toda mi familia que aún estando lejos siempre he tenido su apoyo.

A mis compañeros de aula, de cuarto y de proyecto, sin los cuales mi tránsito por la UCI no sería inolvidable y este trabajo, resultado del apoyo de ustedes hubiera sido mucho más difícil.

A mis dos amigos Sándor e Ismael quienes más que amigos han sido hermanos para mí.

A ti Annia por acompañarme y ser motivo de inspiración en estos últimos tiempos como estudiante.

A nuestro tutor por su profesionalismo y su confianza en nosotros.

A todos mis profesores, forjadores de un profesional de futuro, en especial a Idelkys, Brígida, Lago, Raúl y Zaida que tanto apoyo y ayuda recibí de ellos.

A mi compañero de tesis que tanto batalló a mi lado para salir victoriosos de esta gesta.

Y por último quisiera agradecer a mi mayor inspiración, mi abuela, ya no estas pero siempre confiaste en mí y se que este era uno de tus mayores anhelos.

A todos mil gracias,

Alexei.

Dedicatoria

A mi familia, a mis tíos, a mis abuelos, a mis hermanos, a la persona más especial del mundo…

A ti madre

Dalian

Dedico este trabajo fruto de estos 23 años de aprendizaje a toda mi familia, a mi tía Mercy que siempre fue un ejemplo en lo que a estudio y preparación se refiere, a Yami que fue la primera graduación de educación superior en la cual pude estar y eso me marcó mucho, a mi tía Pyty que mucho me ayudó en el tiempo de pruebas de ingreso, nunca lo olvidaré, a todos mis tíos, a todos mis primos, a ti Dainier que has sido un hermano y en especial a mi madre que tanto se ha sacrificado por mi, a mi padre por corregirme y guiarme con sabiduría y a mi abuela, a mi querida abuela del alma.

Alexei

Resumen

Una de las técnicas de Inteligencia Artificial más utilizadas para controlar el comportamiento de locomoción de agentes autónomos dentro de entornos virtuales son los *Steering Behaviors*. El control de autos dentro de un videojuego con esta técnica es muy eficiente y se obtienen resultados satisfactorios en muy corto tiempo, por lo que su aplicación en videojuegos de carreras de autos es muy recomendable. En este trabajo se propone un módulo de *Steering Behaviors* el cual está conformado por tres *Behaviors* que juntos logran el comportamiento inteligente de los carros autónomos en un entorno de pistas de carreras, estos son el *Behavior_Angle*, el *Behavior_Brake* y el *Behavior_Accelerator* encargados de calcular los valores de ángulo de giro del timón, posición del pedal de aceleración y fuerza de frenado respectivamente.

Índice de Contenido

Índice de Figuras

Índice de Tablas

Introducción.

El hombre se ha aplicado a sí mismo el nombre científico de homo sapiens como una valoración de la trascendencia de sus habilidades mentales tanto para su vida cotidiana como para su propio sentido de identidad. Los esfuerzos del campo de la Inteligencia Artificial se enfocan precisamente en lograr la comprensión de entidades inteligentes.

Los primeros desarrollos de la Inteligencia Artificial comenzaron a mediados de los años 1950 con la publicación del artículo “Computing Machinery and Intelligence” por el destacado matemático británico Alan Turing **[1]**. En este artículo Turing propuso una prueba concreta para determinar si una máquina era inteligente o no, denominada Test de Turing **[2]**, pero no fue hasta el año 1956, en la famosa conferencia celebrada en el Dartmouth College de Hanover (E.E.U.U) convocada por John McCarthy, que se utilizó por primera vez el término de Inteligencia Artificial (*IA*), nacía entonces esta ciencia, cuyas bases las encontraría en las Matemáticas y la Biología, siendo su principal objetivo el estudio de la inteligencia humana.

Años más tarde con el surgimiento del primer programa interactivo visual, se empezó a pensar en cómo lograr que una computadora simulara la inteligencia lo suficientemente real como para interactuar con un usuario y lograr que este creyera que la computadora realmente pensaba sus acciones **[2]**, nacía así la *IA* para computadores. Luego surgen los primeros videojuegos y simuladores, los cuales requerían de una interacción con el usuario lo más realista posible y en ello la *IA* jugó un papel fundamental.

El éxito de los videojuegos se ha extendido hasta hoy día, sus orígenes se remontan al año 1948, año en el cual fue concebida y patentada la idea de un videojuego **[3]** por Thomas T. Goldsmith Jr. y Estle Ray Mann. Luego en el año 1958 se crea “William Higinbotham's Tennis for Two” y posteriormente en el año 1962 fue desarrollado el videojuego “Spacewar” siendo considerado el primer videojuego computacional. Años más tarde en 1974 sale a la venta el primer videojuego comercial “Computer Space”.

Con el paso de los años los videojuegos se desarrollaron rápidamente y con la nueva revolución 3D la cual constituyó un gran salto técnico, este desarrollo ha sido vertiginoso **[4]**.

La *IA* en los videojuegos ha cobrado auge en los últimos años. Dado que las imágenes desplegadas por las consolas cada vez se acercan más a la realidad, ahora el reto radica en hacer que estas imágenes cobren “vida” y realicen comportamientos que el jugador perciba como inteligentes. Dentro

de este gran mundo, los *Steering Behaviors* (Comportamientos de locomoción) juegan un papel protagónico. Los orígenes de los *Steering Behaviors* datan del año 1986 en el que Craig W.Reynolds, experto en gráficos por computadora y vida artificial, desarrolló el modelo "**Boids**" **[5]**.

El modelo original consiste en simular los movimientos coordinados vistos en manadas de aves y se basa en tres *Behaviors* (Comportamientos) diferentes: Ver figuras # Figura # 1, Figura # 2, Figura # 3.

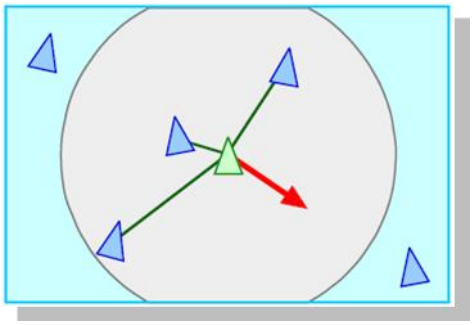


Figura # 1 Behavior Separation.

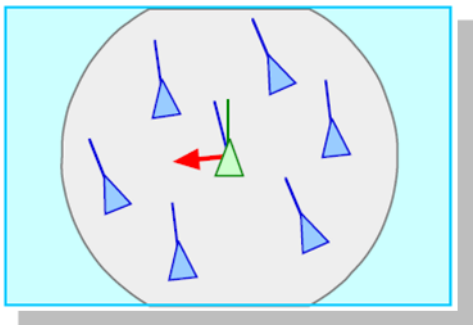


Figura # 2 Behavior Alignment

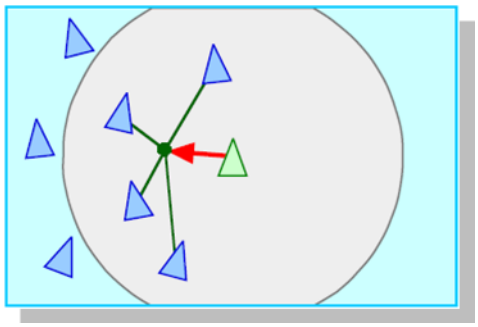


Figura # 3 Behavior Cohesion

En el modelo "**Boids**", Reynolds fue capaz de crear una simulación realista de los comportamientos naturales de un enjambre a través de una hábil combinación de estos tres simples *Behaviors*. Un primer ejemplo de la aplicación de este modelo fue el cortometraje "Stanley and Stella in: Breaking the Ice". (Stanley y Stella en: Rompiendo el hielo) creado por **Symbolics Graphics Division** en una cooperación con **Whitney / Demos Production** en el año 1987 (Ver Figura # 4).

Figura # 4 Escena de 'Stanley and Stella en: Rompiendo el hielo' (1987)

Un año mas tarde en 1988, el propio Reynolds presentó el documento titulado **"Not bumping into things"[1] [6]** en el que expone distintos algoritmos de Evasión de Obstáculos de cada uno de los **"boids"** que se utilizaron en la producción del documental anterior. En este documento describe distintas posibilidades de los algoritmos y las normas para la evasión de obstáculos, además describe la estrategia utilizada, la cual se basó en la utilización de información local, por ejemplo, objetos en el corto radio alrededor del agente.

En el año 1999 en la Conferencia de Desarrolladores de Juegos, Craig W.Reynolds presentó un documento con nuevos avances en el desarrollo de los *Steering Behaviors*, los cuales reforzaron a los

[1] Traducción al español: No chocar con objetos.

presentados en el modelo **“Boids”** en años anteriores. En este documento presenta nuevas piezas para la construcción de complejos sistemas autónomos. Cada uno de estos nuevos Behaviors expuestos solo definen una reacción específica en el entorno simulado del sistema autónomo y dependiendo de la combinación de cada uno de los *Behaviors* del agente puede ser configurado para manejar complejas y diferentes situaciones. Los comportamientos agrupados bajo el nombre de *“Steerings Behaviors”* son solo el nivel más bajo de un sistema autónomo. Además Craig presenta varios ejemplos de la forma en que estos *Behaviors* pueden ser combinados.

En Siggraph 2000, Robin Green presentó un documento titulado *“Steering Behaviors”.* Este trabajo se basó en los comportamientos descritos en el documento de Reynolds **[5]** y pone ejemplos sobre la forma en que estos pueden ser implementados usando el lenguaje C + +, además se discuten varios posibles problemas y sus correspondientes soluciones. Este trabajo se realizó como parte del desarrollo del juego **'Dungeon Master 2'** de la **Bullfrog Productions Ltd**.

La creación de este juego puede ser visto como un éxito en la utilización de los *Steering Behaviors.* Los agentes autónomos en el juego son controlados utilizando simples *Behaviors* y es observable el nivel de realismo que se puede alcanzar con la utilización de los *Steering Behaviors*.

La industria de los videojuegos se encuentra liderada a nivel mundial por los países desarrollados y las grandes y multimillonarias empresas, nuestro país aún estando en vías de desarrollo no se encuentra ajeno a esto, siendo la Universidad de las Ciencias Informáticas (UCI) su pilar fundamental. En ella se encuentra el proyecto Juegos Consola de la Facultad # 5, en el se desarrolla un videojuego de carrera de autos llamado “Rápido y Curioso” en el cual se necesita incorporar carros autónomos que se comporten de forma inteligente (Ver Figura # 5).

Figura # 5 Toma de pantalla del videojuego de carrera de autos "Rápido y Curioso".

De aquí que nuestro **problema científico** sea ¿Cómo simular un comportamiento inteligente en los autos de carrera para el videojuego "Rápido y Curioso"? Siendo el **objeto de estudio** los agentes autónomos (*NPC*) en videojuegos de carreras de autos. El **campo de acción** del trabajo son los *Steering Behaviors* para autos de carrera en videojuegos y nuestro **objetivo general** es definirle los comportamientos basados en *Steering Behaviors* a los autos de carrera del videojuego "Rápido y Curioso". Como **tareas de investigación** se encuentran las siguientes:

- Recopilar información sobre Agentes Autónomos (*NPC*) para definir el comportamiento inteligente de los autos de carrera en videojuegos.
- Analizar las diferentes técnicas y algoritmos de *Steering Behaviors* para la simulación del comportamiento inteligente de autos de carrera en videojuegos.

- Diseñar un módulo para simular el comportamiento inteligente de autos en pistas de carreras en el videojuego.
- Implementar el módulo que permitirá el comportamiento inteligente de los autos de carrera para el videojuego.

Resultados Esperados:

- Obtener un módulo de *Behaviors* capaz de simular el comportamiento inteligente de los autos en pistas de carreras del videojuego “Rápido y Curioso”, siendo flexible a posteriores actualizaciones.
- Obtener un documento que sirva como bibliografía para investigaciones futuras sobre comportamientos inteligentes en entornos virtuales.
- Obtener, siguiendo la metodología RUP, el diseño del sistema a implementar.

Métodos de Investigación:

Métodos teóricos

Analítico, sintético: este método sirve para distinguir las diferentes técnicas y algoritmos relacionados con la generación de *NPC* en Entornos de Realidad Virtual y proceder a realizar una revisión de cada uno de estos por separado, para posteriormente procesar toda la información, poder sintetizar y diferenciar cada uno de estos, además de poder extraer el más adecuado teniendo en cuenta el objeto de estudio.

Histórico, lógico: como está vinculado al conocimiento de las distintas etapas de los objetos en su sucesión cronológica, posibilitará conocer los antecedentes y tendencias de las técnicas de *IA* en Entornos Virtuales en el mundo y en Cuba. Así, con el análisis de su desarrollo histórico lógico podremos llegar a lo más profundo de su esencia.

Métodos empíricos

Observación: se utilizará en la fase de pruebas del módulo, para evaluar el nivel de inteligencia de los *NPCs*, determinando si la simulación tiene la eficiencia y el nivel de realismo requerido o no, y además servirá para encontrar errores de carácter lógico.

Capítulo 1. Fundamentación Teórica

Con sus orígenes por el año 1948 los videojuegos constituyen hoy día uno de los entretenimientos preferidos por las personas, el surgimiento y desarrollo de la *IA* ha jugado un papel fundamental para lograr esta preferencia. Técnicas como los *Steering Behaviors* ideada por Reynolds son muy utilizadas en darle inteligencia a *NPCs* dentro de los videojuegos. En este capítulo se tratarán aspectos, características y soluciones principales de los *Steering Behaviors* así como parte del fundamento matemático que los soporta. Además se abordará sobre los *NPCs* y sobre herramientas, sistemas, lenguajes y metodologías utilizadas en la solución.

1.1 Vectores.

El estudio de los vectores es uno de tantos conocimientos de las matemáticas que provienen de la física. En esta ciencia se distingue entre magnitudes escalares y magnitudes vectoriales. Se llaman magnitudes escalares aquellas en que sólo influye su tamaño. Por el contrario, se consideran magnitudes vectoriales aquellas en las que, de alguna manera, influyen la dirección y el sentido en que se aplican **[7].**

Como ejemplos de magnitudes escalares se pueden citar la masa de un cuerpo, la temperatura y el volumen. Cuando se plantea un movimiento no basta con decir cuánto se ha desplazado el móvil, sino que es preciso decir también en qué dirección y sentido ha tenido lugar el movimiento. No son los mismos los efectos de un movimiento de 90 km a partir de un punto si se hace hacia el norte o si se hace en dirección sudoeste, ya que se llegaría a distinto lugar **[7].**

Aunque el estudio matemático de los vectores tardó mucho en hacerse formalmente, en la actualidad tiene un gran interés, sobre todo para quienes trabajan con los gráficos por computadoras.

Definición de Vectores.

En matemáticas un vector es:

Cantidad que tiene magnitud, dirección y sentido al mismo tiempo.

Por ejemplo, si una cantidad ordinaria o *escalar,* puede ser una distancia de 6 km, una cantidad vectorial sería decir 6 km norte. Los vectores se representan normalmente como segmentos rectilíneos

orientados, como *B* en el diagrama que se muestra a continuación; el punto *O* es el origen o punto de aplicación del vector y *B* su extremo. La longitud del segmento es la medida o módulo de la cantidad vectorial, y su dirección es la misma que la del vector (Ver Figura # 6) **[7]**.

