INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
Subdirección Académica

No Serie: TT – 0285 13 de Mayo del 2002

"VIDEOJUEGO DE SIMULACIÓN EN 3D"

Aréchiga Rocha Jorge Luis
jlarechiga@netscape.net
www.arechiga.50megs.com

Corona Avalos Yair
yair_corona@hotmail.com
www.geocities.com\yair_corona

Cruz Castro Edgar Alberto
edcruz90@hotmail.com
www.edcruz.5omegs.com

Asesor
M. en C. Víctor Márquez García
Instituto Politécnico Nacional
Escuela Superior de Cómputo

Resumen:
En este trabajo se presenta la descripción del sistema "Videojuego de Simulación en 3D" que es Simulador con graficas en tercera dimensión, multiusuario y con inteligencia artificial.
Su funcionamiento se basa en la programación y manipulación de graficas por computadora, dispositivos de entrada y sonido. Establecimiento de la comunicación entre varias computadoras para jugar en red, programación de algoritmos de inteligencia artificial para jugar contra la computadora e implementación de un simulador aplicando física y métodos numéricos.

ÍNDICE TEMÁTICO

1. ANTECEDENTES.

Los videojuegos tienen su origen en las salas de juegos, lugares donde se tienen que comprar fichas para poder jugar, pero la gente se gasta mucho dinero, la evolución en los videojuegos y su rápido crecimiento vino cuando se introdujo el primer sistema de entretenimiento casero, en este caso el Atari.

En ese momento la gente podía gozar de videojuegos casi de la misma calidad que la de las salas de juego pero sin gastar tanto dinero y en la comodidad de su hogar. Poco después, con la llegada del Nintendo, la cantidad de los títulos que se podía adquirir aumentó.

Así, la industria de los videojuegos se volvió una de las industrias más grandes del mundo debido a la gran cantidad de consumidores. Esto ha generado una intensa competencia entre los diferentes desarrolladores de videojuegos.

Para satisfacer la demanda de los usuarios, hoy en día se emplea la más avanzada tecnología y los últimos conocimientos existentes y aplicables a este campo. Dicha tecnología y conocimientos, evolucionan rápidamente no solo en el hardware sino también en software.

Los juegos más actuales tienen la capacidad de envolver al usuario en ambientes virtuales, para lo cual se emplean gráficos tridimensionales que necesitan de herramientas de diseño y de manipulación para poder ser creadas, algoritmos y programas para poder desplegar estos gráficos al usuario durante el juego, etcétera. Estos novedosos juegos para no aburrir al usuario con respuestas monótonas emplean algoritmos complejos de inteligencia artificial que incrementan la dificultad y la predecibilidad del juego.

La necesidad de los usuarios de divertirse más, implica poder competir contra otros usuarios, por lo cual los desarrolladores han logrado que los videojuegos sean multijugador, ya sea pudiendo conectar varios dispositivos a una consola o pc, o pudiéndose jugar en red.

Para que un juego pueda competir con otros en la actualidad debe cumplir con buenas gráficas, buenos algoritmos de I.A., ser multijugador y tener una historia novedosa o por lo menos muy buena. Aunque no solo los juegos complejos son los más divertidos, si es una tendencia que la complejidad de estos, no solo en su creación sino también en su jugabilidad, aumente día con día.

En los últimos videojuegos que se han lanzado al mercado, ha aparecido un nuevo campo que los videojuegos antiguos generalmente no integraban, es la simulación, en varios de los juegos actuales puedes observar vehículos chocando objetos rebotando y le dan un enorme realismo al videojuego, lo que lo hace ver más robusto y mejor diseñado, son videojuegos en los cuales el jugador siente que en realidad esta interactuando con los que se encuentra a su alrededor.

2. ANÁLISIS.

2.1 DIAGRAMA DE CONTEXTO.

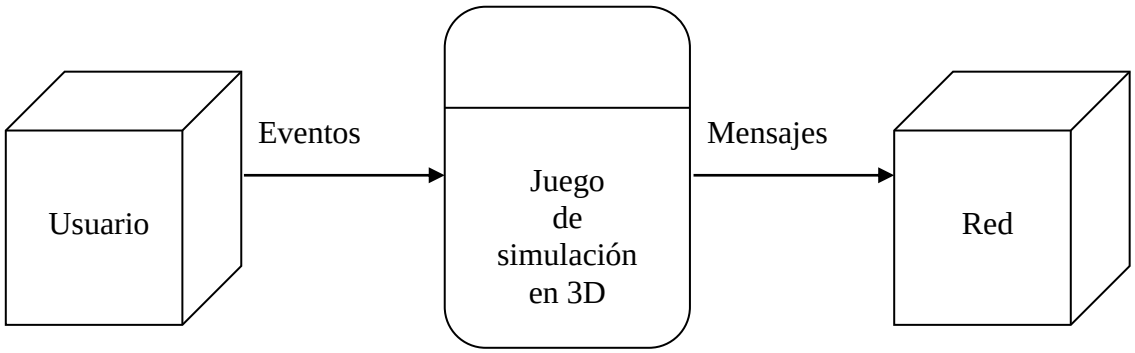


Figura 2.1

Aquí se muestra la interacción del usuario por medio de comandos con el juego, y si se necesita, como sería el caso de la opción de multijugador, la interacción con otros juegos en una red, por medio de mensajes hacia la red.

2.2. DIAGRAMA DE NIVEL 1.

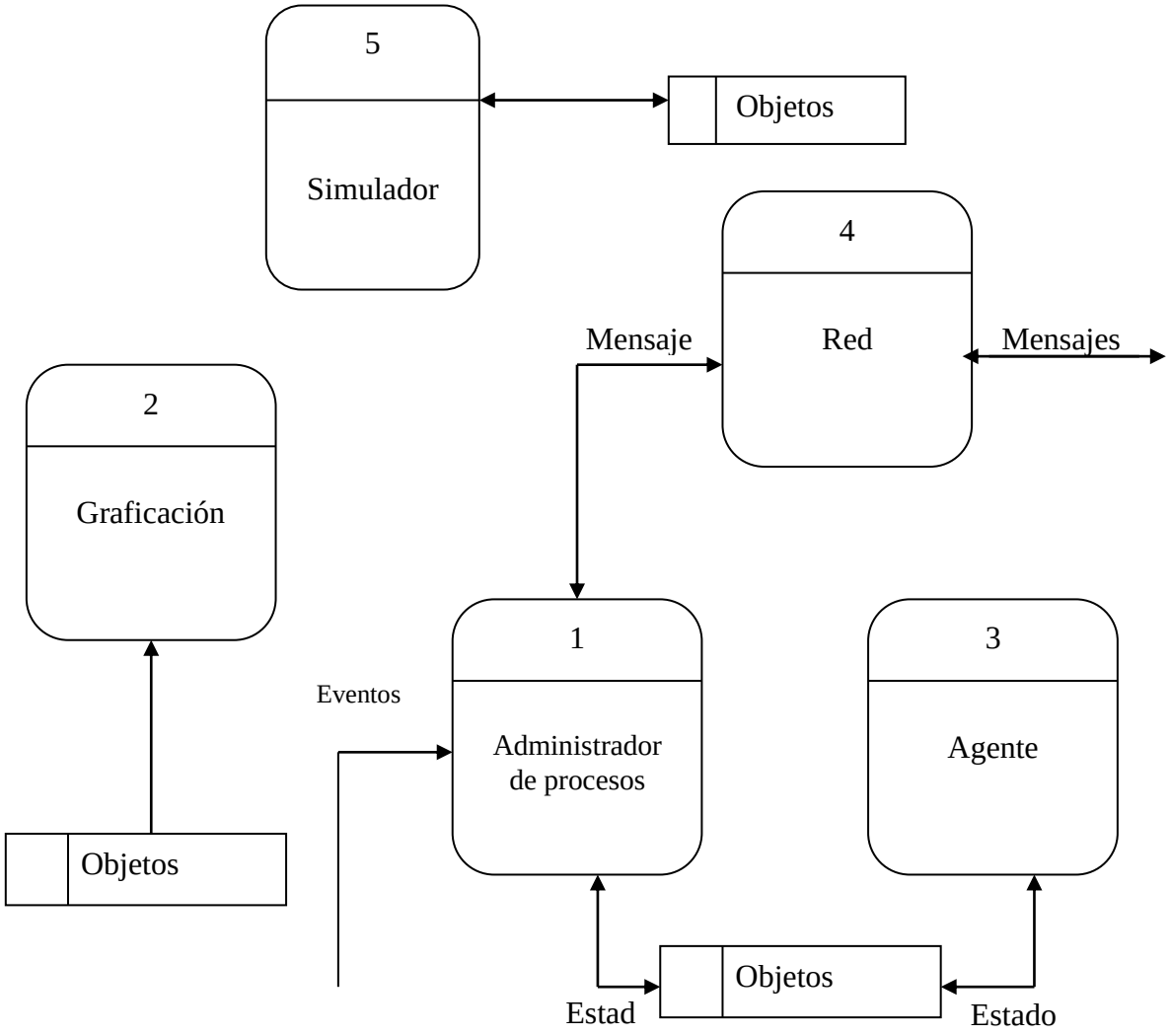


Figura 2.2.1

En este diagrama se pueden observar los módulos principales que compondrán al sistema:

1. Administrador de procesos.- Encargado de llevar el control de los otros procesos, y de los comandos que emita el usuario, así como del juego en general.