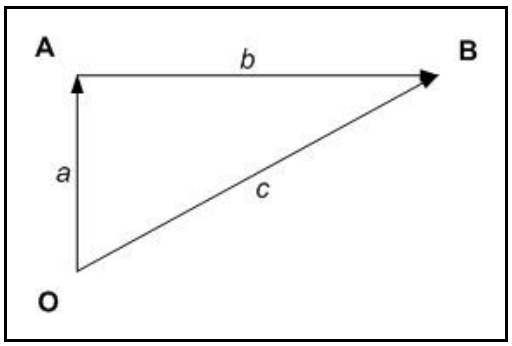


Figura # 6 Definición de Vectores.

El vector está formado por los siguientes elementos:

- La Dirección: Está determinada por la recta de soporte y puede ser vertical, horizontal e inclinada u oblicua.
- La orientación o sentido: Está determinada por la flecha y puede ser horizontal hacia la derecha o hacia la izquierda, vertical hacia arriba o hacia abajo e inclinada ascendente o descendente hacia la derecha o hacia la izquierda.
- El punto de aplicación: Está determinado por el punto origen del segmento que forma el vector.
- La longitud o módulo: Es el número positivo que representa la longitud del vector **[7]**.

1.1.1 Suma de vectores.

Gráficamente, se pueden sumar vectores por dos métodos: el del paralelogramo y el del polígono. El del paralelogramo básicamente consiste en reproducir los dos vectores, contrariamente, es decir, si hay un vector llamado "a" y otro llamado "b", en el método del paralelogramo, la reproducción de "b" estaría al finalizar "a" y viceversa, formando un paralelogramo. El resultado se sacaría uniendo del punto central hacia donde se juntan las reproducciones de "a" y "b" El método del triángulo se parece al método del polígono, a diferencia de que el método del triángulo sólo se admiten 2 vectores y en el del polígono son más de dos. El método del triángulo consiste en hacer una continuación a partir de otro, es decir, tenemos vector "a" y vector "b" donde termina vector v se inicia el vector z y para la resultante sólo se une de un punto de inicio hacia donde termina (Ver Figura # 7) **[7]**.

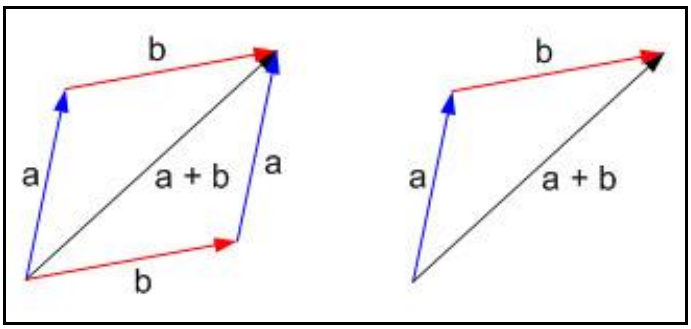


Figura # 7 Suma de vectores.

Partiendo de la representación gráfica de dos vectores, la suma de ambos se consigue colocando el punto de aplicación del segundo vector, a continuación de la flecha del primero, el vector resultante es el que parte del punto de aplicación del primero hasta el final de la flecha del segundo **[7]**.

Analíticamente, partiendo de las coordenadas de los dos vectores:

$$\vec{a} = (a_x, a_y, a_z)\vec{b} = (b_x, b_y, b_z)$$

El vector suma será:

$$\vec{a} + \vec{b} = (a_x, a_y, a_z) + (b_x, b_y, b_z)$$

agrupando:

$$\vec{a} + \vec{b} = (a_x + b_x, a_y + b_y, a_z + b_z)$$

Representando los vectores como combinación lineal de vectores tenemos:

$$\vec{a} = a_x\hat{i} + a_y\hat{j} + a_z\hat{k}\vec{b} = b_x\hat{i} + b_y\hat{j} + b_z\hat{k}$$

El resultado de la suma es:

$$\vec{a} + \vec{b} = (a_x\hat{i} + a_y\hat{j} + a_z\hat{k}) + (b_x\hat{i} + b_y\hat{j} + b_z\hat{k})$$

ordenando los componentes:

$$\vec{a} + \vec{b} = (a_x + b_x)\hat{i} + (a_y + b_y)\hat{j} + (a_z + b_z)\hat{k}$$

Pongamos un ejemplo numérico:

$$\vec{a} = 3\hat{i} + 5\hat{j} - 4\hat{k}\vec{b} = -4\hat{i} + 6\hat{j} - 2\hat{k}$$

el resultado:

$$\vec{a} + \vec{b} = (3\hat{i} + 5\hat{j} - 4\hat{k}) + (-4\hat{i} + 6\hat{j} - 2\hat{k})$$

agrupando términos:

$$\vec{a} + \vec{b} = (3 - 4)\hat{i} + (5 + 6)\hat{j} + (-4 - 2)\hat{k}$$

esto es:

$$\vec{a} + \vec{b} = -\hat{i} + 11\hat{j} - 6\hat{k}$$ **[7]**

1.1.2 Resta de vectores.

La resta de dos vectores es la suma del primero con el opuesto del segundo.

Para hacer la diferencia de dos vectores, basta con aplicar A - B = A + (-B), esto es, sumar el vector opuesto **[7]**.

1.1.3 Producto por un escalar.

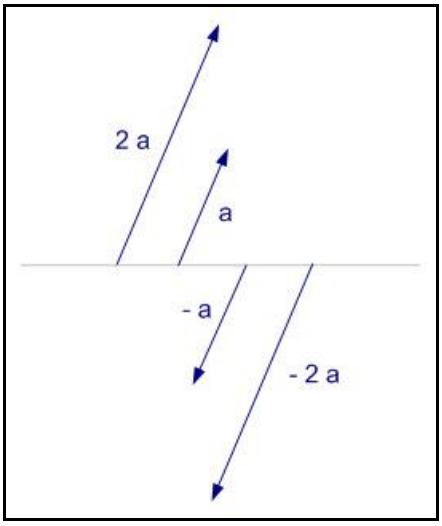


Figura # 8 Producto de un vector por un escalar.

Multiplicar un vector por un escalar es tomar el vector tantas veces como indique el escalar, esto es válido también en los casos en los que el escalar es fraccionario o negativo. Si partimos de la representación gráfica del vector, y sobre la misma línea de su dirección tomamos tantas veces el módulo de vector como marque el escalar (Ver Figura # 8), el resultado es el producto del vector por este escalar, si el signo del escalar es negativo, el sentido del vector será el opuesto al original. Partiendo de un escalar n y de un vector$\vec{a}$, el producto de n por $\vec{a}$ es$n\,\vec{a}$, es el producto de cada una de las coordenadas del vector por el escalar, representando el vector por sus coordenadas: **[7]**

$$\vec{a} = (a_x, a_y, a_z)$$

si lo multiplicamos por el escalar **n**:

$$n \cdot \vec{a} = n \cdot (a_x, a_y, a_z)$$

esto es:

$$n \cdot \vec{a} = (n \cdot a_x, n \cdot a_y, n \cdot a_z)$$

Representando el vector como combinación lineal de los versores:

$$\vec{a} = a_x\hat{i} + a_y\hat{j} + a_z\hat{k}$$

y multiplicándolo por un escalar n:

$$n \cdot \vec{a} = n \cdot (a_x\hat{i} + a_y\hat{j} + a_z\hat{k})$$

esto es:

$$n \cdot \vec{a} = n \cdot a_x\hat{i} + n \cdot a_y\hat{j} + n \cdot a_z\hat{k}$$

Hagamos un ejemplo con valores numéricos, partimos del vector:

$$\vec{a} = 3,2\hat{i} - 1,27\hat{j} + 6\hat{k}$$

y multiplicamos el vector por 2,5:

$$2,5 \cdot \vec{a} = 2,5 \cdot (3,2\hat{i} - 1,27\hat{j} + 6\hat{k})$$

esto es:

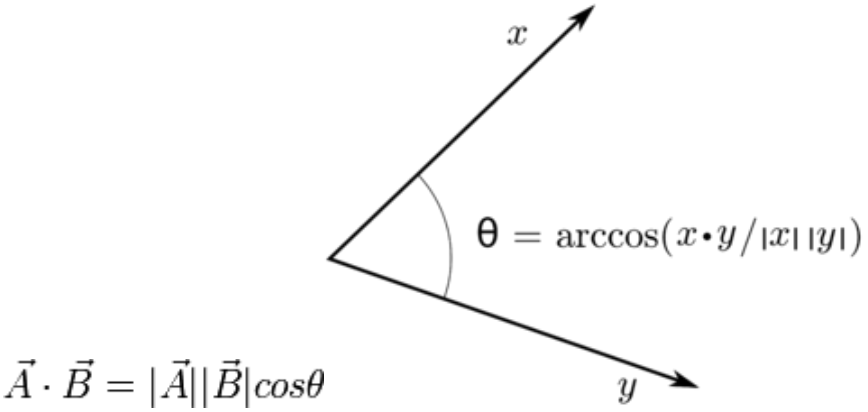
$$2,5 \cdot \vec{a} = 2,5 \cdot 3,2\hat{i} + 2,5 \cdot (-1,27)\hat{j} + 2,5 \cdot 6\hat{k}$$

haciendo las operaciones:

$$2,5 \cdot \vec{a} = 8\hat{i} - 3,175\hat{j} + 15\hat{k}$$ **[7]**

1.1.4 Producto Escalar.

El producto escalar en el caso particular de dos vectores en el plano, o en un espacio euclídeo N-dimensional, se define como el producto de sus módulos multiplicado por el coseno del ángulo θ que forman. El resultado es siempre una magnitud escalar. Se representa por un punto, para distinguirlo del producto vectorial que se representa por un aspa: **[7]**



El producto escalar también puede calcularse a partir de las coordenadas cartesianas de ambos vectores, en una base ortonormal (ortogonal y unitaria, es decir, con vectores de tamaño igual a la unidad y que forman ángulos rectos entre sí): **[7]**

$$\vec{A} \cdot \vec{B} = (a_1, a_2, a_3) \cdot (b_1, b_2, b_3) = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Propiedades del producto escalar en un espacio euclídeo real:

Conmutativa:

$$\vec{A} \cdot \vec{B} = \vec{B} \cdot \vec{A}$$

Asociativa:

$m(\vec{A} \cdot \vec{B}) = (m\vec{A}) \cdot \vec{B} = \vec{A} \cdot (m\vec{B})$ siendo m un escalar.

Distributiva:

$$\vec{A} \cdot (\vec{B} + \vec{C}) = \vec{A} \cdot \vec{B} + \vec{A} \cdot \vec{C}$$

Si los vectores son ortogonales, su producto escalar es nulo (cos 90º = 0), y viceversa **[7]**.

1.1.5 Aplicaciones geométricas analíticas vectoriales.

Denominación	Fórmula Vectorial	Fórmula en coordenadas (en coordenadas cartesianas rectangulares)
Longitud del Vector a.	$a = \sqrt{a^2}$	$a = \sqrt{{a_x}^2 + {a_y}^2 + {a_z}^2}$
Área del paralelogramo construido sobre a y b.	$S = \lvert a\, X\, b \rvert$	$S = \sqrt{\begin{vmatrix} a_y & a_z \\ b_y & b_z \end{vmatrix}^2 + \begin{vmatrix} a_z & a_x \\ b_z & b_x \end{vmatrix}^2 + \begin{vmatrix} a_x & a_y \\ b_z & b_y \end{vmatrix}^2}$
Ángulo entre a y b.	$cos\,\varphi = \frac{ab}{\sqrt{a^2 b^2}}$	$cos\,\varphi = \frac{a_x b_x + a_y b_y + a_z b_z}{\sqrt{{a_x}^2 + {a_y}^2 + {a_z}^2}\ \sqrt{{b_x}^2 + {b_y}^2 + {b_z}^2}}$

Tabla # 1 Ecuaciones vectoriales del plano y la recta [7].

1.2 Agentes Autónomos.

Los agentes autónomos o *NPC* (siglas en inglés, Non Player Character) se utilizan en animaciones de computadoras y otros medios interactivos como los videojuegos y la Realidad Virtual. Estos agentes representan un personaje de una historia o de un juego y tienen la capacidad de realizar sus propias acciones.

La definición de '*agente*' es muy controvertida, son varios los autores que han dado su propia interpretación y definición de agentes, en la actualidad no hay un criterio bien definido. Un primer

pronunciamiento de agente en el año 1996. "*Un agente como una entidad que percibe y actúa sobre un entorno*" **[8]**, a pesar de ser muy sencillo encierra la esencia del tema.

Según diccionarios de la lengua española, en su primera acepción, un agente es una *"Persona que trabaja en una agencia prestando determinados servicios"*. Llevando esta definición al mundo de la computación y viéndolo de manera genérica se puede sustituir el término '*persona*' por '*entidad*', la frase '*trabaja en una agencia*' por '*actúa en un entorno*' y la frase '*prestando determinados servicios*' por '*transformando dicho entorno*'; elaborando un poco el nuevo concepto se puede concluir que un agente es una *entidad que actúa 'de manera autónoma' en un entorno, transformándolo mediante la interrelación con otras entidades.*

Un agente autónomo no es un programa o al menos es algo más que un programa. Una de las características que los distinguen de los programas es la autonomía, lo cual implica que **[9]**:

- ***Son pro-activos:*** No sólo actúan cuando responden a una acción del usuario, sino que también actúan siguiendo sus propios objetivos.
- ***Reactivo:*** El agente debe ser capaz de responder a cambios en el entorno en que se encuentra situado. Actúa como resultado de esos cambios.
- ***Son persistentes:*** No se pueden "apagar"; incluso aunque el usuario no esté interaccionando con ellos, los agentes siguen funcionando, recolectando información, aprendiendo y comunicándose con otros agentes. Esta autonomía implica que sean "agentes situados", es decir, que haya una clara diferencia entre el agente y su entorno. Y la persistencia e independencia del entorno permite que algunos agentes sean móviles.
- ***Social:*** El agente debe comunicarse con otros agentes mediante algún tipo de comunicación.

El movimiento de un *NPC* puede dividirse en tres capas (Ver Figura # 9):

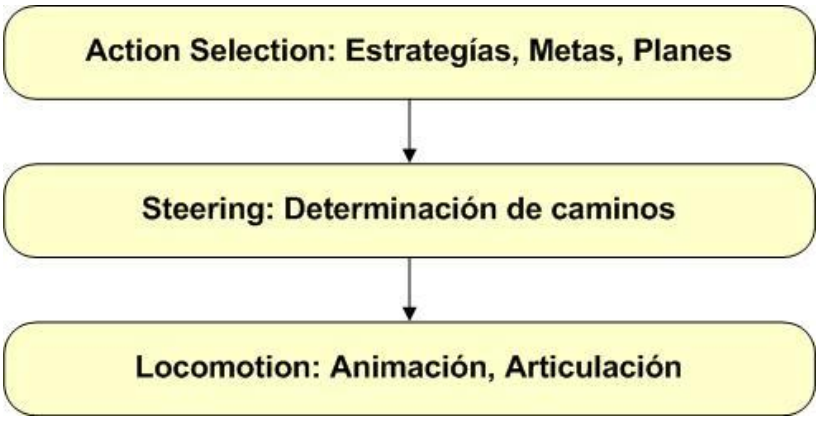


Figura # 9 Capas de movimiento de un NPC.

1. ***Action Selection*:** Esta es la parte de la conducta del agente responsable de elegir sus objetivos y decidir el plan a seguir. Es la parte que dice "ir de aquí" y "A, B, y luego C."
2. ***Steering*:** Esta capa se encarga de calcular las trayectorias deseadas para satisfacer los objetivos y planes de acción establecidos por esta capa. Los *Steering Behaviors* son la implementación de esta capa. Estos describen cuando el agente puede moverse y cuán rápido puede llegar allí.
3. ***Locomotion*:** La capa inferior, la locomoción, representa los aspectos más mecánicos de un agente en movimiento. Es la forma de viajar de A a B. Por ejemplo, si se ha implementado la mecánica de un camello, un tanque, y un perro y luego se dio la orden de viajar al norte, todos utilizan diferentes procesos mecánicos para crear movimiento aunque su intención (moverse al norte) es idéntica.

1.3 Steering Behaviors.

Partiendo de una brillante idea de Craig Reynolds, que ha recorrido el mundo y que mantiene impresionante actualidad, la manipulación de los *Steering Behaviors* de los elementos virtuales y sus aplicaciones, siguen siendo aplicadas en diversas tareas relacionadas con juegos y entornos virtuales. Con una implementación y fundamento muy sencillo, con un basamento físico-matemático accesible y eficiente, sigue mostrando resultados muy positivos en esta rama. De varios artículos y libros se han conocido con mayor o menor grado la manera de aplicarlos y de llegar a resolver un problema determinado **[10]**.

1.3.1 Behaviors Seek y Flee.

Los *behaviors Seek* y *Flee* implementan patrones de comportamiento complementarios. La lógica es la misma para ambos, su única diferencia se encuentra en la fuerza directriz resultante. *Seek* se utiliza para dirigir el agente hacia un punto en específico, un ejemplo podría ser el de un pez nadando directamente a su comida. El objetivo puede ser un punto fijo en el espacio u otro agente en movimiento. El *behavior Flee* es usado para simular una forma simple de evasión y como en el caso del *Seek* el objetivo puede encontrarse estacionario o en movimiento **[11]**.

La fuerza directriz se calcula basándose en el valor de la velocidad actual del agente y la posición del objetivo. Primero la posición del objetivo es expresada como un vector entre el objetivo y el agente, este representa la velocidad deseada. Y la fuerza directriz resulta de la diferencia entre este vector y el vector de la velocidad actual (Ver Figura # 10).

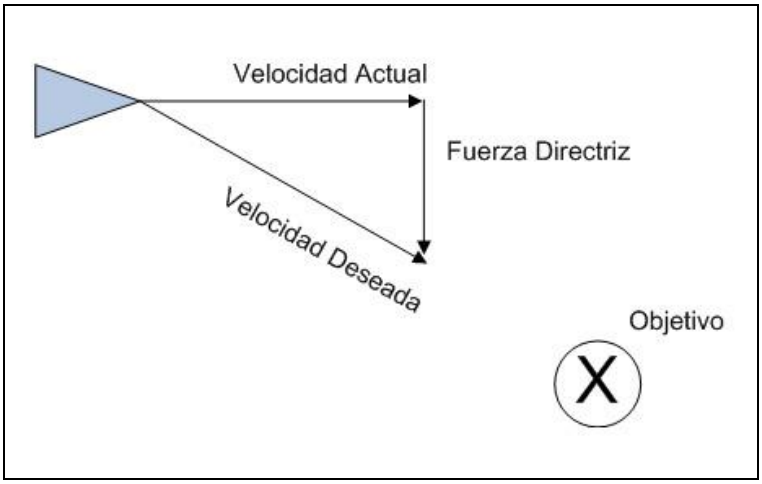


Figura # 10 Vector fuerza del behavior Seek [11].

En el *behavior Seek* a la velocidad deseada se le sustrae la velocidad actual para el cálculo de la fuerza directriz. En cambio en el *behavior Flee* este cálculo se realiza sustrayendo la velocidad deseada de la velocidad actual. Haciendo uso de estos simples cálculos el agente siempre puede ser guiado hacia su objetivo o forzado a evadir este. Como en todos los *behaviors*, la fuerza calculada no debe exceder la fuerza máxima del agente **[11]**.

Este presenta un inconveniente, desde el momento en que el *behavior Seek* comienza a calcular continuamente la fuerza directriz, en muchos casos surgen resultados no deseados. Si el objetivo se encuentra estático o moviéndose a una velocidad mucho más baja que la del agente que lo persigue.

En primer lugar la fuerza directriz lo guiará hacia el objetivo. Pero si se mantuviera viajando en esa misma dirección hasta pasar el objetivo, en ese momento una fuerza en el sentido opuesto sería calculada y el agente se movería marcha atrás por donde mismo vino. El movimiento resultante sería como el de una mariposa nocturna en torno a una lámpara **[11]**.

1.3.2 Behavior Arrive.

El *behavior Arrive* es una extensión del *behavior Seek*, al igual que este el *Arrive* es usado para guiar al agente hacia un objetivo específico. La diferencia más importante es el modo de arribar al destino. El *behavior Seek* llega al destino a toda velocidad por lo que ocurren problemas como los explicados anteriormente mientras que el *behavior Arrive* disminuye la velocidad del vehículo de manera controlada de acuerdo a las especificaciones hechas por el usuario y lo detiene en la posición deseada **[11]**.

El cálculo de la fuerza directriz comienza de la misma manera que en el *behavior Seek*. El vector resultante es la diferencia entre la velocidad deseada y la velocidad actual. Hasta donde se ha visto no existe diferencia entre los dos *behaviors*. Mientras el objetivo se encuentre fuera de la distancia de activación especificada en el atributo *Distancia,* ninguna fuerza directriz será generada **[11]**.

Ya dentro de esta área, la fuerza será modificada por la distancia hasta el objetivo dividida entre un factor de desaceleración. Esto hace que el vehículo disminuya su velocidad de acuerdo a la distancia actual del destino. El resultado final es que el vehículo se detenga en el destino especificado (Ver Figura # 11).

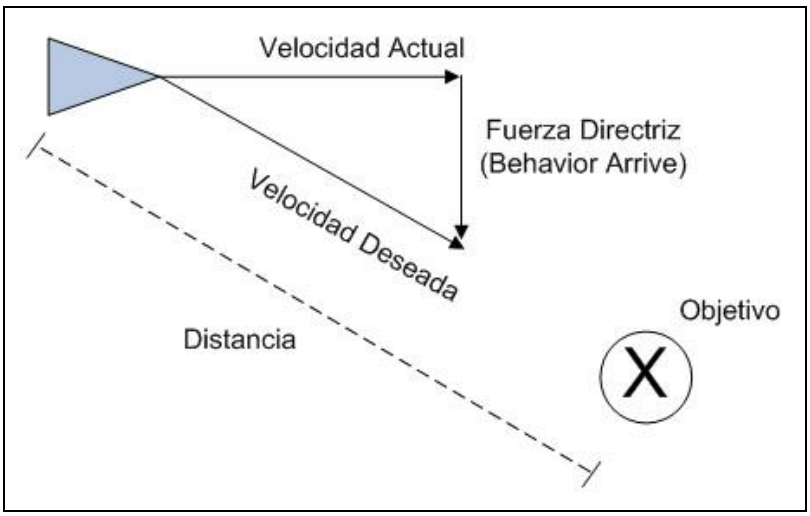


Figura # 11 Vector fuerza del behavior Arrive [11].

Este algoritmo puede presentar un problema, mediante el uso de la distancia dividida por el número de pasos para alcanzar la meta, el vehículo se acerca al destino solo asintóticamente.

1.3.3 Behavior Separation.

El *behavior Separation* se utiliza para mantener una cierta distancia entre los vehículos. Esto ayuda en la prevención de las colisiones de ellos. Los seres humanos hacen lo mismo de manera automática. Sólo las personas en la inmediata distancia se tienen en cuenta en sus acciones, la mayor parte del tiempo este conjunto se reduce sólo a las personas delante de él **[11]**.

La base de este *behavior* es la teoría del comportamiento de partículas con carga eléctrica que evitan el hacinamiento, repeliéndose mutuamente. Lo primero que hay que hacer es encontrar todos los agentes dentro de una cierta distancia cercana al nuestro. Esto se logra utilizando el objeto de simulación “Neighborhood” (Vecindario). (Ver Figura # 12).

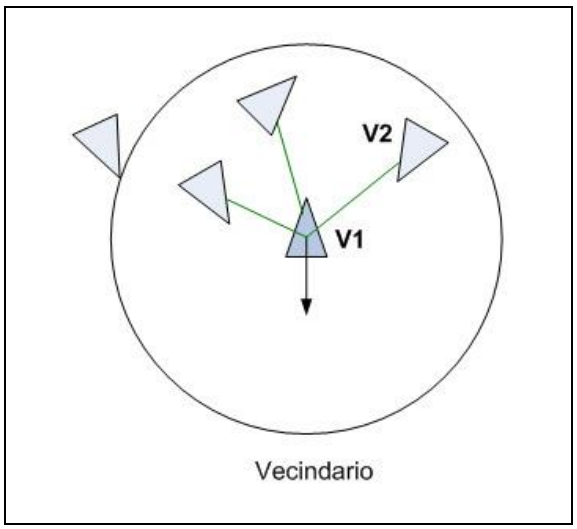


Figura # 12 Selección de los vecinos [11].

De acuerdo a la distancia entre los dos objetos la fuerza será más fuerte si la distancia es menor y más suave si se encuentra más lejos. Todas las fuerzas calculadas se suman entre si y forman la fuerza directriz **[11]**.

1.3.4 Behavior Alignment.

Los agentes utilizan el *behavior Alignment* para tratar de formar grupos con otros. Para realizar el agrupamiento el agente es dirigido por la fuerza promedio de los agentes circundantes. Para esto un iterador sobre todos los agentes dentro del área radial es creado (Ver Figura # 13).

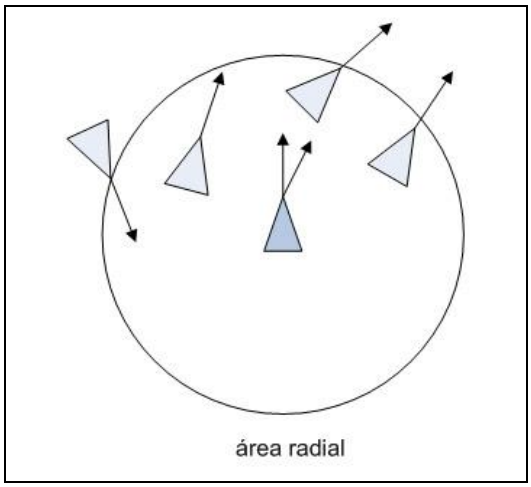


Figura # 13 Candidatos para el alineamiento.

Los vectores de velocidad resultantes de la serie de agentes dentro del área radial se suman. Después de esto el resultado es dividido entre la cantidad de agentes para calcular la dirección promedio **[11]**.

1.3.5 Obstacle Avoidance.

Para dirigir el agente a través de un camino angosto de forma realista es necesario moverlo en forma de espectador el cual se encuentra a la espera y anticipa los sucesos. De aquí la necesidad de definir un *behavior* el cual permita el manejo de obstáculos. En la vida real, tanto humanos como animales eluden chocar con los obstáculos. Cuando alguien siente que muere de ganas por tomar un vaso de agua fría, este no se mueve a la nevera en línea recta. Inconscientemente escoge un camino que lo guía por entre todos los obstáculos de ese mundo, como sillas, mesas y otras cosas las cuales se encuentran muy bien situadas acorde al ambiente casero. El *behavior* "*Obstacle Avoidance*" implementa esto inconscientemente evadiendo los obstáculos predefinidos.

La primera cosa cuando se implementa el "*Obstacle Avoidance*" es la definición de los obstáculos dentro del entorno virtual. Desde entonces esta simulación es basada en una escena bidimensional, usando formas geométricas simples en substitución de los objetos complejos de la vida real. Una posible representación simple de los obstáculos es el círculo (o la esfera si se trata de un entorno 3D).

Este debe contener el obstáculo completo. Es necesario tener cuidado no perder sin necesidad mucho espacio en la representación, ejemplo para una larga pared el círculo es una mala aproximación. Las ventajas de usar el círculo (o la esfera si se tratase de un entorno 3D) son las simples fórmulas para determinar intersecciones con otros cuerpos geométricos. La desventaja es la mala aproximación de los objetos complejos (Ver Figura # 14).

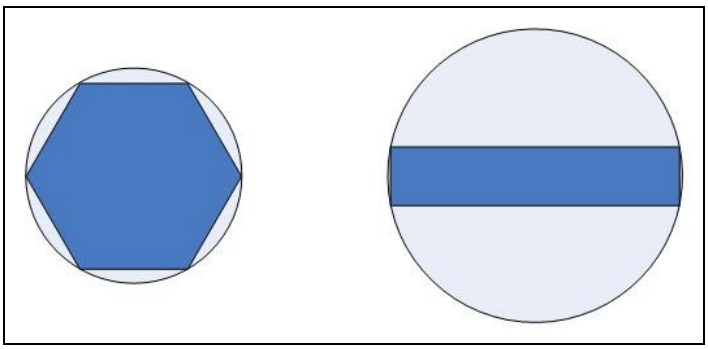


Figura # 14 Buena y mala aproximación usando círculo.

Los resultados obtenidos en un espacio bidimensional pueden ser aplicados sin problemas a un entorno 3D.

El primer paso para el *behavior Obstacle Avoidance* es encontrar cuales objetos se hallan en una vecindad cercana. Después de tener estos objetos es necesario mantener en el agente un área rectangular (una caja de detección), extendiéndose al frente del agente libre de colisiones. El ancho de la caja de detección debe ser igual al radio del agente y su longitud será proporcional a la velocidad actual del agente (mientras mayor sea la velocidad del agente, mayor será el tamaño de la caja de detección) **[12]**.

Para la detección del punto de colisión más cercano con los objetos debemos hacer varios pasos **[12]**:

1. El agente solo tendrá en cuenta aquellos objetos que se encuentren dentro del alcance de su caja de detección. Inicialmente el algoritmo del *Obstacle Avoidance* recorre todos los obstáculos del mundo y marca aquellos dentro de su rango para un próximo tratamiento**.**
2. El algoritmo debe transformar las coordenadas de todos estos obstáculos marcados a coordenadas locales del agente. Esto haría más fácil el tratamiento ya que los objetos con coordenada *x* negativa serían descartados como se muestra en la Figura # 15.

3. El algoritmo ahora deberá chequear cuales obstáculos se superponen a la caja de detección. Las coordenadas locales son muy útiles en este momento y todo lo que se necesita es expandir el radio del objeto a la mitad del ancho de la caja de detección (el radio del ancho del agente) y entonces se verifica si el valor de la coordenada *y* del objeto es menor que este valor. Si no es así entonces quiere decir que el objeto no intercepta la caja de detección y por tanto no hay colisión y este objeto puede ser descartado como se muestra en la Figura # 15.