2. Graficación.- Su única función es reflejar el mundo donde se está desarrollando el juego así como las acciones que se toman en este, a la pantalla del usuario.

3. Agente.- Su tarea es tomar las decisiones de la computadora, para poder jugar en modo de usuario contra computadora.

4. Red.- Proceso que tiene como único objetivo transmitir los mensajes del administrador de procesos y recibir los de otros juegos.

5. Simulador.- Se encargara de representar en ecuaciones a nuestros modelos y al evaluarlas les dará un sentido físico real.

Objetos.- El conjunto de objetos que se encuentran en el mundo como vehículos o disparos y que tienen posiciones y estados.

2.3. ADMINISTRADOR DE PROCESOS.

Objetivo: Administrar todos los demás procesos al comunicar las acciones de cada uno de ellos, detecta colisiones entre objetos, recibe los comandos del usuario y envía mensajes al módulo de red.

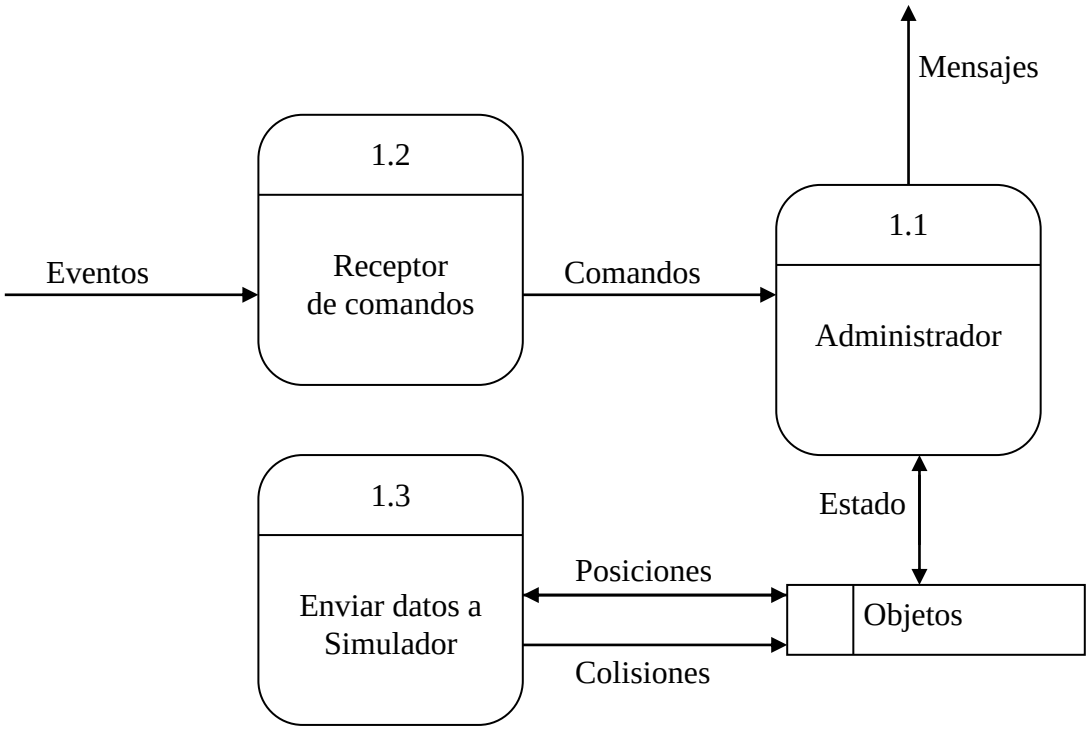


Figura 2.3.1

Requerimientos: Procesador a 450Mhz. 64 MB en RAM. Windows 95.

Flujos:

- Eventos: Acciones del usuario sobre el teclado, mouse o control y que recibe el sistema operativo, el tipo de dato de los eventos es UINT.

- Comandos: Tipo de dato int que genera el receptor de comandos para el administrador, a partir de un evento recibido.

- Mensajes: Son de tipo int y corresponden a códigos que manda el administrador a las demás instancias del juego para sincronizarse, actualizar o mencionar algún cambio en el juego.

- Posiciones: De tipo flotante, define las coordenadas del objeto dentro del mundo.
- Colisiones: Corresponden a diferentes tipos de colisiones entre objetos, como un choque o un disparo acertado, se identifican con enteros.

- Estado: Se refiere al estado del objeto, y por lo tanto engloba las posiciones y colisiones de este, en la implementación, este tipo de flujo será cambiado por dos flujos: posiciones y colisiones.

1.1 Administrador: Se encarga de modificar el estado de los objetos de acuerdo a los comandos generados por el receptor de comandos o provenientes del módulo de red, una vez que el detector de colisiones a validado o invalidado los movimientos del usuario, el administrador decide comunicar o no estos eventos al módulo de red, y después cederá tiempo al agente para que emita una respuesta y al módulo de graficación para que cumpla con su trabajo.

1.2 Receptor de comandos: Proceso que atiende los eventos generados por el usuario y los interpreta para enviárselos al administrador como un comando.

1.3 Enviar datos al Simulador: Se encarga de recibir la información del usuario, es decir de las entradas de los dispositivos como teclado y joystick, además de recibir la información de los agentes inteligentes, esta función preparar estos datos y los envía al simulador.

2.4. GRAFICACIÓN.

Objetivo: Graficar en pantalla el mundo, conformado por los objetos que se tienen en este y desde la perspectiva del jugador.

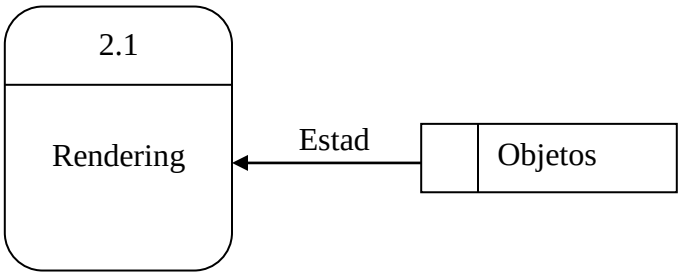


Figura 2.4.1

Requerimientos: Tarjeta de video a 16 bits, con 8 MB de memoria en video.

Flujos:

- Estado: Se refiere al estado del objeto, y por lo tanto engloba las posiciones y colisiones de este, en la implementación, este tipo de flujo será cambiado por dos flujos: posiciones y colisiones.

2.1 Rendering: Único proceso en este módulo, se encarga de dibujar o desplegar en pantalla la perspectiva del mundo que tiene el jugador en particular.

2.5. AGENTE.

Objetivo: Controlar un tanque para competir contra el usuario.

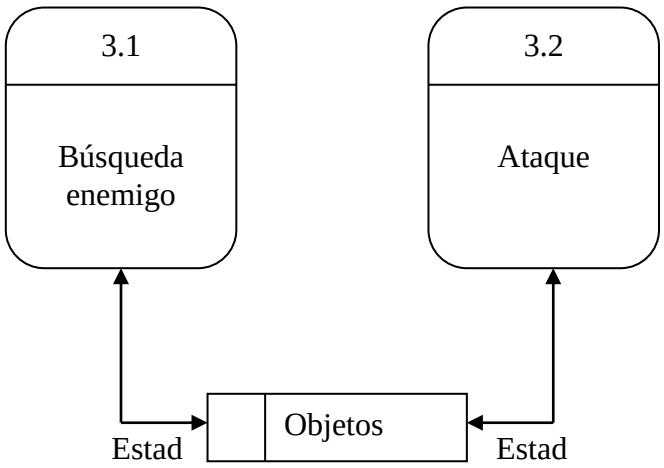


Figura 2.5.1

Requerimientos: Procesador a 450Mhz. 64 MB en RAM.

Flujos:

- Estado: Se refiere al estado del objeto, y por lo tanto engloba las posiciones y colisiones de este, en la implementación, este tipo de flujo será cambiado por dos flujos: posiciones y colisiones.

3.1 Búsqueda enemigo: Su misión es localizar al enemigo en el escenario para poderse colocar a una distancia y posición desde donde se le pueda atacar al rival, usando posiblemente el método de división de cuadrantes.

3.2 Ataque: Su función es lograr colocarse en dirección del contrario a fin de poder acertarle un disparo, usando probablemente lógica difusa.

2.6. RED

Objetivo: Transmitir los mensajes recibidos del administrador hacia las diferentes instancias del juego en la red.