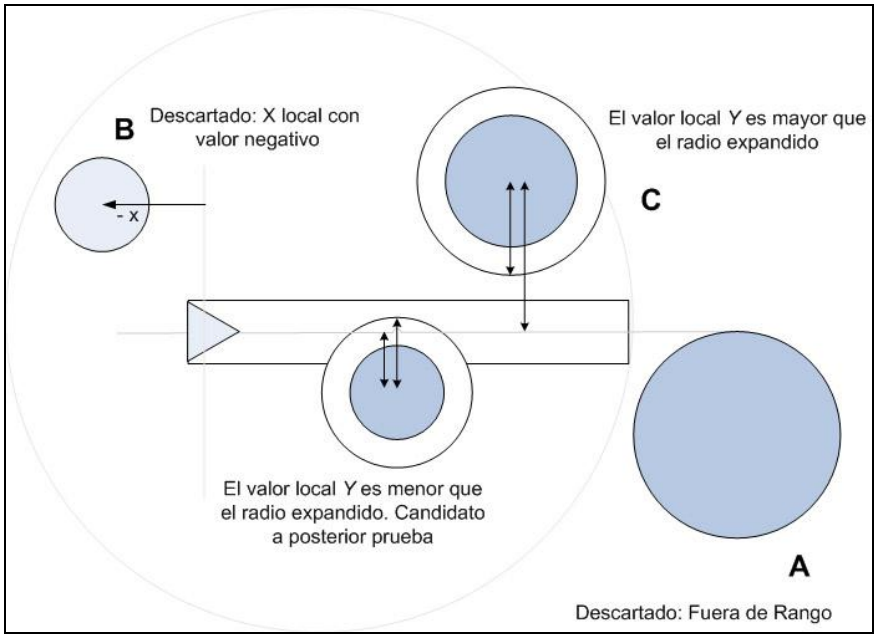


Figura # 15 Pasos dos y tres.

4. Ya en este punto solo quedan aquellos objetos que interceptan la caja de detección. Ahora es necesario encontrar el punto de intersección más cercano al agente. Usando la expansión del paso 3 y una simple prueba de intercepción de una línea y un círculo podemos encontrar donde el círculo expandido corta al eje-*x.* Aquí tenemos dos puntos de intercepción como se muestra en la Figura # 16. Nótese que es posible tener un obstáculo frente al agente y este puede tener un punto de intercepción detrás del agente. Este se muestra en la figura con el obstáculo A, el algoritmo debe descartar este caso y solo considerar los puntos de intercepción que se encuentren en el eje-*x* positivo.

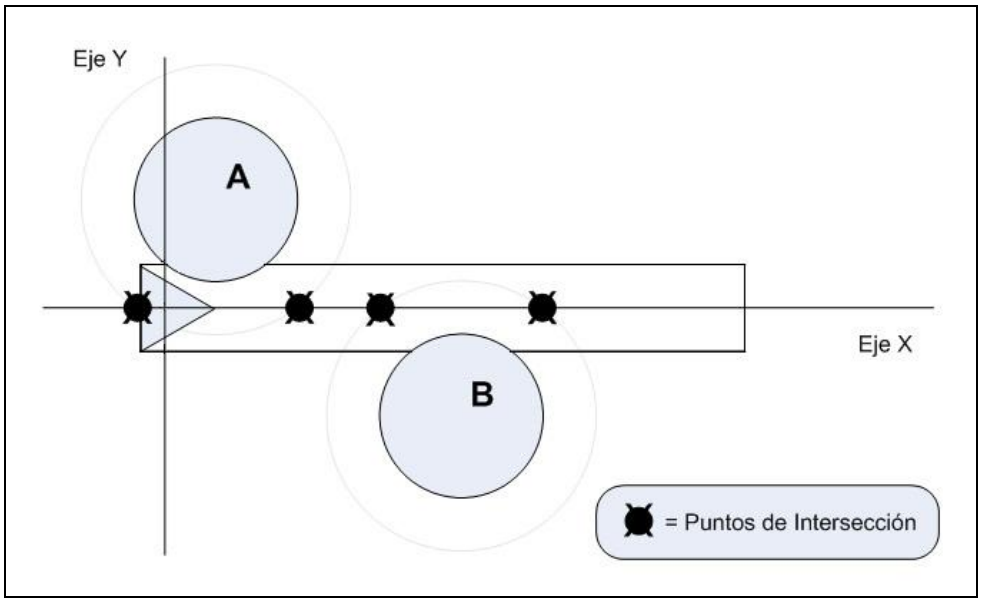


Figura # 16 Paso cuatro.

1.4 Motores Gráficos.

Un Motor Gráfico (*Graphic Engine*, en inglés), es el componente de software principal de un video juego o de otra aplicación interactiva que se ejecute en tiempo real. Su uso simplifica el desarrollo de la aplicación y a menudo permite que el juego pueda correr en múltiples plataformas, tales como Consolas de Videos Juegos y Sistemas Operativos de Mac OS, GNU/Linux y Microsoft Windows.

Los *Graphic Engine* están construidos sobre plataformas tecnológicas de software/hardware subyacentes a ellos como los NVidia Chipset, OpenGL, XNA/DirectX, etc. Los servicios de estas plataformas son generalmente más primitivos y más complejos de explotar que los servicios ofrecidos por el Graphic Engine por lo que la abstracción del hardware es una de sus principales características.

Los *Graphic Engine* ofrecen un conjunto de herramientas de desarrollo visual, además de componentes de software reutilizables que agrupados en subsistemas que presentan alta cohesión en relación a sus comportamientos **[14]**. Los subsistemas más comunes de un *Graphic Engine* son:

- Procesamiento de Entradas.
- Gráficos.

- Animación.
- Audio.
- Comportamiento e Inteligencia Artificial.
- Conectividad / Red.

Cada subsistema es un componente por derecho propio, un componente con una estructura particular e independiente, y formado por otros subsistemas más especializados.

Los *Graphic Engines* son productos altamente complejos y especializados, capaces de ahorrar tiempo y dinero a través del fomento de la reutilización enfocada a diferentes aspectos del video juego.

Tener un *Graphic Engine* ayuda a ser productivo, a crear juegos sofisticados con menos esfuerzo, y a recuperar y aplicar las mejores prácticas que los expertos de la industria de producción de videos juegos han acumulado a lo largo de los años.

Importantes Engines Open Source **[13]**:

1. Crystal Space.
2. Ogre3D.
3. Irrlicht.
4. jMonkeyEngine (jME).
5. G3D Engine.
6. The Nebula Device 2.
7. Realm Force.
8. Blender Game Engine.
9. OpenSceneGraph.

1.4.1 G3D Engine.

La selección del *Graphic Engine* a usar es de suma importancia, porque ello repercutirá en la calidad del producto final. Existe una amplia gama de *Graphic Engines*, pero entre los lugares cimeros se encuentra G3D Engine.

G3D es un *Engine* Open Source bajo licencia BSD, es muy usado en juegos comerciales (Ver Figura # 17) y aplicaciones para simuladores militares. G3D soporta rendering en tiempo real, off-line rendering como el raytracing, y propósitos generales en el cómputo del GPU (Graphics Processing Unit, en

inglés). G3D proporciona un conjunto de rutinas y estructuras comunes que son necesitadas en todas las aplicaciones gráficas. Garantiza el uso de librerías de bajo nivel como OpenGL y los sockets para la red se pueden usar sin restricciones de las funcionalidades y de rendimiento **[15]**. G3D presenta una arquitectura robusta y optimizada, permitiendo integrar otras librerías de forma sencilla y rápida; convirtiéndose en una excelente herramienta para el desarrollo de videos juegos y simuladores **[16]**.

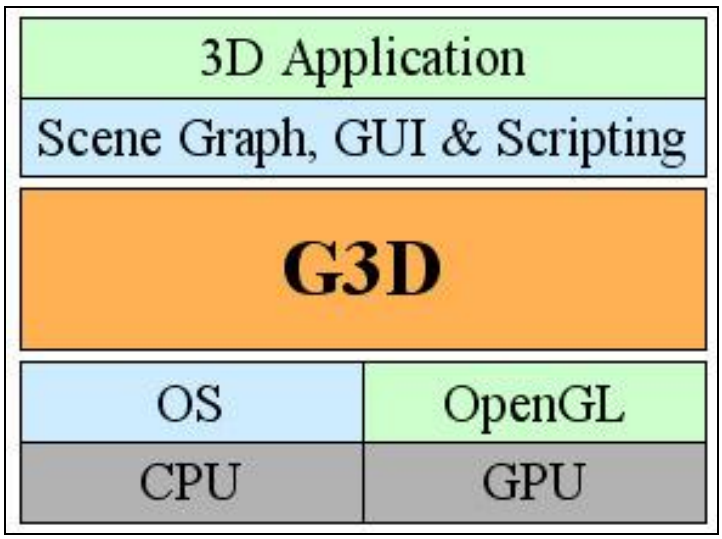


Figura # 17 Estructura G3D Engine. (*Tomada de la ayuda de G3D*).

A continuación se muestra una tabla de las características fundamentales de G3D *Engine* **[14]**

Author	**Morgan McGuire**		
Graphics API	OpenGL, DirectX, Software, Other	**Operating Systems**	Windows, Linux
Programming Language	C/C++	**Status**	Alpha
Documentation	Yes		

Tabla # 2 Principales características de G3D.

1.5 Metodología y Herramientas de desarrollo.

Para llevar a cabo la realización de nuestro módulo de *IA* existen en el mundo un gran número de aplicaciones y herramientas de desarrollo. Para seleccionarlas se utilizaron una serie de parámetros tales como ventajas, tendencia actual, dominio y fortaleza.

1.5.1 RUP.

Durante más de 40 años las técnicas de desarrollo software han ido evolucionando, en pro de la calidad de los productos obtenidos y de disminuir el esfuerzo, los tiempos y costos de los proyectos que las utilizan. Así han surgido y se han documentado y puesto en práctica diversos paradigmas y metodologías fundamentalmente en las últimas dos décadas, yendo desde el clásico y conocido método Cascada (Waterfall) hasta los más recientes, como lo es el Proceso Unificado desarrollado por Rational Software Corporation (RUP).

La metodología aplicada en el desarrollo del trabajo es RUP debido a que puede especializarse para una gran variedad de sistemas de software con diferentes áreas de aplicación, diferentes tipos de organizaciones, diferentes niveles de aptitud y diferentes tamaños de proyecto.

Existen tres características fundamentales que lo hacen una metodología robusta y poderosa:

- Dirigido por Casos de Uso.
- Es un proceso centrado en la arquitectura.
- Es iterativo e incremental.

Que RUP esté dirigido por Casos de Uso significa que el proceso de desarrollo sigue una trayectoria que avanza a través de los flujos de trabajo generados por los Casos de Uso. Los Casos de Uso se especifican y diseñan en el principio de cada iteración y describen la funcionalidad total del sistema, pensada en términos de la importancia de la misma para el usuario.

La arquitectura involucra los elementos más significativos del sistema y está influenciada entre otros por las plataformas software, los sistemas operativos, los sistemas gestión de bases de datos, además de otros como sistemas heredados y requerimientos no funcionales. Por esta razón se dice que RUP está centrado en la arquitectura, lo que involucra más la relación con los principios de la usabilidad.

Es iterativo e incremental porque el proyecto o el desarrollo de una aplicación se pueden dividir en partes y desarrollarlas de manera iterativa, incrementándose a medida que se integran unas con otras

hasta llegar a formar la tarea final. Las iteraciones hacen referencia a pasos en el flujo de trabajo y los incrementos, el crecimiento del producto.

1.5.2 Visual Studio 2005.

El Visual Studio 2005, es una herramienta poderosa, fuerte y voluminosa que tiene una gran integración de varios lenguajes entre ellos el C++, C#, y Asp.Net. Tiene la posibilidad de implementar aplicaciones para soluciones integrales que aprovechen de manera óptima la ventaja de cada lenguaje, además de tener una interfaz amigable con el usuario. Visual Studio 2005 proporciona una amplia gama de herramientas que ofrecen multitud de ventajas para desarrolladores individuales y equipos de desarrollo de software:

- Mayor productividad y obtención más rápida de resultados.
- Creación de soluciones dinámicas basadas en Windows, la web, dispositivos móviles y Office.
- Comunicación y colaboración más eficaz en sus equipos de software.
- Garantía de calidad rápida y continua en todo el proceso de desarrollo.

1.5.3 Rational Rose 2003.

Rose es una herramienta con plataforma independiente que ayuda a la comunicación entre los miembros del equipo, a monitorear el tiempo de desarrollo y a entender el entorno de los sistemas. Una de las grandes ventajas de Rose es que utiliza la notación estándar en la arquitectura de Software (UML), la cual permite a los arquitectos de software y desarrolladores visualizar el sistema completo utilizando un lenguaje común. Otra ventaja de Rose es que los diseñadores pueden modelar sus componentes e interfaces en forma individual y luego unirlos con otros componentes del proyecto. Además Rose soporta la construcción de componentes en lenguajes como C++, VisualBasic, Java, Ada, genera IDL's para aplicaciones CORBA. Por todo lo anterior Rose es la herramienta de Análisis, Diseño, Modelado y Construcción de software Orientado a Objetos líder en el mercado.

Algunas características incluidas en Rational Rose 2003:

- Integración entre WinDNA y Microsoft VisualStudio.
- Mejoras en la generación de código con Java y aplicaciones CORBA.
- Integración con ClearCase.

- Mejora en la comunicación entre los miembros del equipo de desarrollo.

1.6 Lenguaje de Modelado.

El lenguaje de modelado de objetos es un conjunto estandarizado de símbolos y de modos de disponerlos para modelar (parte de) un diseño de software orientado a objetos.

Algunas organizaciones los usan extensivamente en combinación con una metodología de desarrollo de software para avanzar de una especificación inicial a un plan de implementación y para comunicar dicho plan a todo un equipo de desarrolladores. El uso de un lenguaje de modelado es más sencillo que la auténtica programación, pues existen menos medios para verificar efectivamente el funcionamiento adecuado del modelo.

1.6.1 UML.

El Lenguaje Unificado de Modelado (UML) es una consolidación de muchas de las notaciones y conceptos más usados orientados a objetos. Empezó como una consolidación del trabajo de Grade Booch, James Rumbaugh, e Ivar Jacobson, creadores de tres de las metodologías orientadas a objetos más populares.

Previamente, un diseño orientado a objetos podría haber sido modelado con cualquiera de la docena de metodologías populares, causando a los revisores tener que aprender las semánticas y notaciones de la metodología empleada antes que intentar entender el diseño en sí. Ahora con UML, diseñadores diferentes modelando sistemas diferentes pueden sobradamente entender cada uno los diseños de los otros.

Entre los diagramas estándares mencionados anteriormente podemos encontrar a:

- Diagramas de Casos de Uso para modelar los procesos 'business'.
- Diagramas de Secuencia para modelar el paso de mensajes entre objetos.
- Diagramas de Colaboración para modelar interacciones entre objetos.
- Diagramas de Estado para modelar el comportamiento de los objetos en el sistema.
- Diagramas de Actividad para modelar el comportamiento de los Casos de Uso, objetos u operaciones.
- Diagramas de Clases para modelar la estructura estática de las clases en el sistema.

- Diagramas de Objetos para modelar la estructura estática de los objetos en el sistema.
- Diagramas de Componentes para modelar componentes.
- Diagramas de Implementación para modelar la distribución del sistema.

1.7 Lenguaje de Programación.

Un lenguaje de programación es un lenguaje que puede ser utilizado para controlar el comportamiento de una máquina, particularmente una computadora. Consiste en un conjunto de reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos, respectivamente.

C++.