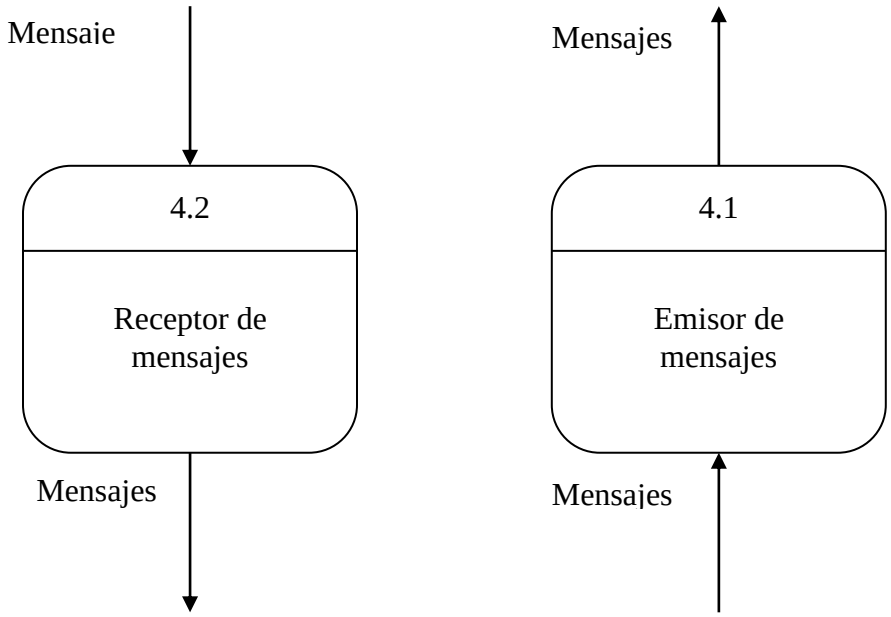


Figura 2.6.1

Requerimientos: Tarjeta de red y protocolo TCP/IP.

Flujos:

- Mensajes: Son de tipo int y corresponden a códigos que manda el administrador a las demás instancias del juego para sincronizarse, actualizar o mencionar algún cambio en el juego.

4.1 Emisor de mensajes: Encargado de recibir el mensaje del administrador que desea enviar el mensaje y de enviarlo a los demás juegos que estén en la misma partida.

4.2 Receptor de mensajes: Recibe los mensajes de otros módulos de red, y los transmite al administrador.

2.7. SIMULADOR.

Partiendo de funciones matemáticas, representaremos modelos de la realidad con modelos tridimensionales, y las propiedades de estos serán evaluadas, para conocer si los modelos se están moviendo, están colisionando y de esta forma poder acercar a nuestros modelos a lo que sucede en la realidad.

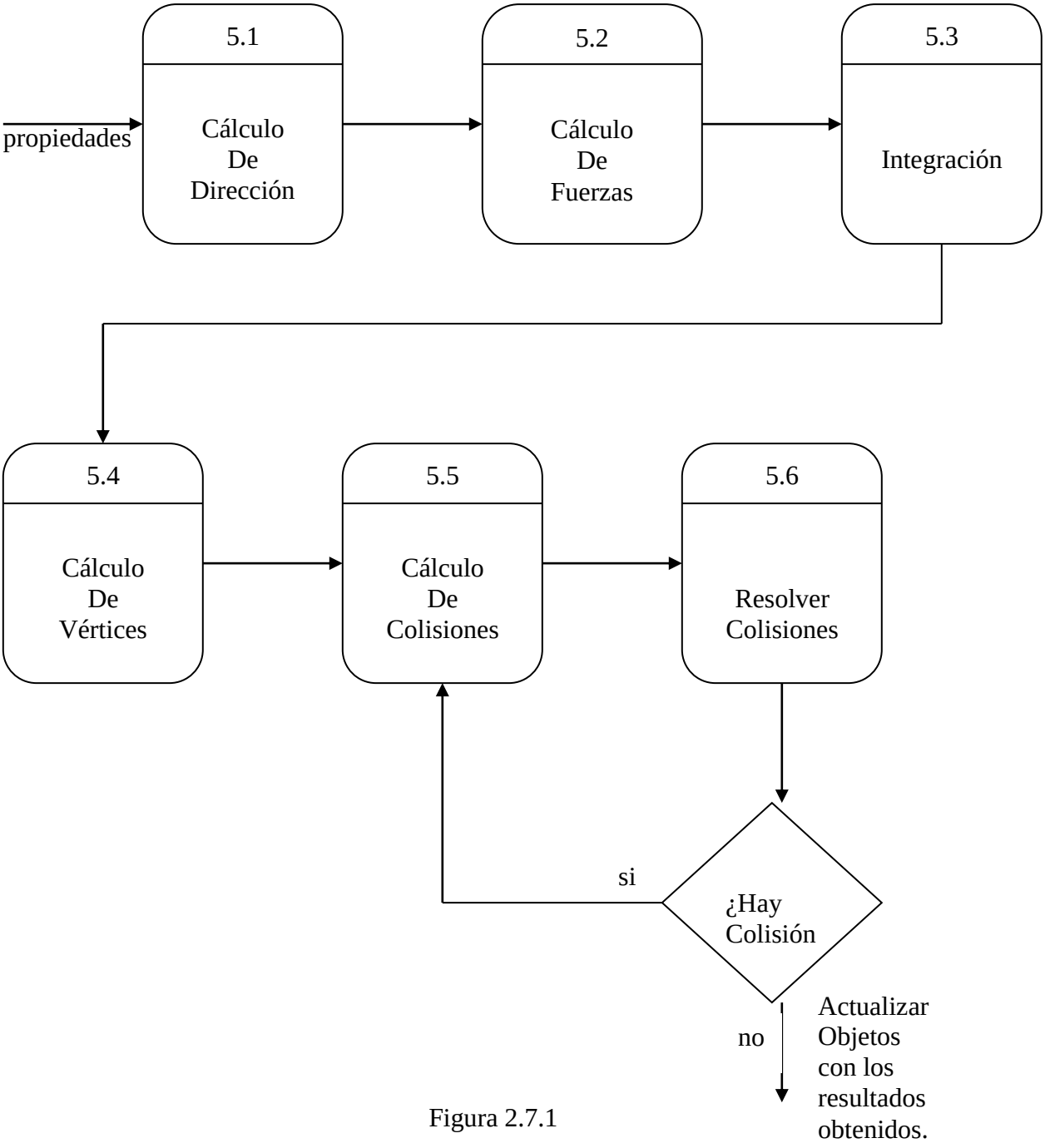


Figura 2.7.1

5.1 Calculo de dirección: Esta encargada de recibir los movimientos realizados por el jugador (Persona o Agente Inteligente), Al recibir los datos, los actualiza y los prepara para ser evaluados por las siguientes funciones.

5.2 Calculo de Fuerzas: Esta función recibe los datos procesados por el calculo de dirección y evalúa que fuerzas están provocando que el vehículo controlado por el usuario se mueva o rote, estas fuerzas pueden ser la gravedad ,aceleraciones provocadas por el usuario, fricción, etc.

5.3 Integración: Principalmente los datos que obtenemos son aceleraciones y fuerzas y conocemos la posición y rotación inicial de los objetos, necesitamos integrar estas aceleraciones y fuerzas para obtener las nuevas posiciones y rotaciones, esto se explicara mejor en la sección Simulador.

5.4 Calculo de vértices: Nos apoyamos de vértices o puntos que envuelven a nuestro objeto para realizar el calculo de colisiones, estos vectores son actualizados después de la integración para que después puedan ser evaluados por el calculo de colisiones.

5.5 Calculo de colisiones: Esta función contiene un conjunto de funciones preparadas para evaluar los datos de los objetos interactuando con los demás objetos y con el escenario tridimensional que se este utilizando, nos dirá que objeto esta colisionando y con quien o con que esta colisionando.

5.6 Resolver Colisiones: Esta Función se encarga de recibir la respuesta de el calculo de colisiones, realiza una evaluación de los objetos que están colisionando y calcula los impulsos que provocaran que el objeto cambie de dirección. De la respuesta de esta función se crea un ciclo ya que debemos saber si después de haber obtenido el resultado, todavía hay colisión o si ya no hay, si todavía hay colisión volvemos a evaluar la función en el calculo de colisiones, si no hay colisión, terminamos el calculo.

3. DISEÑO E IMPLEMENTACIÓN

3.1. GRÁFICOS, ILUMINACIÓN Y DISPOSITIVOS

3.1.1. GRÁFICOS

- Creación de un programa en modo exclusivo:

Para la realización de un programa en modo exclusivo, fue indispensable obtener los siguientes puntadores:

1. LPDIRECT3D: Apuntador que obtiene la versión actual de DirectX que esta instalada en la maquina en la que se esta ejecutando la aplicación.

2. LPDIRECTDEVICE: Apuntador por el cual tendremos una interfaz entre la tarjeta de video y la aplicación. Todas la computadoras tienen diferentes tarjetas de video, algunas mas poderosas que otras, por lo que es necesario obtener las capacidades de la tarjeta de video en la que se esta ejecutando la aplicación.

   HAL (Hardware Aplication Layer): Tarjetas de video soportan la graficación por hardware.

   HEL (Hardware Emulation Layer): Tarjetas de video que no soportan la graficación por hardware, entonces proceden a emularla.

3. WNDCLASS: Información referente al tipo de ventana que se desea crear, para crear una aplicación en modo exclusivo hay que indicarlo aquí.

- Creación de un programa en modo exclusivo que visualice un cubo en 3D

Una de las figuras básicas en la graficación en 3D es el cubo, por lo que el siguiente paso fue la realización de un programa en modo exclusivo que visualice un cubo en 3D. Para hacerlo, fue indispensable definir las caras y vértices del objeto en una estructura.

- Creación de un programa en modo exclusivo que visualice un archivo .x en 3D

Para la realización del juego es necesario dibujar figuras complejas, definirlas como fueron definidas en el programa anterior seria simplemente algo imposible, por lo que se diseño el vehículo en 3D Estudio y fue convertido a un archivo .x de DirectX. El programa obtiene la información del archivo y lo dibuja tal cual como fue diseñado en 3D Estudio.

Todos los objetos que se cargaran en el sistema, son mallas tridimensionales, formada por un conjunto puntos y estos a su vez forman triángulos, la unión de todos estos triángulos formara la malla que representara al modelo, las siguientes imágenes muestran un ejemplo de malla que se puede cargar en el videojuego.

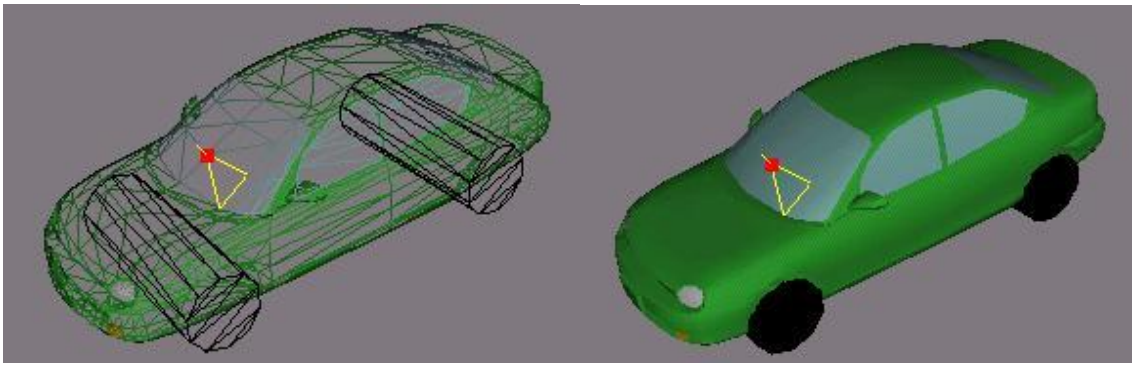


figura 3.1.1

La imagen de la izquierda, muestra el modelo de un vehículo formado por polígonos (triángulos), el punto es uno de los vértices que forman el objeto el triangulo marcado también es uno del conjunto de triángulos que le dan forma al vehículo.

La imagen de la derecha, es el mismo vehículo, la diferencia esta en que al segundo se le han aplicado algoritmos que se encargan de rellenar estos triángulos con el color que se desee, pero no solo se le aplican algoritmos que rellenan triángulos, si así fuera, el vehículo se vería opaco y sin ninguna variación en el color, reflejo, etc. Para que el modelo se vea mas real, es indispensable aplicar algoritmos de iluminación y además de esto se aplican algoritmos de sombreado, DirectX utiliza principalmente el algoritmo de pong y dado que devuelve buenos resultados, no realizaremos cambios.

De lo que se encarga esta herramienta de sombreado, es que utilizando como parámetros las normales de los vértices (se muestran en la figura siguiente) y los parámetros de iluminación, realiza una interpolación en cada cara o polígono del modelo, esta interpolación determina la intensidad y la iluminación de cada punto dentro de los triángulos que forman el modelo, esto permite que en la imagen de la derecha se note que una fuente de luz esta incidiendo en el modelo.

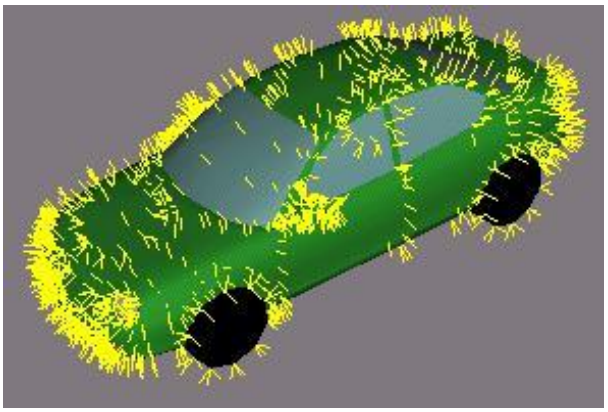


Figura 3.1.2

Cada línea que sale de cada uno de los vértices del modelo es su normal.

Al inicializarse el videojuego, el usuario tendrá la oportunidad de seleccionar el modelo que quiere utilizar y el escenario en el que quiere jugar, el usuario si lo desea podrá diseñar sus propios vehículos o escenario en el programa de diseño que desee, y cumpliendo con algunos parámetros que se mencionaran

Mas adelante, podrá cargar estos modelos en el videojuego.

El sistema de coordenadas que se maneja en Direct3D es el de la regla de la mano izquierda

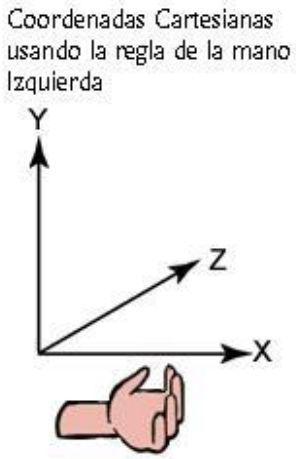


figura 3.1.3

Para la creación de escenarios y vehículos se tendrá que tener en cuenta esta configuración.

3.1.2. INTEGRACIÓN DE UNA FUENTE LUMINOSA.

Para la obtención del mayor realismo posible, se inserto una luz en el programa, dicha luz se obtiene con un apuntador del tipo D3DLIGHT8, la cual tiene la posición de un punto en tres dimensiones, su rango, etc.
A la hora de aplicar el render de la escena, se visualiza el reflejo de la luz en nuestro objeto que fue cargado de un archivo .x

3.1.3. DISPOSITIVOS DE ENTRADA (JOYSTICK).

Una de las características principales del programa, es su capacidad de ser jugado con Joystick, para lograr esto, es indispensable obtener dos apuntadores:

1. LPDIRECTINPUT8: Interfaz entre la aplicación y el Joystick.

2. LPDIRECTINPUTDEVICE8: Apuntador que contendrá la información de las capacidades del Joystick, es decir, cuantas palancas y botones tiene.

Primeramente la aplicación enumera los dispositivos, con esto se especifica por medio de que puerto esta conectado el Joystick, posteriormente se establece un tiempo de refresco, este tiempo esta establecido en segundos, con esto cada cierto tiempo se obtendrá la información del Joystick.

El jugador tendrá la oportunidad de reconfigurar los botones de su joystick al inicial el videojuego, esto permitirá que el jugador utilice cualquier tipo de dispositivo, que lo inicialice en Windows y ya que este lo haya detectado, al iniciar el videojuego, este también lo reconocerá.

3.2. SONIDO.

El sonido del juego es sonido en 3D, es decir, cada objeto representa una fuente de sonido, y el jugador escuchará los sonidos como si estuviera en el juego. Para hacer esto tomamos en cuenta que en el mundo real la percepción de un sonido en el espacio se afecta por distintos factores, el más importante de ellos es la posición. Algunas pistas que nos dan los sonidos por ellos mismos para entenderlo son las siguientes:

- Volumen del sonido. Mientras una fuente de sonido se aleja más del que la escucha su sonido decrece proporcionalmente. Este fenómeno es conocido como rolloff o atenuación.

- Diferencia de intensidad interaural. Un sonido emitido por una fuente proveniente de la derecha del que escucha suena más fuerte en el oído derecho que en el oído izquierdo.

- Diferencia de tiempo interaural. Un sonido emitido por una fuente a la derecha del que escucha llegará al oído derecho antes que al oído izquierdo. La duración de este retraso es aproximadamente un milisegundo.

- Amortiguación. La forma y orientación de las orejas causa que el sonido que proviene de detrás del que escucha está ligeramente amortiguado comparado con los sonidos que viene de frente. Además, si un sonido viene de la derecha, cuando el sonido llegue al oído izquierdo estará amortiguado tanto por la masa de la cabeza como por la orientación del oído izquierdo.

Aunque estas pistas no son todas las que la gente usa para discernir la posición de un sonido, son los principales y las que se implementan en DirectAudio, herramienta que se usará para la programación del sonido con ayuda de la interfaz DirectSound.

El sonido se lleva al programa en la clase Sonido, en la cual se definirán las posiciones tanto de las fuentes como del que escucha, las intensidades de los sonidos, los sonidos que se usarán y la velocidad del objeto que emite el sonido.

La velocidad también se toma en cuanta por que la interfaz DirectSound también lo toma en cuenta para simular el efecto Doppler ya que cuando la fuente de sonido y el que escucha están en movimiento relativo con respecto al medio material en el cual el sonido se propaga, la frecuencia de las ondas escuchadas es diferente de la frecuencia de las ondas emitidas por la fuente.

La clase Sonido está instanciada en el programa por la variable s, y contiene las siguientes funciones.

- Inicializar(). Se encarga de inicializar los objetos de sonido de DirectX.

- int Play(CHAR[]). Su función consiste primeramente en revisar si el objeto s

(instancia de la clase Sonido en el programa) tiene espacio para reproducir el sonido que se le pasa de parámetro, si lo tiene, asigna los recursos necesarios y devuelve el número de buffer donde se encontrará ese sonido, para que cada objeto que solicito el sonido, lo guarde para futuras operaciones. Inmediatamente después empezará a tocar el archivo hasta que se le de la instrucción de detenerlo.