El lenguaje C++ como lenguaje de programación orientado a objetos en sí tiene múltiples ventajas en las que se encuentran:

- Eficiencia.
- Uniformidad.
- Comprensión: Los datos componen los objetos y los procedimientos que los manipulan están agrupados en clases que se corresponden con las estructuras de información que el programa trata.
- Flexibilidad: Al tener relacionados los procedimientos que manipulan los datos con los datos a tratar, cualquier cambio que se realice sobre ellos quedará reflejado automáticamente en cualquier lugar donde estos datos aparezcan.
- Estabilidad: Dado que permite un tratamiento diferenciado de aquellos objetos que permanecen constantes en el tiempo sobre aquellos que cambian con frecuencia permite aislar las partes del programa que permanecen inalterables en el tiempo.
- Reusabilidad: La noción de objeto permite que programas que traten las mismas estructuras de información reutilicen las definiciones de objetos empleadas en otros programas e incluso los procedimientos que los manipulan. De esta forma, el desarrollo de un programa puede llegar a ser una simple combinación de objetos ya definidos donde estos están relacionados de una manera particular.

- Otra de las razones por las cuales se utilizó C++ como lenguaje de programación es debido a su increíble versatilidad. Con él pueden programarse desde los programas más simples hasta los programas más complicados como son los sistemas operativos.

Es además portable, es decir, un programa con el código escrito en C++, se podrá compilar en cualquier sistema operativo o sistema informático sin necesidad de cambiar casi el código fuente. Este es por ejemplo uno de los grandes secretos de Linux, al estar el código escrito en este lenguaje(al menos en su concepción original), es más fácil portarlo a diferentes ordenadores como PC´s, Macintosh, incluso superordenadores.

Otras de las grandes ventajas del C++, es que es un lenguaje multi-nivel, es decir, puedes usarlo tanto para programar directamente el hardware(dependiendo del sistema operativo, eso sí), como para crear aplicaciones tipo Windows definidas todas por poseer una misma interfaz.

Capítulo 2. Solución Propuesta

El juego “Rápido y Curioso” está soportado sobre G3D, poderoso engine gráfico, cuenta con un incipiente editor de pistas “Trak-Edit” el cual se presenta como un módulo dentro del proyecto brindando pocas funcionalidades para la implementación de los *Steering Behaviors* básicos tratados en el capítulo anterior, entre estas funcionalidades se encuentra la de poder editar la pista a través de la inserción de nodos unidos por aristas. La física del mismo ha sido modelada con la librería de clases ODE.

Se puede querer en algún momento que el auto frene justo antes de llegar a una curva o que disminuya la velocidad justo antes de estar cerca de un punto determinado de la pista pero en otro momento se puede querer que gire el timón con el ángulo necesario para doblar una curva o incluso aumentar o disminuir la velocidad según su posición en la pista, quizás simplemente que llegue a un lugar determinado lo más rápido posible, en fin son varios los casos que se pudieran tener, más o menos complejos. El carro está implementado de manera tal que necesite solo tres valores fundamentales para moverse, el ángulo de giro de las gomas, la posición del pedal de aceleración y la fuerza de frenado. ¿Cómo saber qué ángulo de giro de las gomas darle al carro en cada momento? ¿Cómo saber la posición del pedal de aceleración? ¿Cómo saber la fuerza de frenado en caso de que sea necesario aplicarla en una curva o en un momento determinado? Todo esto depende de los comportamientos que se quieran obtener del auto.

A continuación se explican los comportamientos propuestos. Cada uno de ellos contenido en una clase que hereda de la genérica ‘S*teeringBehaviors*’. Para el desarrollo de estos *behaviors* es necesario verificar el fichero de configuración del carro que va a incorporar estos comportamientos puesto que se trabaja con la velocidad máxima admitida por el vehículo según las características de la caja de velocidad del mismo, la masa de este y el máximo ángulo de giro del timón. Para el correcto funcionamiento de estos *behaviors* dentro del juego es necesario situar los nodos en las rectas lo más al centro de la carretera, para las curvas se debe situar uno a la entrada y otro a la salida y si es posible se recomienda escalarlos al ancho de la pista en los lugares que lo admitan.

2.1 Behavior Angle.

El *Behavior Angle* se utiliza en el cálculo del ángulo de giro que debe tomar el vehículo para moverse a un punto específico. El ángulo se calcula basándose en el vector dirección del vehículo y la dirección entre el nodo actual y el nodo Target. Primero la dirección entre los nodos es expresada como el vector que los une y el ángulo sería el formado entre estos dos vectores, vector dirección del vehículo y el vector entre el nodo actual y el Target (Ver Figura # 18).

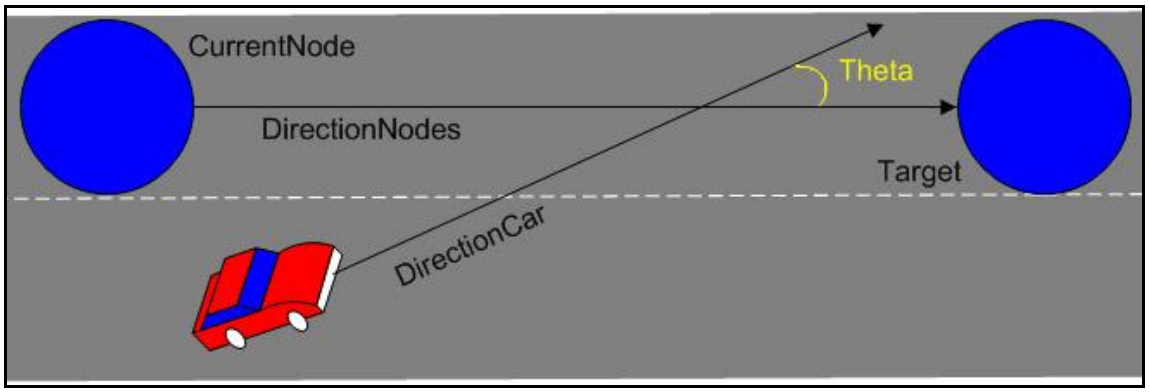


Figura # 18 Ángulo formado entre los vectores DirectionCar y DirectionNodes.

Una vez que se tengan los vectores de *DirectionCar* y *DirectionNodes*, se calcula el ángulo de giro utilizando la ecuación matemática que expresa el ángulo entre dos vectores:

$$Cos\,(Theta) = \frac{DirectionNodes * DirectionCar}{|DirectionNodes| * |DirectionCar|}$$

$$Theta = aCos\left(\frac{DirectionNodes * DirectionCar}{|DirectionNodes| * |DirectionCar|}\right)$$

La simulación del giro después de calculado el ángulo tiene sus particularidades puesto que un chofer no gira con todo el ángulo necesario sino que dobla el timón con un valor proporcional y que converja al ángulo deseado para corregir su trayectoria. Para lograr esto es necesario el cálculo de un valor de convergencia el cual se calcula dividiendo la velocidad actual del vehículo (*CurrentVelocity*) entre la velocidad deseada (*DesiredVelocity*) como se muestra a continuación. Teniendo este valor de convergencia solo queda multiplicarlo por el valor del ángulo de giro.

$$Convergencia = \frac{CurrentVelocity}{DesiredVelocity}$$

Una vez calculado el valor de ángulo si este valor es 0.0 (radianes), es decir son paralelos los vectores de *DirectionNodes* y *DirectionCar*, entonces la posición actual del ángulo de giro del timón se mantiene si está en el centro, o se corrige a este (Ver Figura # 19).

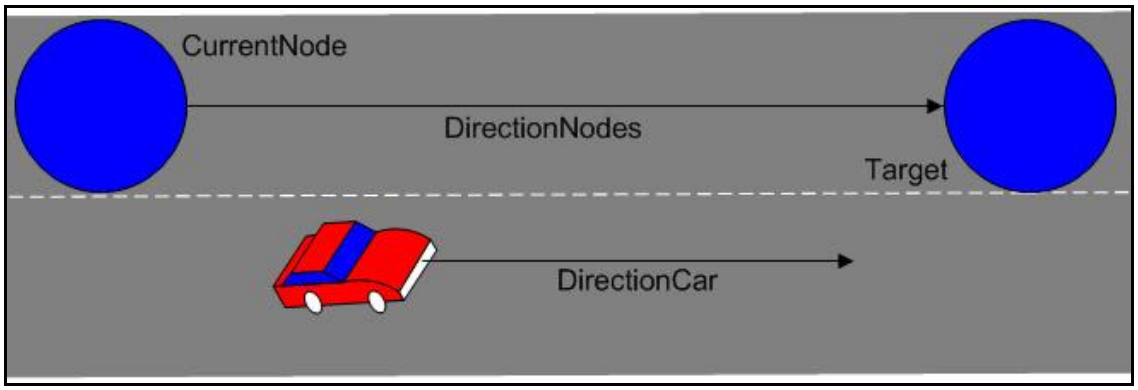


Figura # 19 Paralelismo entre los vectores *DirectionCar* y *DirectionNodes*.

En caso de que el valor calculado del ángulo sea distinto de 0.0 (radianes) es decir que los vectores no son paralelos, tendríamos entonces que definir hacia que lado darle el ángulo de giro al vehículo, hacia la derecha o hacia la izquierda.

El mundo en el videojuego “Rápido y Curioso” está orientado al eje de abscisas (-Z) en el sistema de coordenadas globales tal y como se muestra en la Figura # 20.

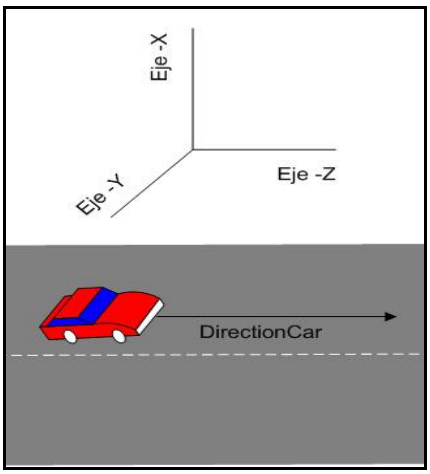


Figura # 20 Sistema de coordenadas globales.

Contando con esta información debemos convertir entonces las coordenadas globales del nodo Target en coordenadas locales del carro. Estas coordenadas locales toman como centro el centro del vehículo, orientando a la abscisa (-Z) sobre el vector *DirectionCar* (Ver Figura # 21).

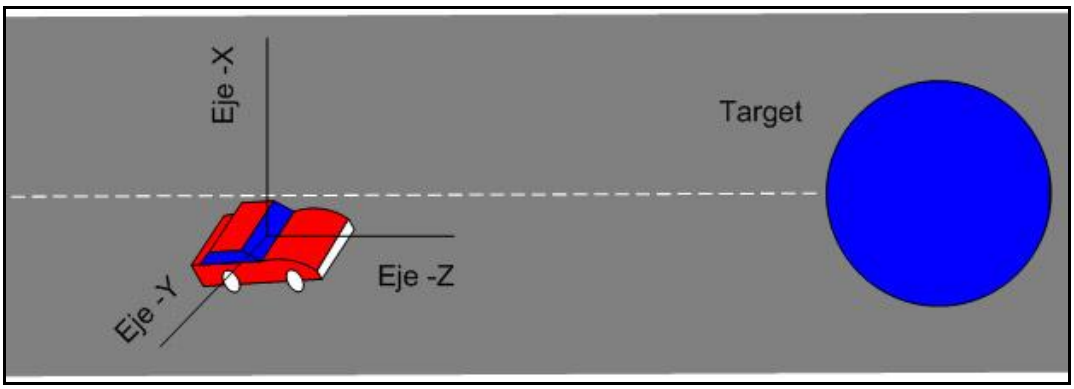


Figura # 21 Sistema de coordenadas locales.

Una vez que hemos convertido en coordenadas locales las coordenadas globales del nodo Target y sabiendo que el vector *DirectionCar* coincide con la abscisa (-Z) basta con comparar el valor de la coordenada X del nodo Target, si el valor es positivo el vehículo se encuentra a la izquierda y sino se encuentra a la derecha del nodo.

2.2 Behavior Accelerator.

El *Behavior Accelerator* se utiliza para calcular la posición del pedal de aceleración del carro. Para calcular esta posición se debe tener en cuenta que la posición del pedal del acelerador oscila entre los valores de 0-100 y la velocidad máxima del carro en cada instante (*Max_Speed_Car*) depende de la velocidad de la caja de aceleración que se esté ejecutando en cada momento, conociendo esto y la velocidad actual (*CurrentVelocity)* del vehículo podemos entonces, realizando una regla de tres, calcular esta posición como se muestra a continuación:

$$Max\_Position\_Accelerator - Max\_Speed\_Car$$

$$Current\_Position\_Accelerator - CurrentVelocity$$

$$Current\_Position\_Accelerator = \frac{Max\_Position\_Accelerator * Current\_Velocity}{Max\_Speed\_Car}$$

En caso de que el nodo Target sea un nodo que se encuentra encima de una curva la posición del pedal de aceleración es entonces igual a cero.

$$Current\_Position\_Accelerator = 0$$

Por otra parte si la velocidad actual del carro es cero entonces la posición del pedal del acelerador es la posición máxima que este puede tomar:

$$Current\_Position\_Accelerator = Max\_Position\_Accelerator$$

Para el buen funcionamiento del behavior es importante destacar que sobre cada curva del mapa debe existir un nodo curva.

2.3 Behavior Brake.

El *Behavior Brake* se utiliza para calcular el valor de la fuerza de frenado aplicada al carro en un instante determinado. Para realizar el cálculo de esta fuerza de frenado es necesario conocer la velocidad y posición actual del carro, su masa, la posición del nodo Target que es la posición por la cual, según un umbral de distancia de frenado se va a aplicar dicha fuerza al carro, este nodo contiene información de si es un nodo curva o no, además si es una curva pronunciada o no. Además es necesario conocer un umbral de velocidad el cual indica la velocidad máxima con la cual el vehículo puede tomar una curva determinada la cual está en función del tipo de curva, estas pueden ser de dos tipos curvas pronunciadas o no pronunciadas (Ver Figura # 22 Behavior_Brake.).

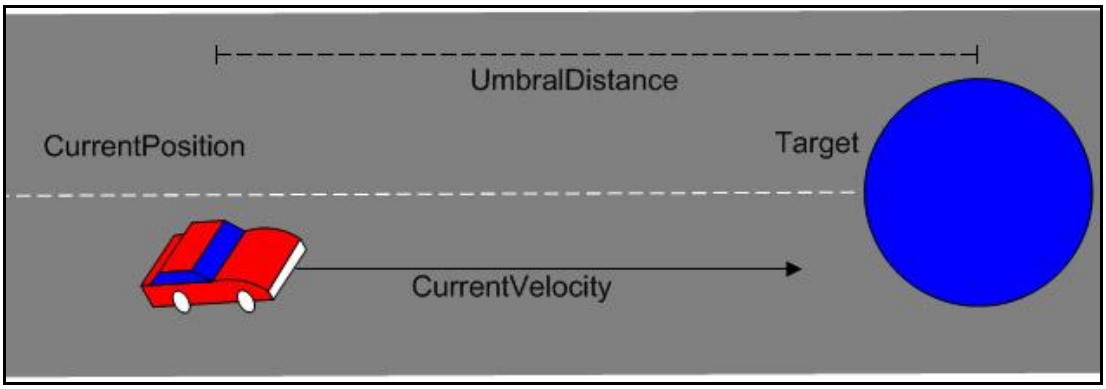


Figura # 22 Behavior_Brake.

Una vez que el vehículo entra en la distancia definida en el umbral (***UmbralDistance***) entonces calculamos el valor de la fuerza de frenado mediante la ecuación que se muestra a continuación y la comenzamos a aplicar mientras se encuentre dentro de esta distancia.

$$FuerzaFrenado = \frac{VelocidadActual^{2}}{Distancia} * Masa$$

Capítulo 3. Descripción de la Solución Propuesta

Esta descripción a nivel conceptual de la solución propuesta en el capítulo anterior, se encarga de definir las reglas para una correcta aplicación y funcionamiento del módulo de *IA*. Se tratarán los aspectos fundamentales para el entendimiento de la solución mediante un modelo de dominio. Además de recoger los requisitos funcionales y no funcionales del sistema.

3.1 Reglas del Negocio.

Para una correcta aplicación del módulo de *IA* el juego debe estar sustentado sobre G3D, la física del mismo debe estar implementada con la librería ODE, debe contar con un editor de pistas el cual incluya funcionalidades como posicionar nodos en cualquier lugar de la pista, cambiar el diámetro de dichos nodos y proveer toda la información de estos.

3.2 Modelo de Dominio.

El modelo a continuación recoge los conceptos que se manejan en el desarrollo de la solución así como las relaciones que se establecen entre estos.

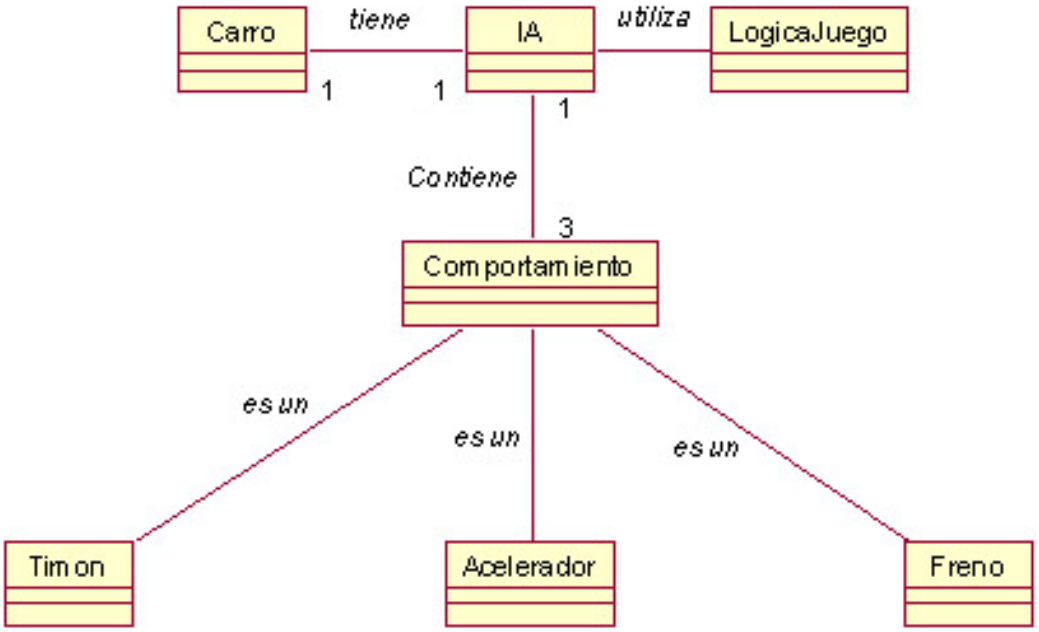


Figura # 23 Modelo de Dominio

3.3 Glosario de Términos.

Se relacionan a continuación todos los conceptos presentes en el modelo anterior así como sus especificaciones para una mejor comprensión de la solución.

***Carro*:** NPC que se moverá de forma inteligente por la pista.

***IA*:** Es quien detecta de forma inteligente que comportamiento ejecutar en una situación determinada.

***Comportamiento*:** Representa la abstracción más general de un comportamiento.

***Timón*:** Es quien calcula el ángulo de giro que debe tomar el NPC ante una situación dada.

Acelerador: Es quien calcula la aceleración que debe tomar el NPC en cada instante.

Freno: Es quien calcula la fuerza de frenado a tomar por el NPC en determinado momento.

3.4 Captura de Requisitos.

A continuación se exponen las condiciones que deben existir y funcionalidades que se deben alcanzar para dar solución a nuestro problema en forma de requisitos funcionales y requisitos no funcionales.

3.4.1 Requisitos Funcionales.

1. Recorrer la pista de forma autónoma e inteligente.
    1.1 Verificar donde se encuentra el carro (recta o curva).
    1.2 Acelerar en recta o cuando la velocidad sea cero.
    1.3 Desacelerar si se encuentra ante una curva.
    1.4 Verificar umbral de proximidad a una curva.
    1.5 Verificar velocidad máxima permitida ante una curva.
    1.6 Aplicar fuerza de frenado en el momento requerido.
2. Girar el timón con el ángulo preciso.
    2.1 Verificar paralelismo de la trayectoria.
    2.2 Calcular factor de convergencia.
    2.3 Calcular ángulo de giro.

3. Acelerar y desacelerar.

    3.1 Calcular posición del acelerador.

4. Frenar en el momento necesario.

4.1 Calcular fuerza de frenado.

3.4.2 Requisitos no Funcionales.

Representan las características, cualidades y limitaciones del producto, reconocibles porque responden a la pregunta ¿Cómo el producto cumple con los Requerimientos Funcionales?

1. Usabilidad

- Cualquier videojuego de autos que quiera incluir NPCs.

2. Requerimientos de Software

- Sistema Operativo Windows XP o superior.

3. Requerimientos de Hardware

- Microprocesador Intel Pentium 3 o superior.
- Memoria RAM de 256 MB o superior.
- Tarjeta de Video 64 MB o superior.

4. Restricciones en el Diseño y la implementación

- Lenguaje de programación C++ bajo el paradigma de programación Orientado a Objeto.

5. Requerimientos de Soporte

- Compatibilidad con el Sistema Operativo Windows.

6. Legales

- Se regirá por las normas ISO 9000.

7. Diseño e Implementación

- Visual Studio 2005 Sp1
- Librería G3D v6.0.
- Librería ODE.

3.5 Modelo de Casos de Uso del Sistema.

Los casos de uso son un artefacto clave en el Proceso Unificado de Desarrollo de Software, ya que son el depósito principal de los requisitos funcionales que gobiernan el diseño, la construcción, las pruebas, y muchos otros aspectos de este proceso, por esto RUP está dirigido por casos de uso.

3.5.1 Definición de los Actores del Sistema.

Actores	Justificación
Sistema	Se beneficiará con las funcionalidades que brinda el módulo, de manera general: moverse de forma autónoma e inteligente por la pista.

Tabla # 3 Actores del sistema, justificación.

3.5.2 Casos de Uso del Sistema.

1. Recorrer la pista de forma inteligente.
2. Girar_timón.
3. Acelerar.
4. Frenar.

3.5.3 Especificación de los casos de uso.

CU1	Jugar inteligentemente.
Actor	Sistema.
Descripción:	Mover de forma inteligente el *NPC* por la pista.
Referencia:	R1, R1.1, R1.2, R1.3, R1.4, R1.5, R1.6, CU_Girar_timon(include), CU_Determinar_Aceleracion(include), CU_Calcular_Fuerza_de_Frenado(include).

Tabla # 4 Descripción caso de uso Jugar inteligentemente.

CU2	Girar_timon.
Lo Inicia	CU_Jugar inteligentemente.
Descripción:	Proveer el ángulo de giro de las gomas del *NPC* (carro).
Referencia:	R2, R2.1, R2.2, R2.3

Tabla # 5 Descripción caso de uso Girar_timon.

CU3	Determinar_ Aceleracion.
Lo Inicia	CU_ Jugar inteligentemente.
Descripción:	Calcular posición del pedal de aceleración del carro.
Referencia:	R3, R3.1

Tabla # 6 Descripción caso de uso Determinar_Aceleracion.

CU4	Calcular Fuerza de Frenado.
Lo Inicia	CU_ Jugar inteligentemente.
Descripción:	Determinar la fuerza de frenado necesaria para aplicar al *NPC*.
Referencia:	R4, R4.1

Tabla # 7 Descripción caso de uso Calcular Fuerza de Frenado.

3.5.4 Diagrama de Casos de Uso del Sistema.

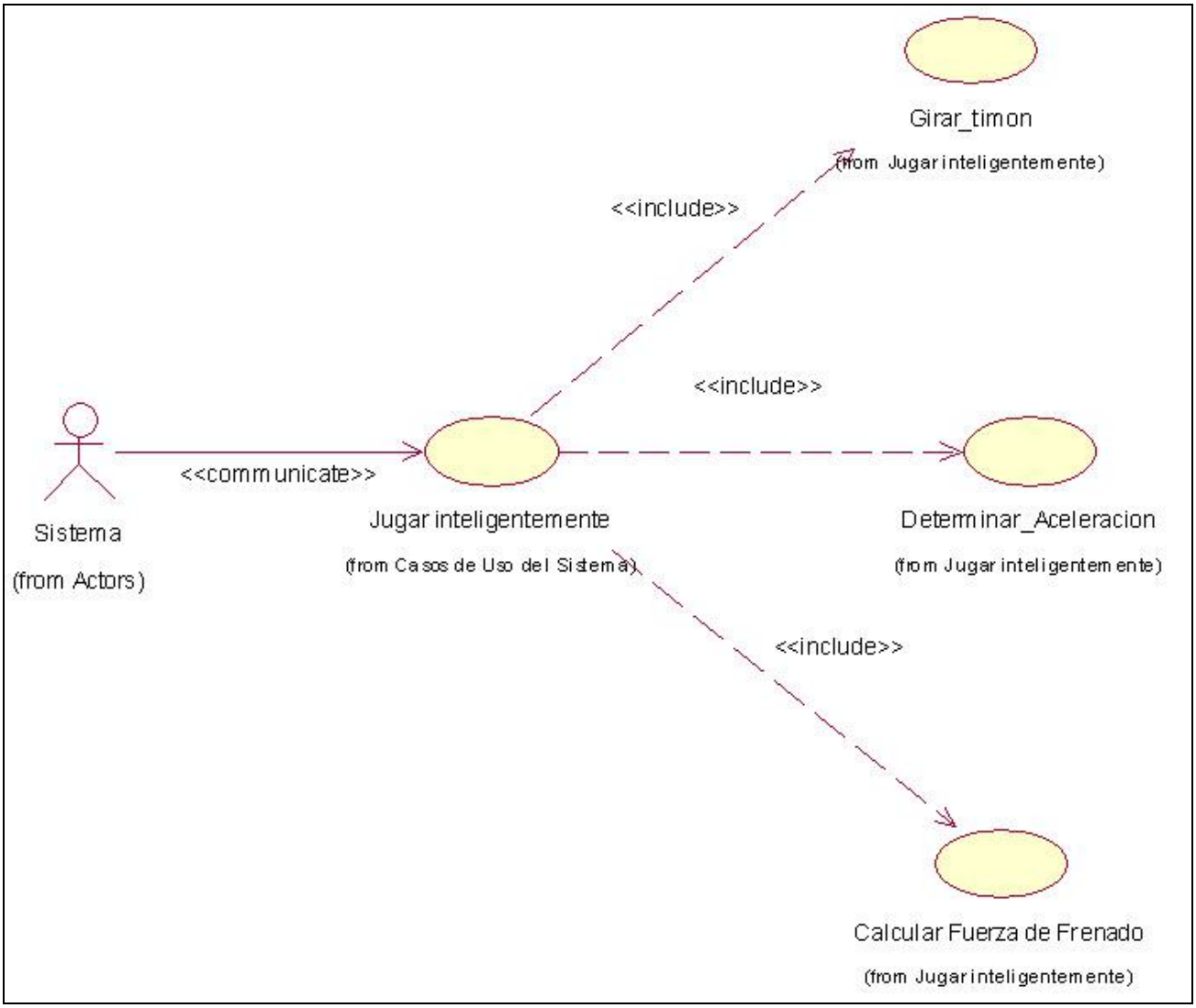


Figura # 24 Modelo Casos de Uso del Sistema.

3.5.5 Especificación de los casos de usos en formato expandido.

Nombre del CU	Jugar inteligentemente.
Actores	Sistema
Propósito	Mover de forma inteligente el *NPC* por la pista.

Resumen:	El caso de uso se inicia cuando el actor encuesta a su inteligencia para moverse por la pista, está en dependencia de la situación en que se encuentre manda a calcular el ángulo de giro de las gomas, la posición del pedal de aceleración y la fuerza de frenado.
Referencias	R1, R1.1, R1.2, R1.3, R1.4, R1.5, R1.6, CU_Girar_timon(include), CU_Determinar_Aceleracion(include), CU_Calcular_Fuerza_de_Frenado(include).
Precondiciones	Debe estar creado el grafo de caminos.
Curso normal de los eventos:	
Acción del actor	**Respuesta del sistema**
1- Encuesta la inteligencia.	2- Se verifica la situación en la que se encuentra el *NPC*.
	3- Verifica si está en una recta o si la velocidad es cero.
	4- Manda a calcular la aceleración.
	5- Verifica el umbral de distancia permitido ante una curva.
	5- Se chequea la velocidad máxima permitida antes de entrar a la curva.
	6- Se desacelera el carro.

	7- Se manda a calcular la fuerza de frenado.
	8- Se le entregan estos valores al *NPC*.
Poscondiciones	Calculados los valores necesarios para recorrer la pista.

Tabla # 8 Descripción CU Recorrer la pista de forma inteligente.

Nombre del CU	Girar_timon.
Lo Inicia	CU_ Jugar inteligentemente.
Propósito	Proveer el ángulo de giro de las gomas del *NPC* (carro).
Resumen:	El caso de uso se inicia cuando se necesita calcular el ángulo de giro de las gomas para que el *NPC* se mueva por la pista.
Referencias	R2, R2.1, R2.2, R2.3
Precondiciones	Los nodos del camino deben estar situados correctamente.
Curso normal de los eventos:	
Acción del actor	**Respuesta del sistema**
1- Se manda a calcular el ángulo de giro.	2- Se verifica el paralelismo de la dirección del carro y el vector entre los nodos.
	3- Se calcula el factor de convergencia.
	4- Se calcula el valor del ángulo de giro.
Poscondiciones	Calculado el valor del ángulo de giro.

Tabla # 9 Descripción CU Girar_ Timon.

Nombre del CU	Determinar_ Aceleracion.
Lo Inicia	CU_ Jugar inteligentemente.
Propósito	Calcular posición del pedal de aceleración del carro.
Resumen:	El caso de uso se inicia cuando se necesita calcular la aceleración del carro.

Referencias	R3, R3.1
Precondiciones	Los nodos del camino deben estar situados correctamente y proveer la información de nodos curvas o no.
Curso normal de los eventos:	
Acción del actor	**Respuesta del sistema**
1- Se manda a calcular la posición del pedal de aceleración.	2- Se calcula la posición del pedal aceleración.
Poscondiciones	Calculada la posición del pedal de aceleración.

Tabla # 10 Descripción CU Determinar_ Aceleracion.