- Stop(int). Al pasar como parámetro el número de buffer donde se encuentra el sonido que queremos dejar de reproducir, lo detendrá y liberará los recursos referentes a este buffer para que pueda ser asignado posteriormente.

- Maximoyminimo(float max, float min, int buffer). Indica la magnitud del sonido con respecto a que distancia cubrirá este.

- Cambiarposicionfuente(float x, float y, float z, int buffer). Cambia la posición de la fuente en el espacio 3D al pasar como parámetro el buffer donde se encuentra esta fuente.

- Cambiarposicionreceptor(float x, float y, float z). Cambia la posición del receptor, en este caso el jugador.

- Actualizar(). Actualiza todas las operaciones como son: intensidad de los sonidos, posición de las fuentes de sonido y posición del receptor, todas de una vez para agilizar el proceso.

- Terminar(). Libera todos los recursos asignados al sonido.

3.3. RED.

Ya que el juego ofrece la capacidad de ser multiusuario, se crea la clase de Red. Esta clase nos proveerá de las interfaces suficientes para poder crear sesiones multijugador y soportar el juego en red, para esto la clase se ayuda del objeto DirectPlay.

En los juegos generalmente se usan dos topologías: cliente – servidor y peer to peer.

3.3.1 CLIENTE - SERVIDOR

- Cliente – Servidor. En esta topología, un jugador es el servidor y los demás son clientes. Los clientes solo se pueden comunicar con el servidor, así que el servidor es el encargado de enviar a todos los jugadores el mensaje de algún cambio en el juego. El servidor tiene varias responsabilidades, entre las cuales se encuentra el servir como canal de comunicación entre las computadoras, lo cual reduce el tráfico de la red, con lo cual se obtiene una ventaja, el incremento de tráfico en la red es linear al número de jugadores que se encuentran en la partida. Por esto, esta topología es buena para juegos donde participen muchos jugadores. Otra ventaja de esta topología es que el proceso encargado de mantener el universo del juego, no consume tiempo de procesamiento en los usuarios, tan solo en el servidor ya que este es el encargado de llevarlo a cabo.

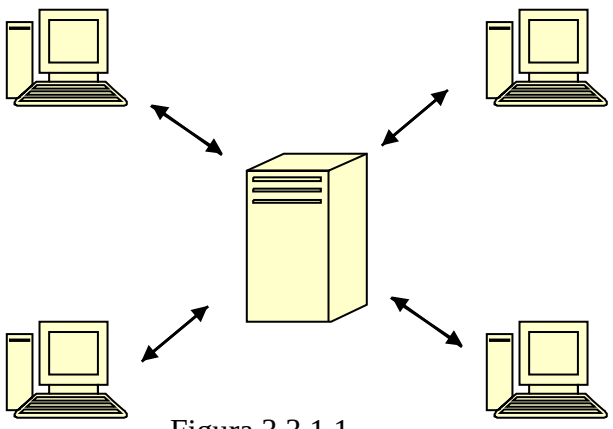


Figura 3.3.1.1

3.3.2. PEER TO PEER

- Peer-to-peer. En esta topología, cada jugador se comunica directamente con los otros clientes, así que si un jugador se mueve este deberá mandar un mensaje por cada jugador conectado así que el número de mensajes enviados crece geométricamente conforme crece el número de jugadores. Como en esta topología no existe el servidor, entonces un jugador o computador será el encargado de ser el host (anfitrión), el cual llevará a cabo ciertas tareas de logística como es el recibir a nuevos jugadores. Los juegos bajo esta topología tienen la ventaja de que aun cuando el host se caiga o salga del juego, este puede continuar si otro jugador toma su función. A pesar de que el número de mensajes enviados crece más rápido que en una topología Cliente – Servidor, esta topología puede alcanzar para juegos donde participen hasta 20 o 30 personas dependiendo el ancho de banda.

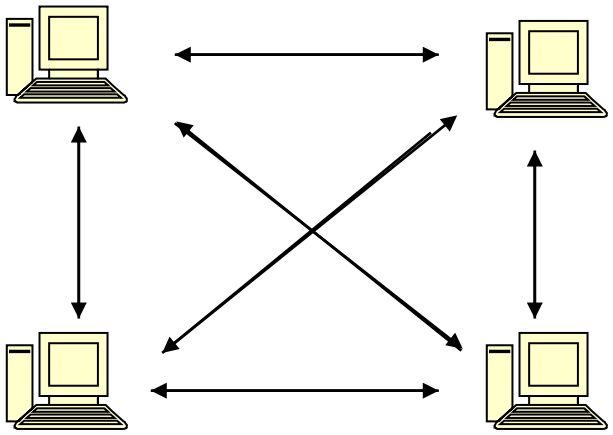


Figura 3.3.2.1

Tomando como base las características anteriores, escogimos la topología peer to peer, ya que el número de participantes que puede soportar con buen rendimiento esta topología es más que suficiente para nuestro juego que está pensado para un máximo de 8 jugadores y es ideal ya que aunque se caiga el host, los demás jugadores podrían seguir jugando característica que es muy importante y que pocos juegos ofrecen.

La funcionalidad de la parte de red en el juego está encapsulada en la clase Red e instanciado en la variable r dentro del programa.

La clase Red presenta las siguientes funciones:

- Red(). Es el constructor de la clase y se encarga de inicializar las variables, así como la tabla de números de jugador.

- inicializar(). Se encarga de inicializar y preparar la conexión.

- InitDirectPlay(). Se encarga de inicializar el objeto DirectPlay.

- terminar(). Libera los recursos ocupados por la red.

- WaveToAllPlayers(int mensaje, DPNID elid). Se encarga de mandar mensajes a las demás computadoras involucradas en el juego, en sus argumentos se encuentran el tipo de mensaje y el id de la computadora que envió el mensaje.

En el juego existen varios tipos de mensaje, como son: petición de un jugador para entrar en el juego, recepción de numero de jugador, aviso a los participantes del enlace de un nuevo jugador, actualización de la posición de un jugador, aviso de disparo por parte de un participante, salida de un jugador. Para cada tipo de mensaje se usa la misma estructura de datos pero cada campo se trata de diferente manera.

El siguiente algoritmo muestra la forma en que trabaja la red, pensando que la red trabaja de acuerdo a eventos como lo son enviar algún mensaje, recibir una petición de conexión y recibir mensaje.

```
Inicializar red.
Si es el host.
        Asignar números a participantes.
Si no es el host
        Recibir números de participantes.
Actualizar tabla de números.
Si hay información que enviar.
        Enviar información.
Si hay información para recibir.
        Si es una confirmación de salida.
                Actualizar tabla de participantes.
        Procesar información.
```

Asignar números a participantes. Para asignar los números el host recibe una petición de conexión, verifica si hay cupo, si no lo hay se niega la conexión, si hay cupo analiza la tabla de números y asigna un número que esté vacío.

Actualizar tabla de números. Si es el host, se enviará el número asignado con el id al que pertenece a todos los participantes, así todos tendrán la tabla de participantes y en caso de que el host salga del juego otro participante podrá tomar las funciones del host.

Procesar información. Involucra actualizar la información de los objetos correspondientes a las máquinas que mandan los mensajes.

3.4 SIMULADOR

En la actualidad los videojuegos se han hecho cada vez gráficamente mas realistas esto gracias al avance en el desarrollo de nuevos dispositivos de Hardware capaces de procesar funciones de manipulación de gráficos y funciones matemáticas, todo esto embebido en el dispositivo que es la tarjeta aceleradora de gráficos. Estos gráficos debido a su realismo son capaces de envolver al usuario en el videojuego, pero además de esto hay algo mas que puede hacer al videojuego aún más sólido y más real, que es la implementación de un simulador capas de procesar información recibida por el usuario o generada por la inteligencia artificial, afectarla por propiedades del escenario en 3d generado en la PC y de esta forma poder manipular y mover el vehículo u objeto que controle el jugador a través de este escenario manejando, aceleraciones, velocidades, fuerzas, impulsos, colisiones, etc. Para de esta forma poder obtener resultados que se acerquen a la realidad.

Destacamos que el avance en los dispositivos de aceleración grafica es muy importante ya que esto ha permitido que mientras un dispositivo se encarga de procesar únicamente gráficos, el procesador de la PC se encarga de otras tareas varias entre estas el manejo del simulador, lo que permite que el sistema tenga un buen rendimiento ya que para un videojuego es muy importante que los procesos que se realicen den resultados casi en tiempo real.

3.4.1 FÍSICA

La física es una parte de las matemáticas que nos permitirá representar en ecuaciones lo que este pasando con nuestros modelos y les dará un sentido físico real, creamos ecuaciones de nuestro modelo físico y esto nos retornara ecuaciones que nos dirá el comportamiento del modelo a través del tiempo.

La Física es un enorme campo, y en este caso, solo estamos interesados en una parte llamada dinámica, que es la parte de la mecánica encargada del análisis de los cuerpos en movimiento y de ahí nos interesa otra parte mas especifica que es la dinámica de cuerpos rígidos. Pero La dinámica se divide en 2 partes, cinemática y cinética. La primera se encarga del estudio del movimiento, y se usa para relacionar el desplazamiento, la velocidad, la aceleración y el tiempo sin hacer referencia a la causa del movimiento, y la cinética se usa para predecir el movimiento causado por fuerzas conocidas o para determinar las fuerzas necesarias para producir el movimiento.