Nombre del CU	Calcular Fuerza de Frenado.
Lo Inicia	CU_ Jugar inteligentemente.
Propósito	Determinar la fuerza de frenado necesaria para aplicar al *NPC*.
Resumen:	El caso de uso se inicia cuando se necesita determinar la fuerza de frenado.
Referencias	R4, R4.1
Precondiciones	Los nodos del camino deben estar situados correctamente y proveer la información de nodos curvas o no.
Curso normal de los eventos:	
Acción del actor	**Respuesta del sistema**
1- Se necesita determinar la fuerza de frenado.	2- Se calcula la fuerza de frenado necesaria.
Poscondiciones	Calculada la fuerza de frenado óptima.

Tabla # 11 Descripción CU Calcular Fuerza de Frenado.

Capítulo 4. Diseño, Implementación y Resultados

A continuación se muestra el diagrama de clases del sistema, resultado del refinamiento de las etapas anteriores. También se describen cada una de las clases del diseño y se muestra el diagrama de componentes como muestra de la realización de las clases en componentes físicos que se traducen en ficheros .h y .cpp correspondiente a la implementación en C++.

4.1 Diagrama de clases del diseño.

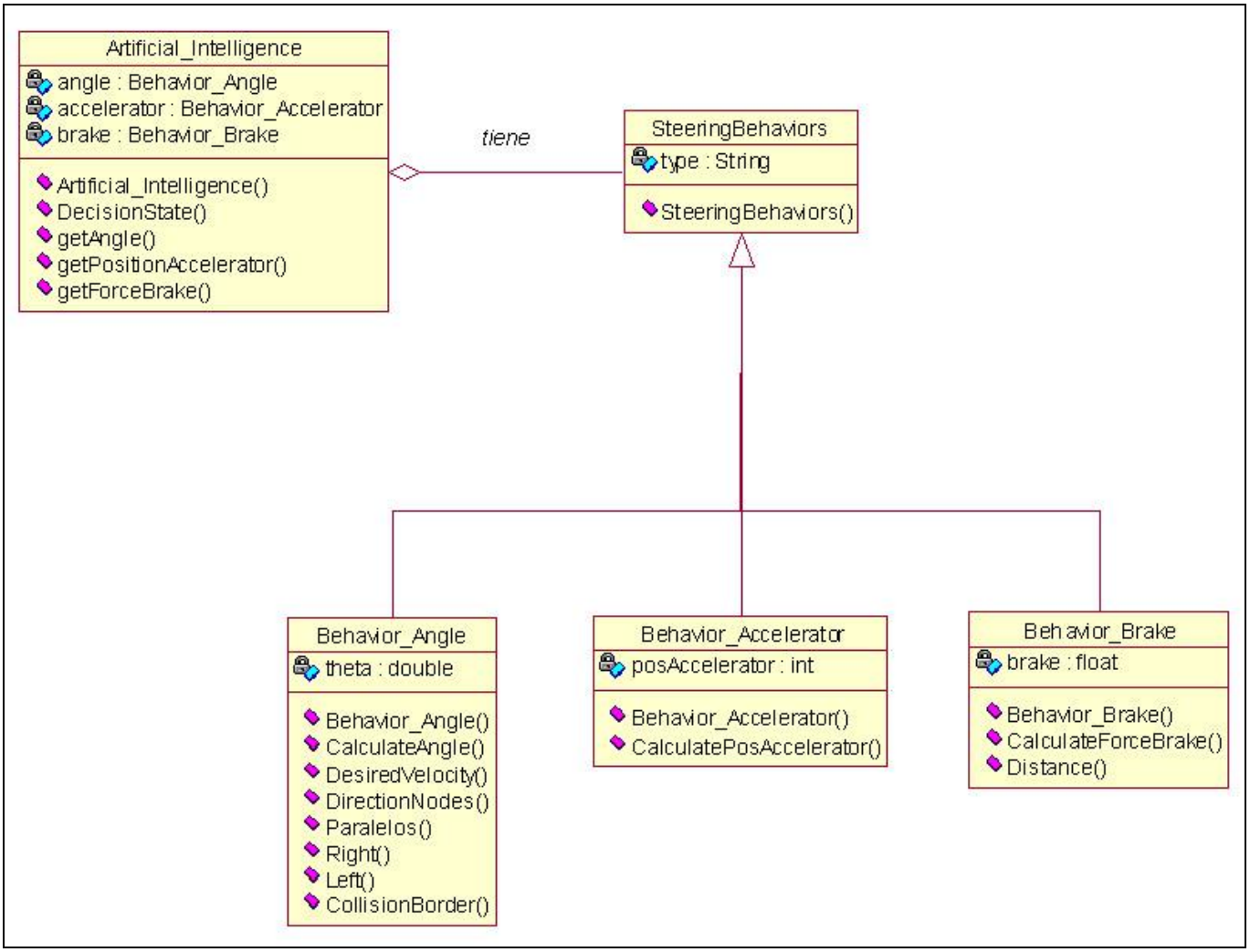


Figura # 25 Diagrama de clases del diseño.

4.2 Descripción de las clases del diseño.

Nombre:	Artificial_Intelligence	
Tipo de clase:	Controladora	
Atributo		**Tipo**
angle		Behavior_Angle
accelerator		Behavior_Accelerator
brake		Behavior_Brake
UmbralDistance		int
UmbralVelocity		int
Para cada responsabilidad:		
Nombre:	DecisionStates	
Descripción:	Función que se encarga, pasándole la lógica del juego y el carro, de ejecutar los distintos comportamientos de acuerdo al estado actual del juego.	
Nombre:	getAngle	
Descripción:	Función que se encarga, pasándole la posición del nodo actual, la posición del próximo nodo, el radio del próximo nodo, la velocidad actual del carro, la posición actual del carro, la dirección del carro, el ancho del carro y el coordinateFrame del carro, de llamar al método CalculateAngle() del Behavior_Angle.	
Nombre:	getPositionAccelerator	
Descripción:	Función que se encarga, pasándole la velocidad actual del carro y la velocidad actual de la caja de velocidad del carro, de llamar al método CalculatePosAccelerator() del Behavior_Accelerator.	
Nombre:	getForceBrake	
Descripción:	Función que se encarga, pasándole la velocidad actual del carro, la posición actual del carro, la posición del próximo nodo a donde se dirige el carro, la masa del carro, los umbrales de velocidad y distancia, de llamar al método CalculateForceBrake() del Behavior_Brake.	

Tabla # 12 Descripción de la clase Artificial_Intelligence.

Nombre:	SteeringBehavior	
Tipo de clase:	Entidad	
Atributo		**Tipo**
type		String
Para cada responsabilidad:		
Nombre:		
Descripción:		

Tabla # 13 Descripción de la clase SteeringBehavior.

Nombre:	Behavior_Brake	
Tipo de clase:	Entidad	
Atributo		**Tipo**
brake		float
Para cada responsabilidad:		
Nombre:	CalculateForceBrake	
Descripción:	Función que se encarga, pasándole la velocidad actual del carro, la posición actual del carro, la posición del próximo nodo, la masa del carro, los umbrales de distancia y velocidad, de calcular el valor de la fuerza de frenado del carro.	
Nombre:	Distance	
Descripción:	Función que se encarga, pasándole la posición actual del carro y la posición del próximo nodo, de calcular la distancia entre el carro y el próximo nodo.	

Tabla # 14 Descripción de la clase Behavior_Brake.

Nombre:	Behavior_Angle	
Tipo de clase:	Entidad	
Atributo		**Tipo**
angle		double

Para cada responsabilidad:	
Nombre:	CalculateAngle
Descripción:	Función que se encarga, pasándole la posición del nodo actual, la posición del próximo nodo, el radio del próximo nodo, la velocidad actual del carro, la posición actual del carro, la dirección del carro, el ancho del carro y el coordinateFrame del carro, de calcular el valor del ángulo de giro de las gomas del carro.
Nombre:	DesiredVelocity
Descripción:	Función que se encarga, pasándole la posición actual del carro y la posición del próximo nodo, de calcular el valor de la velocidad máxima deseada del carro.
Nombre:	DirectionNodes
Descripción:	Función que se encarga, pasándole la posición del nodo actual y la posición del próximo nodo, de calcular el vector formado entre estos dos nodos.
Nombre:	Paralelos
Descripción:	Función que se encarga, pasándole el vector dirección del carro y el vector formado entre los nodos actual y siguiente, de verificar si estos vectores son paralelos.
Nombre:	Izquierda
Descripción:	Función que se encarga, pasándole la posición del próximo nodo, de verificar si se encuentra a la izquierda del carro.
Nombre:	Derecha
Descripción:	Función que se encarga, pasándole la posición del próximo nodo, de verificar si se encuentra a la derecha del carro.
Nombre:	CollisionBorder
Descripción:	Función que se encarga, pasándole la posición del próximo nodo, el radio del próximo nodo y el ancho del carro, de verificar si se colisiona o no con los bordes de la pista.

Tabla # 15 Descripción de la clase Behavior_Angle.

Nombre:	Behavior_Accelerator	
Tipo de clase:	Entidad	
Atributo		**Tipo**
PosAccelerator		int
Para cada responsabilidad:		
Nombre:	CalculatePosAccelerator	
Descripción:	Función que se encarga, pasándole la velocidad actual del carro y la velocidad de la caja de velocidad que se está ejecutando del carro, de calcular el valor del pedal de aceleración del carro.	

Tabla # 16 Descripción de la clase Behavior_Accelerator.

4.3 Diagrama de Secuencia.

En este diagrama se muestran las interacciones entre objetos, ordenadas en secuencia temporal durante un escenario concreto.

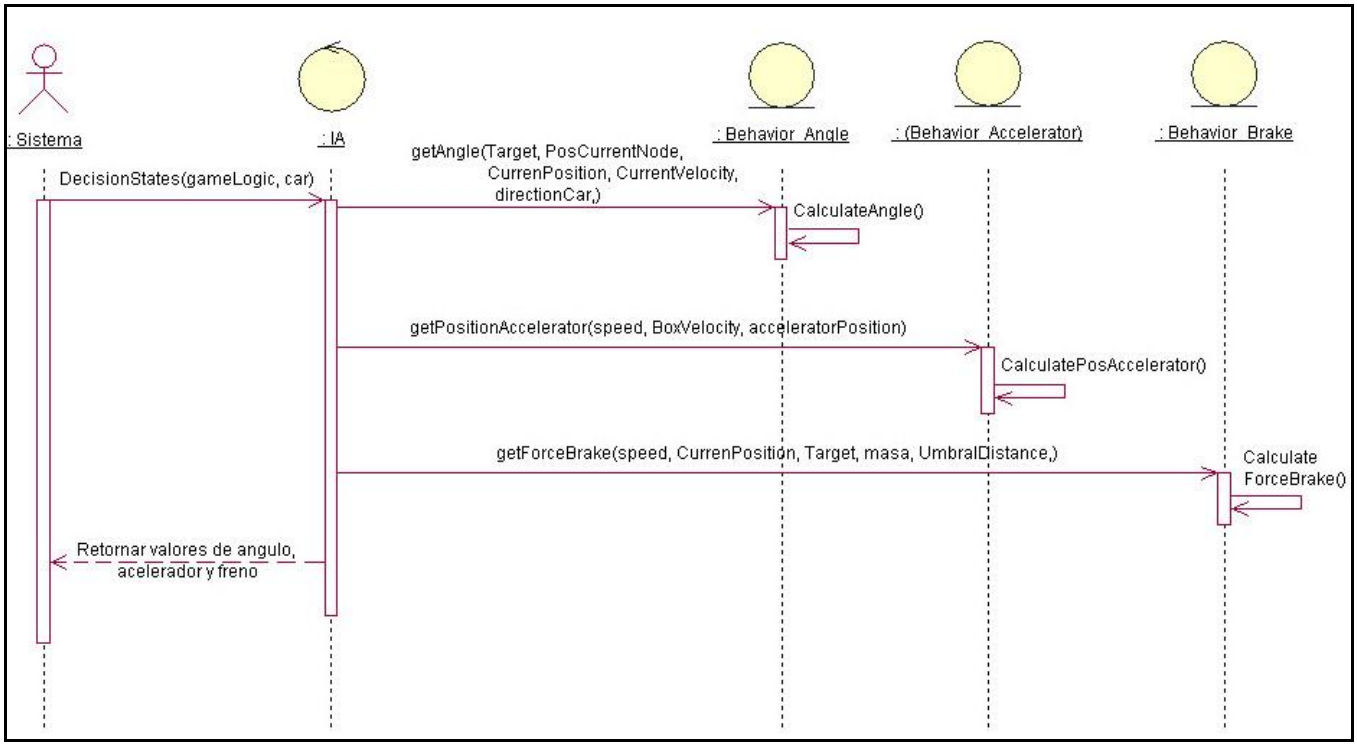


Figura # 26 Diagrama de Secuencia.

4.4 Diagrama de Componentes.

En este diagrama se modela la vista estática del sistema. Muestra la organización y las dependencias lógicas entre el conjunto de componentes software, sean éstos componentes de código fuente, librerías, binarios o ejecutables.

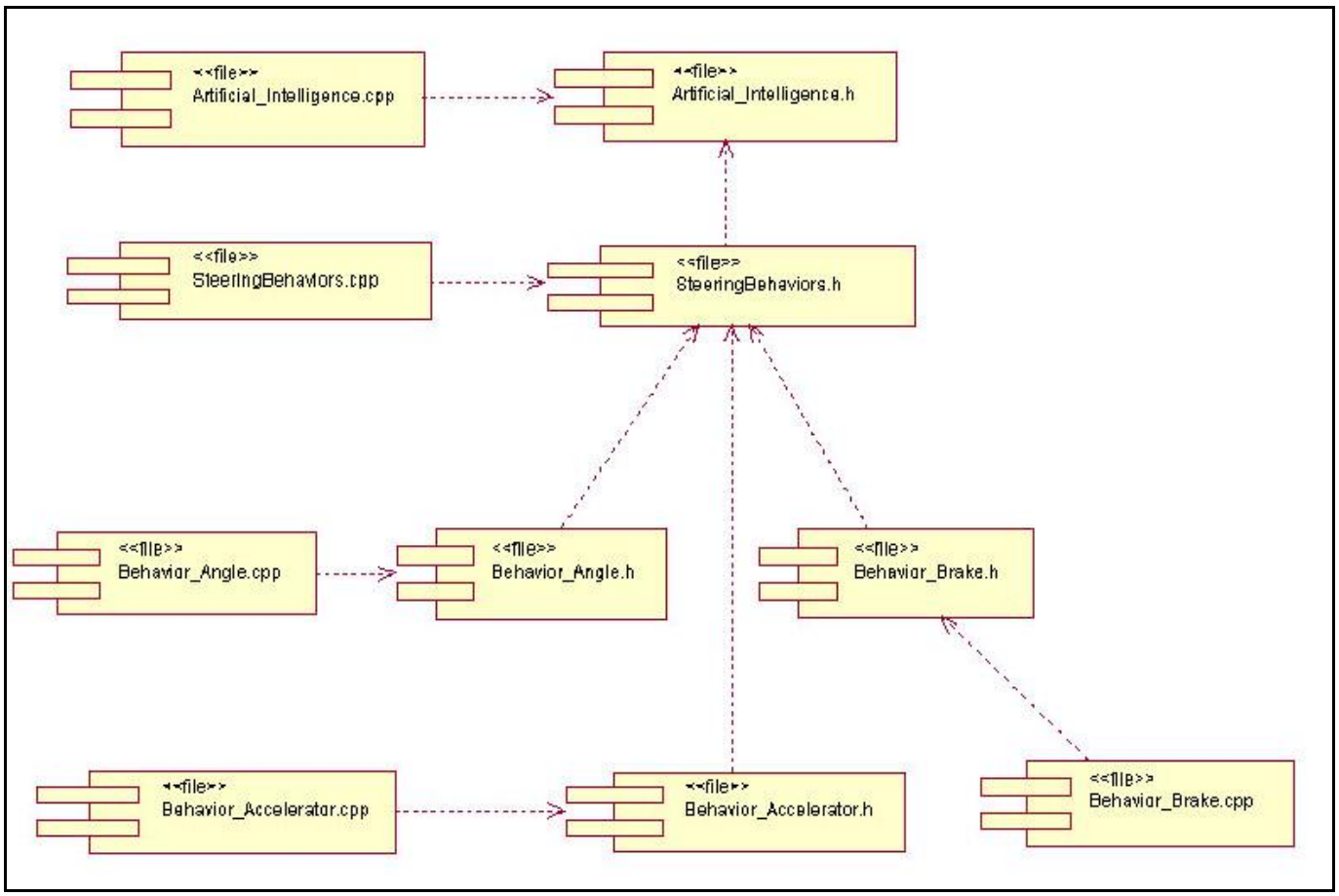


Figura # 27 Diagrama de Componentes.

4.5 Diagrama de despliegue.

Los autores del presente trabajo decidieron no incluir el mismo ya que todos los procesos se desarrollan en el nodo servidor de la aplicación, por tanto el diagrama de despliegue sería muy sencillo, representándose en él un solo nodo.

4.6 Resultados.

Con la implementación del módulo de *Steering Behaviors* se logró proveer de un comportamiento inteligente a los carros autónomos del videojuego “Rápido y Curioso”, lo que hará este juego más interesante e interactivo, garantizando mayor aceptación por parte de los usuarios.

4.6.1 Estadísticas de los resultados.

A continuación se muestran dos gráficas que recogen los tiempos de competencia de dos jugadores promedio y los del NPC.

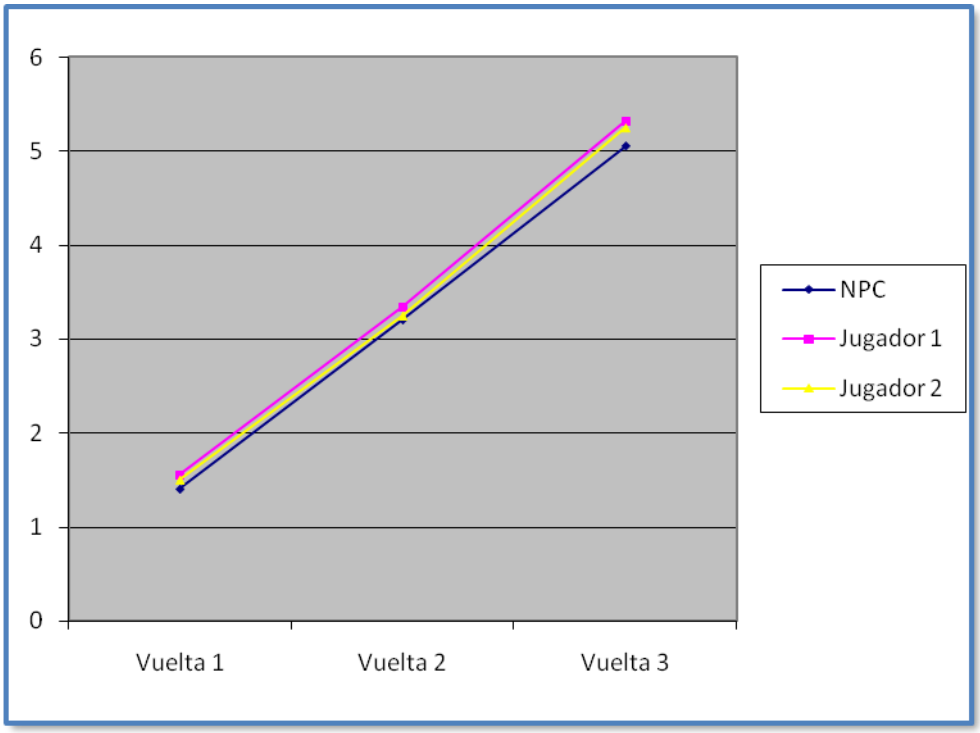


Figura # 28 Gráfica de tiempos por vuelta de los competidores.

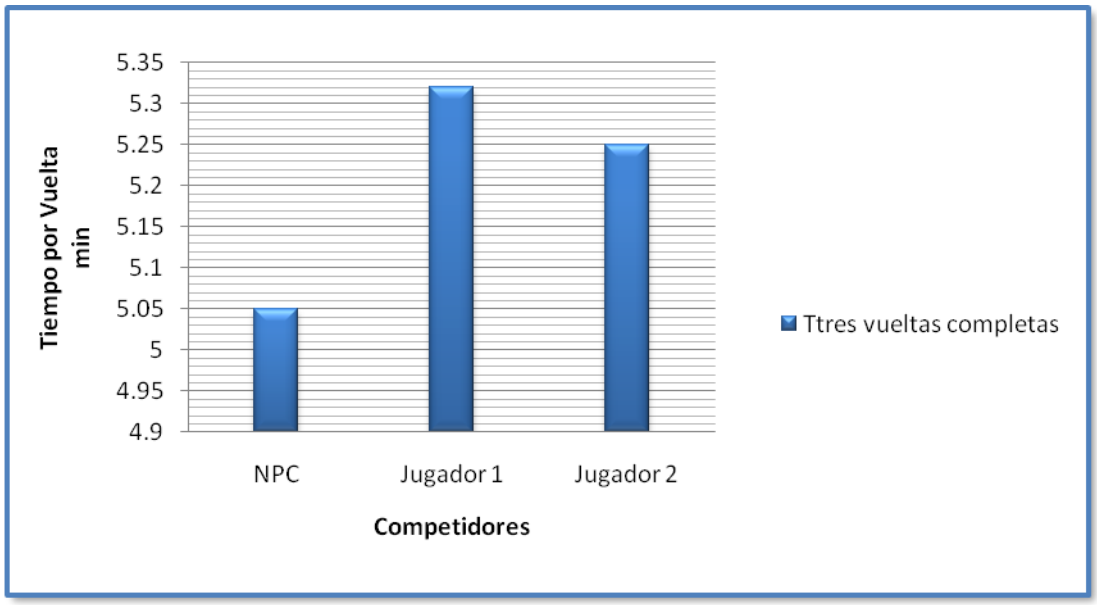


Figura # 29 Gráfica del tiempo total logrado por los competidores.

Como se puede observar el tiempo logrado por el NPC es menor al completar las vueltas que el tiempo logrado por jugadores con una habilidad media-alta.

4.6.2 Análisis de los Resultados.

El comportamiento logrado actualmente es el más eficiente, ya que aprovecha y emplea todas las potencialidades del auto para darle la mayor rapidez y destreza al mismo.

En las rectas el auto alcanza la mayor velocidad permisible (ver Figura # 30) y el frenado en las curvas se calibró de manera que el carro pudiese tomar dichas curvas a la mayor velocidad posible (figuras # Figura # 31 y Figura # 32**Error! Reference source not found.**).

Figura # 30 Velocidad del NPC en recta.

Figura # 31 Velocidad a la que el NPC toma una curva suave.

Figura # 32 Velocidad del NPC en una curva cerrada.

Conclusiones

El diseño e implementación de un módulo de *Steering Behaviors,* resultado de este trabajo, el cual se encuentra conformado por tres comportamientos (*Behaviors*) básicos encargados de dirigir la dirección del carro, la aceleración y el frenado del mismo, recoge la definición del dominio del problema y la captura de requisitos, los cuales llevaron a la identificación de cuatro casos de uso que dirigieron el proceso de desarrollo y el diseño de un sistema de clases como solución final de nuestro módulo. Con este se garantiza que los carros autónomos se muevan eficientemente dentro del entorno, recorriendo la pista en un tiempo record de 05:05min para tres vueltas completas. Siendo este resultado una prueba palpable del realismo y eficiencia que se logra con la aplicación de esta técnica en videojuegos de carrera de autos. Dando total cumplimiento a los objetivos trazados.

Recomendaciones

- Acoplar el módulo a la nueva versión de Rápido y Curioso.
- Incorporar nuevos comportamientos y actualizar los existentes una vez mejoradas las condiciones del entorno y del Editor de Pistas.

Glosario de Términos

B:

Behavior_Angle: *Comportamiento encargado de calcular el ángulo de giro del carro.*

Behavior_Accelerator: *Comportamiento a cargo de calcular la aceleración del carro.*

Behavior_Brake: *Comportamiento que calcula la fuerza de frenado a imprimirle al auto.*

C:

Convergencia: *Valor hallado para que el ángulo calculado por el Behavior_Angle no se le aplique completamente al carro, sino que multiplicado por este ayude a que la simulación sea más real y el carro rectifique su trayectoria sin dar un giro brusco al timón de no ser necesario.*

CurrentNode: *Nodo actual por el que se encuentra o pasó el carro.*

D:

DirectionCar: *Vector que representa la dirección del carro.*

DirectionNodes: *Vector entre los nodos CurrentNode y Target.*

G:

Grafo de caminos: *Grafo usado en entornos virtuales para brindar información acerca de la estructura del entorno y para la aplicación de algoritmos de inteligencia artificial como la búsqueda de caminos.*

I:

Inteligencia Artificial (IA): *Se denomina a la ciencia que intenta la creación de programas para máquinas que imiten el comportamiento y la comprensión humana.*

M:

Max_Position_Accelerator: *Variable que representa la máxima posición del acelerador del carro.*

N:

Nodo*: Estructura que representa una posición en el mapa o en el Grafo*.

NPC: *Un agente como una entidad que percibe y actúa sobre un entorno.*

S:

Steering Behaviors: *Comportamientos de locomoción, describen como el NPC debe moverse y cuán rápido puede llegar allí.*

T:

Target: *Nodo objetivo hacia el que se dirige el auto.*

Trak-Edit: *Software utilizado en la Realidad Virtual para editar los grafos de caminos de los mapas de la misma.*

U:

UmbralDistance: *Distancia de activación de la fuerza de frenado.*

Referencia Bibliográfica

1. *Computer Machinery and Intelligence.* **Turing, Alan.** 1950.

2**. Hodges, Andrew.** *Alan Turing: the enigma.* 1992. 0-09-911641-3.

3. **Guerenabarrena, Ines y Llano, Iurgi.** TIPOS DE VIDEOJUEGOS Y HABILIDADES. *Mundo Hogar.* [En línea] febrero de 2008. http://www.mundogar.com/ideas/ficha.asp?ID=7841.

4. **Marmad.** Los tipos de videojuegos que existen. *Gamers.* [En línea] febrero de 2008. http://www.gamers.com.mx/noticias/12556_Los_tipos_de_videojuegos_que_existen.html.

5**. Reynolds, Craig W.** Boids: Background and Update. *Craig W. Reynolds.* [En línea] enero de 2008. http://www.red3d.com/cwr/.

6. —. Not Bumping Into Things. *Craig W. Reynolds.* [En línea] enero de 2008. http://www.red3d.com/cwr/nobump/nobump.html.

7. **I.BRONSHTEIN, K.SEMENDIAEV.** *Manual de Matemáticas para Ingenieros y Estudiantes.* Moscú : Editorial MIR., 1990.

8. **Stuart J. Russell, Peter Norvig.** *Inteligencia Artificial: Un enfoque moderno.* 1996.

9. *Agentes Inteligentes: el siguiente paso en la Inteligencia Artificial.* **V. Julián, V. Botti.** s.l. : NOVATICA , junio. 1999, Vol. Edición Especial mayo.

10. *AGENTES INTELIGENTES. APLICACIÓN A LA REALIDAD VIRTUAL.* **Coca, Yuniesky.** s.l. : sin publicar, 2008.

11. **Christian Schnellhammer, Thomas Feilkas.** steeringbehaviors. *SteeringBehaviors.* [Online] diciembre de 2007. http://www.steeringbehaviors.de/theory_main.html.

12. **Buckland, Mat.** *Programming Game AI by Example.* s.l. : Wordware Publishing, 2005 . ISBN:1556220782.

13. **Hawkins, Kevin y Astle, Dave.** Chapter 20 - Building a Game Engine. *OpenGL Game Programming.*

14.3D Engines Database. *DevMaster.net.* [En línea] abril de 2008. http://www.devmaster.net/engines/list.php?start=30&fid=14&sid=0.

15. **Skilton, Frank.** 3D Engines Database. *DevMaster.net.* [En línea] mayo de 2008. http://www.devmaster.net/engines/.

16. **McGuire, Morgan.** *G3D Manual and Library source code.* [En línea] abril de 2008. http://g3d-cpp.sourceforge.net/html/index.html.

Bibliografía

- **David M. Bourg, Glenn Seeman.** *AI for Game Developers.* s.l. : O'Reilly, July 2004. ISBN : 0-596-00555-5 .

- **Jeff, Orking.** *AI Game Programming Wisdom.* s.l. : Steve Rabin, 2002. ISBN : 1-58450-077-8.

- **Buckland, Mat.** *AI Techniques for games programming.* 2002.

- *Evolving robust and specialized car racing skills.* **Togelius, Julian and Lucas, Simon M.** 2006.

- *Evolving Controllers for Simulated Car Racing.* **Togelius, Julian and Lucas, Simon M.** 2005.

-*Evolving Controllers for Simulated Car Racing using Object Oriented Genetic Programming.* **Agapitos, Alexandros, Togelius, Julian and Lucas, Simon M.** 2007.

- Evolutionary Robotics. [Autor.] Dario Floreano, Phil Husbands and Stefano Nolfi. *Handbook of Robotics.* 2007.

- *Arms races and car races.* **Togelius, Julian and Lucas, Simon M.** 2006.

- *Multi-population competitive co-evolution of car racing controllers.* **Togelius, Julian, Burrow, Peter and Lucas, Simon M.** 2007.

- Colin McRae Rally 2.0 (PlayStation). *Generation5.* [En Línea] 03 28, 2001. [Citado: 12 05, 2007.] http://www.generation5.org/content/2001/cmr2_psx.asp.

- Jeff Hannan. *Generation5.* [En Línea] 04 24, 2001. [Citado: 12 10, 2007.] http://www.generation5.org/content/2001/hannan.asp.

- **Charles, Darryl, et al.** *Biologically Inspired Artificial Intelligence for Computer Games.* 2008.

- **Munakata, Toshinori.** *Fundamentals of the New Artificial Intelligence. Neural, Evolutionary, Fuzzy and More.* 2008 (Second Edition).

- **McGonigal, Jane.** *All Game Play is Performance: The State of the Art Game.* 2005.

- **Buckland, Mat.** *Programming Game AI by Example.* 2005.

- **Autores, Colectivo de.** *AI game Programming Wisdom.* 2002.

- *ai-junkie.* [En Línea] 2002. [Cited: 12 05, 2007.] http://www.ai-junkie.com/misc/hannan/hannan.html.

- **DeLoura, Mark.** *Game Programming Gems 2.* 2002.