Nos apoyaremos entonces de la cinética y de la cinemática para realizar nuestros cálculos de tal forma que con la cinética nos encargaremos del estudio de las fuerzas y masas que provocan que cantidades cinemáticas cambien a través del tiempo.

Habíamos mencionado que solo estábamos interesados en la dinámica de cuerpos rígidos, esto se refiere a objetos que no cambiaran su forma a lo largo del tiempo, de tal forma que el objeto se comportara como un cuerpo de madera o metal. Se utilizaran objetos de este tipo debido a que previo a la simulación, inteligencia artificial y red, esta nuestro motor de gráficos, capas de cargar archivos generados en programas de diseño como 3D studio o Autocad, etc., estos archivos están conformados por una enorme lista de puntos o vértices,

así como del orden en el que se deben dibujar estos puntos y las imágenes y colores que contienen, en conjunto, le dan forma al objeto. Todos estos valores son constantes durante la ejecución del videojuego, la manipulación de estos vértices individualmente, así como integrarlos al simulador, requeriría de una computadora muy poderosa y bastante tiempo de procesamiento, además de que talvez sería útil solo para programas de diseño, herramientas de morphing, etc., menos para un videojuego, al menos por ahora.

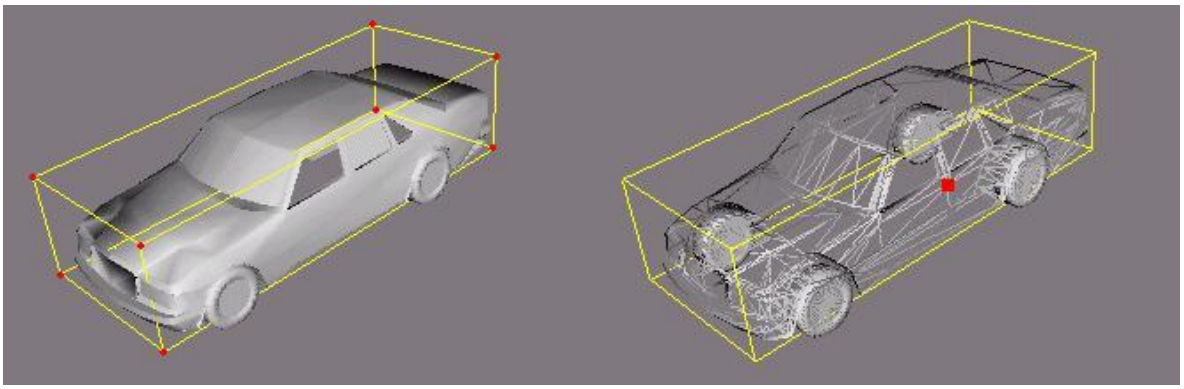


figura 3.4.1.1

La imagen de la izquierda muestra un vehículo diseñado en 3d studio, envuelto por 8 vectores, que son los que utilizaremos en el calculo de colisiones, en la derecha vemos como esta constituido este vehículo, por una enorme cantidad de puntos que unidos entre ellos forman triángulos, seria muy complicado realizar la simulación y calculo de colisiones para cada uno de ellos, como dato el modelo tridimensional de la imagen esta formado por mas de 700 vértices.

Ahora es importante dar un breve repaso de cómo calcular las cantidades cinemática y cinéticas a partir de otras que conozcamos.

3.4.2. CÁLCULOS LINEALES

Por facilidad, las explicaciones se realizaran en 2D en lugar de 3D, la mayoría de las ecuaciones, podrán ser extendidas a 3d con un poco de trabajo extra, pero es más sencillo realizar la explicación en 2D.

Las cantidades más básicas que podemos conocer y calcular del objeto, son la posición, velocidad y la aceleración, las tres son cantidades cinemáticas.

La posición del centro de masa de un cuerpo rígido, esta definido en 2D por el vector (X, Y), en 3D sería (X, y, Z). Sabemos que la derivada del vector posición es la velocidad de tal punto en (X, Y).

Posición = r **velocidad = v** **aceleración = a**

Velocidad $\frac{dr}{dt} = v = \dot{r}$

Si integramos la velocidad, obtenemos la posición del punto.
La aceleración es la derivada de la velocidad y la segunda derivada de la posición:

Aceleración $\frac{d^2r}{dt^2} = \ddot{r} = \frac{d\dot{r}}{dt} = \frac{dv}{dt} = \dot{v} = a$

Integrar la aceleración no dará la velocidad e integrarla 2 veces nos dirá la posición del punto.

$$a = \frac{dv}{dt} \Rightarrow dv = adt \Rightarrow \int_{vo}^{vf} dv = \int_{0}^{t} adt \Rightarrow vf - vo = a\int_{0}^{t} dt \Rightarrow vf - vo = at \therefore$$

$$vf = vo + at$$

partiendo del resultado anterior, la posición del objeto se podría calcular de la siguiente forma:

$$\frac{dr}{dt} = vo + at$$

$$\int_{ro}^{r} dr = \int_{0}^{t} (vo + at)dt \Rightarrow r - ro = vot + \frac{1}{2}at^2$$

$$r = ro + vot + \frac{1}{2}at^2$$

De esta forma podríamos seguir deduciendo algunas otras formulas que pudieran ser útiles.

Hasta aquí, tenemos lo más básico de la cinemática en 2D, pero necesitamos conocer fuerzas y masa que afectaran a nuestro objeto, aquí es donde entra la cinética y también entran algunas nuevas variables como el **momento p**, el cual se obtiene de la operación de la **masa** pos la derivada de la velocidad y la segunda ley de newton relaciona a la **fuerza** con la derivada del momento p de tal forma que:

F = fuerza **momento = p** **masa = m**

$$F = \dot{p} = \frac{dp}{dt} = \frac{d(mv)}{dt} = m\dot{v} = ma$$

Hasta aquí, conociendo las fuerzas que afectan al objeto, podríamos obtener la aceleración del objeto, dividiendo la fuerza entre la masa, y al integrar esta la aceleración, después la velocidad y por ultimo la posición del objeto.

Nuestro objeto rígido, tendrá toda su masa distribuida en todo el objeto, para lo que necesitamos hacer unos nuevos cambios.

Supongamos que nuestro objeto tiene varios puntos que definen a su masa, de tal forma que el momento total sería $\mathbf{p^t}$ este es la sumatoria de todos los momentos de todos los puntos que componen a nuestro objeto. Lo que se simplifica si usamos el centro de masa del

objeto, el cual es la combinación lineal de los vectores a todos los puntos en el cuerpo rígido por sus masas dividido entre la masa total M.

$$r^{CM} = \frac{\sum_i m^i r^i}{M}$$

utilizando esta definición, se puede simplificar el momento p que partiendo de su definición anterior para una sola partícula se puede llevar a:

$$p^T = \sum_i m^i v^i \Rightarrow p^T = MV^{CM}$$

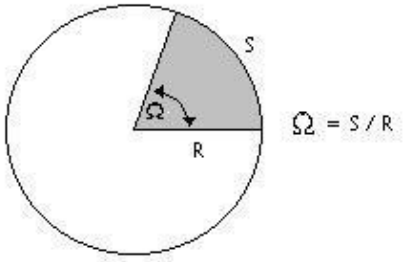
Partiendo de lo anterior, la fuerza total entonces se calcula de la siguiente manera:

$$F^T = \dot{p}^T = M\dot{v}^{CM} = Ma^{CM}$$

Con esto ya tenemos las funciones básicas para hacer cálculos lineales, pero no conocemos donde se aplican las fuerzas y no tenemos efectos angulares.

3.4.3. EFECTOS ANGULARES O ROTACIONALES

Siguiendo en 2D, necesitamos otras cantidades cinemáticas como la **Orientación Ω,** con lo que necesitamos 2 sistemas de coordenadas, uno global y que no se mueve y otro ajustado en movimiento y rotación con respecto al objeto, de tal forma que Ω es la diferencia entre ambos en radianes, esto se conoce como **desplazamiento angular**.



Mientras nuestro objeto rota en nuestro sistema de coordenadas global, nos llevara a otra cantidad cinemática que es la **velocidad angular ω** y con la derivada de esta obtenemos la **aceleración angular α**.

$$\frac{d^2\Omega}{dt^2} = \frac{d\omega}{dt} = \dot{\omega} = \alpha$$

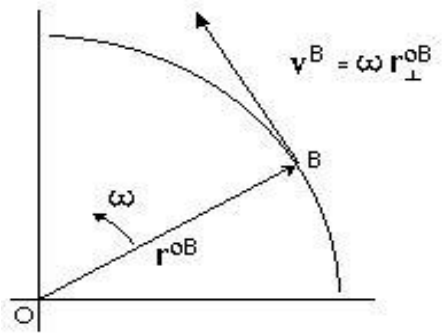
figura 3.4.3.1

La integración de aceleración angular α nos llevara a obtener velocidad angular ω y al integrar esta ultima obtendremos el desplazamiento angular u orientación Ω.

El objetivo ahora, es unir nuestras ecuaciones angulares con las lineales para que dada una fuerza sobre nuestro objeto y después de calcular la aceleración lineal y la aceleración angular, final mente integrando estas aceleraciones podamos obtener la nueva posición del objeto así como su orientación.

Primero necesitamos calcular la velocidad lineal de nuestro objeto y para esto ocupamos la velocidad angular de el. De tal forma que tomando un punto como origen "O" de tal forma que el objeto este en el origen de las coordenadas globales y este rotando pero no se este trasladando, la forma de calcular la velocidad de O a un punto B es:

$$V^B = \omega r_{\perp}^{OB}$$



Donde r representa el vector del origen a B. La velocidad de un punto de un cuerpo rotando se calcula multiplicando la perpendicular del vector del origen al punto por la velocidad angular.

Figura 3.4.3.2

Si nuestro objeto además de estar rotando, se esta trasladando, entonces consideraremos cualquier movimiento de un cuerpo rígido como una simple rotación del resto del cuerpo alrededor de ese punto.

Consideramos entonces al vector O como un simple punto de traslación y después usamos la velocidad angular ω para mantener la rotación del objeto alrededor de O, lo cual nos lleva a la formula:

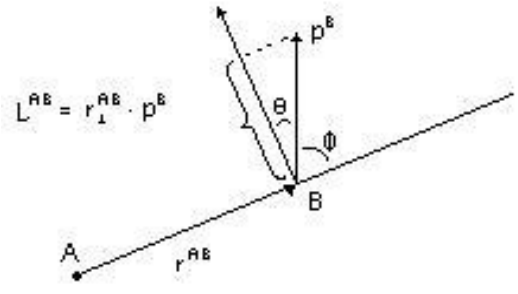
$$v^B = v^o + \omega r_{\perp}^{oB}$$

Lo que nos dice que podemos calcular la velocidad de cualquier punto dentro del objeto en movimiento tomando su velocidad lineal conocida v^0 y sumándosela a la velocidad generada por la rotación del cuerpo.

Ahora necesitamos calcular el equivalente de la fuerza angular de la fuerza lineal, primero definiremos el **momento angular L^{AB}** de un punto B acerca de otro A:

$$L^{AB} = r_{\perp}^{AB} \cdot p^B$$

A es el punto acerca del cual el momento es medido y B es el punto cuyo momento angular esta siendo medido, de tal forma que el momento angular de un punto es el producto punto de la magnitud del vector perpendicular de r con el momento lineal del punto B.



El producto punto en L^{AB} esta midiendo el coseno de θ entre la perpendicular de r^{AB} y P^B.

Figura 3.4.3.3

De la misma forma que se utilizo el momento lineal para definir la fuerza, se usara el momento angular, la derivada de esta para

definir a la fuerza angular, mejor conocida como momento de **torsión τ**.

$$\tau^{AB} = \frac{dL^{AB}}{dt} = \frac{d\left(r_{\perp}^{AB} \cdot p^{B}\right)}{dt} = r_{\perp}^{AB} m a^{B} = r_{\perp}^{AB} F^{B}$$

Esta ecuación ya usa el punto donde la fuerza fue aplicada y la derivada del momento angular no dirá entonces que tanta de la fuerza aplicada a un punto B esta haciendo rotar a este punto alrededor de A.

Este momento angular esta aplicado a un solo punto, de la misma forma en que el momento lineal se extendió para todo el objeto como momento total, extenderemos la definición de momento angular a momento angular total
El momento angular total acerca de un punto A se denotara por L^{AT}:

$$L^{AT} = \sum_{i} r_{\perp}^{AB} p^{i} = \sum_{i} r_{\perp}^{AB} m^{i} v^{i}$$

recordemos que la velocidad de un punto en términos de la velocidad angular era:

$$v^{B} = \omega r_{\perp}^{AB}$$

sustituyendo obtenemos:

$$L^{AT} = \sum_{i} r_{\perp}^{Ai} m^{i} \omega r_{\perp}^{Ai} = \omega \sum_{i} m^{i} r_{\perp}^{Ai} r_{\perp}^{Ai} = \omega \sum_{i} m^{i} \left(r_{\perp}^{Ai}\right)^{2} = \omega I^{2}$$

Aquí a aparecido una nueva cantidad I que se le conoce como momento de inercia del punto A. I^{A} es la sumatoria de las distancias del punto A a los demás puntos i al cuadrado multiplicado por la masa de cada punto.

$$I^{A} = \sum_{i} m^{i} \left(r^{Ai}\right)^{2}$$

Ahora ya que se calculo el momento angular total L^{AT} y de ahí llegamos al momento de inercia finalmente podemos unir las ecuaciones de cinética y cinemática, si diferenciamos el momento angular total, obtendremos el momento de torsión total.

$$\tau^{AT} = \frac{dL^{AT}}{dt} = \frac{d\left(I^{A}\omega\right)}{dt} = I^{A}\dot{\omega} = I^{A}\alpha$$

Si conocemos el momento de torsión sobre el cuerpo podemos encontrar su aceleración angular dividiendo el momento de torsión entre el momento de inercia, si conocemos la aceleración, con una integración de esta podremos encontrar la velocidad angular y de ahí con otra integración encontraremos la orientación.

Utilizando las ecuaciones anteriores, podremos realizar todos los cálculos necesarios para encontrar la posición del objeto y la orientación del objeto, que son las propiedades que requiere nuestro motor de graficación para actualizar el modelo que se estará dibujando en la pantalla del monitor.

3.4.4. COLISIONES Y RESPUESTA A COLISIONES

Ya tenemos ecuaciones que le permitirán a nuestro objeto moverse en nuestro escenario, afectados por fuerzas (como la gravedad) y otros factores generados por las acciones del jugador (persona o inteligencia artificial), pero ahora falta algo muy importante, el calculo de las colisiones, sin esto, los jugadores estarían navegando en el escenario, pasarían sobre las paredes de esta como si no hubiera, se mezclarían con los objetos o vehículos de otros jugadores como si no hubiera nada. Para evitar esto, necesitamos primero, detectar cuando un vehículo choca con alguna pared o contra otro vehículo y segundo, debemos saber que hacer una vez que se ha detectado una colisión.

3.4.4.1 ESCENARIO

Primero necesitamos conocer como son los escenarios tridimensionales que estaremos utilizando en el videojuego.

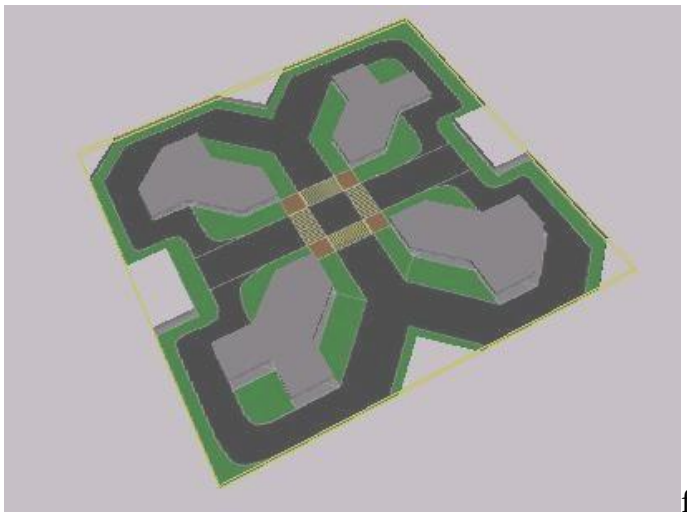


figura 3.4.4.1.1

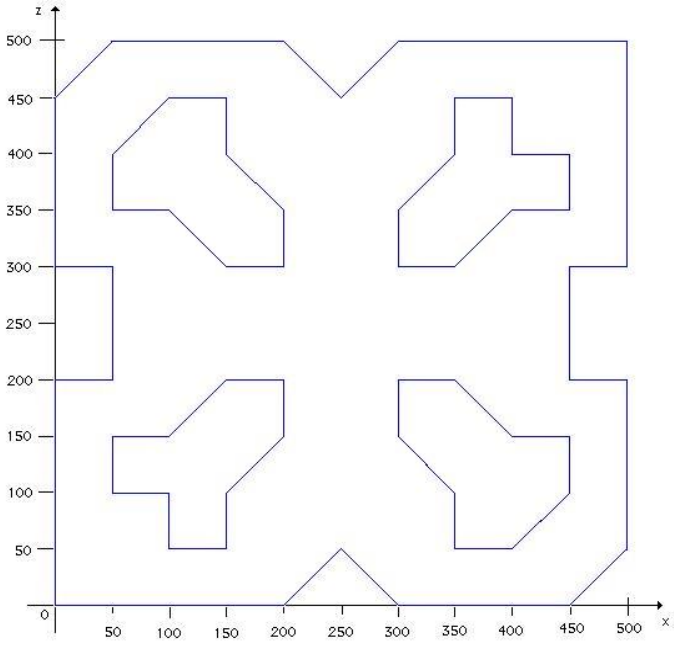


figura 3.4.4.1.2

La primer imagen del escenario (figura 3.4.4.1.1), es el modelo tridimensional diseñado en 3d studio, que cargara en la PC nuestra herramienta de graficación, la segunda imagen (figura 3.4.4.1.2), nos muestra la vista superior de este escenario, la vista superior es el plano x, z.

Cada línea del plano x, z se tomara como una pared del escenario, entonces además del modelo tridimensional, el sistema recibirá un archivo de texto que contiene las coordenada iniciales y finales de cada una de las líneas de este plano y un parámetro extra que nos permitirá decidir después la dirección de la normal de la línea.

Ejemplo:

X1	Z1	X2	Z2	Dirección
0.0	0.0	0.0	200.0	1
0.0	200.0	50.0	200.0	-1
50.0	200.0	50.0	300.0	1
50.0	300.0	0.0	300.0	1
0.0	300.0	0.0	450.0	1
0.0	450.0	50.0	500.0	-1
50.0	500.0	200.0	500.0	-1
...	...	...	...	...

Tabla 3.4.4.1

Así para cada una de las líneas del escenario, nuestro sistema tiene la capacidad de recibir cualquier escenario, basta con que se le envíe el modelo tridimensional y el archivo con la lista de paredes del escenario.

Una vez que el sistema a recibido estos 2 archivos, el segundo, la lista de paredes será útil tanto para el calculo de colisiones como para la inteligencia artificial, la lista puede ser muy grande, de tal forma que utilizaremos una estructura de datos especial para trabajar con esta lista de paredes, a continuación se explica brevemente como se creara esta estructura.

El sistema tiene una función encargada de leer el archivo, los datos son almacenados temporalmente en un arreglo, después se realizan cálculos con ellos que se mencionaran en la sección de inteligencia artificial.

Una vez que se tienen los datos, se crea una estructura de datos llamada Quad Tree, el cual consiste en una árbol de 4 hojas, cada hoja representara una sección del escenario, en la figura siguiente (fig. 3.4.4.1.3), se muestra el escenario con las divisiones que se generan. El nodo **raíz**, es un nodo que contiene las dimensiones de todo el escenario en base a esas dimensiones, el escenario se divide en 4 partes (A1, A2, A3, A4), cada uno de estas partes será un nodo hijo de la raíz y tendrán sus propias dimensiones, después a cada una de estas 4 partes las dividimos en 4 partes mas, de tal forma que A1 tendrá 4 hijos (B1, B2, B3, B4) y así sucesivamente hasta A4, estos nuevos nodos hijos “B” serán divididos también en 4, de tal forma que cada nodo B tendrá 4 hijos (C1, C2, C3, C4). La figura 3.4.4.1.4 es un ejemplo del árbol (quad tree)que se genera.

Una vez que el árbol se ha formado, se realizara un recorrido de todas las paredes del escenario y verificaremos en que nodo o sección de nuestro quad tree se encuentra, ya que se localizo su posición, esta línea se almacena en una arreglo que contiene cada nodo termina (las C), este arreglo tendrá las paredes que se encuentran dentro de el.

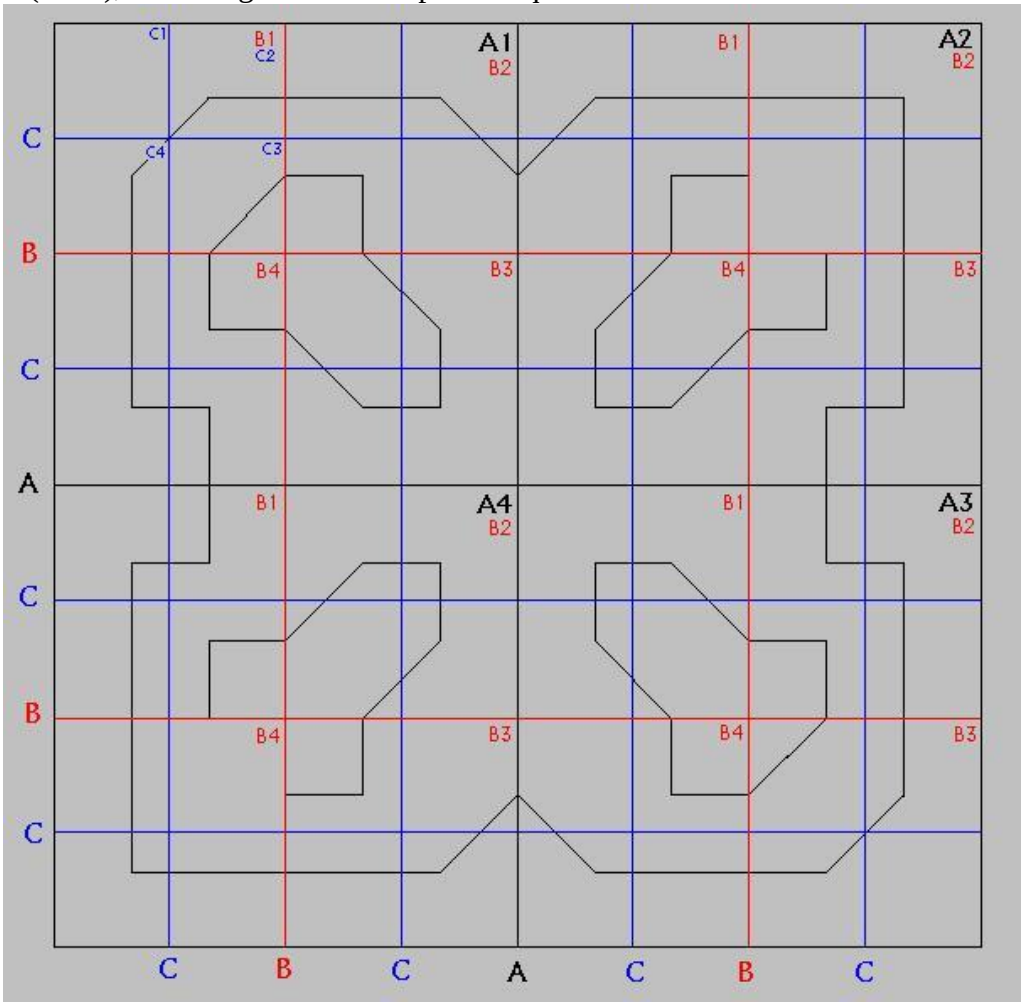


figura 3.4.4.1.3

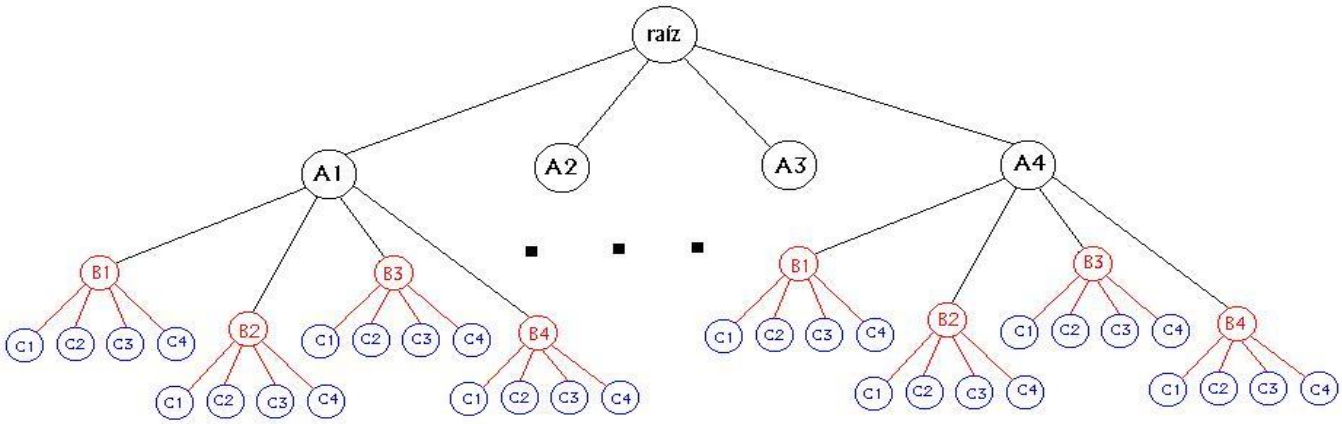


figura 3.4.4.1.4 (quad tree)

3.4.4.2. OBJETOS (VEHÍCULOS)

Los vehículos que se estarán moviendo en el escenario y que serán controlados por el usuario o por el agente, son un conjunto de caras (polígonos) que en conjunto le dan forma al objeto, seria muy complejo calcular cuando alguno de estos polígonos o los puntos que lo forman tienen una colisión contra algún objeto o contra el escenario. Por tanto una forma más sencilla (para el procesador de la maquina) sería evaluar cuando algunos puntos representativos del objeto tienen colisión contra otro objeto o contra el escenario.

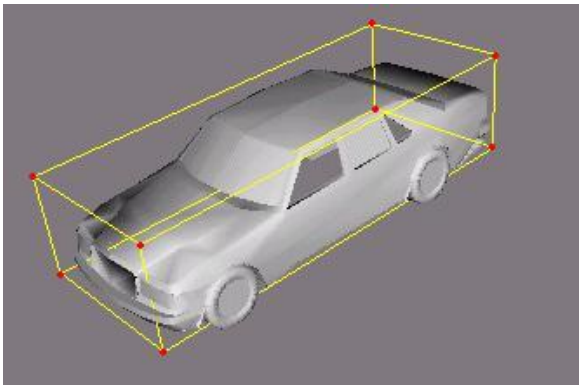


figura 3.4.4.2.1

En la figura se ilustra como 8 puntos pueden envolver a un objeto, formando un prisma rectangular, este calculo se puede mejorar si agregamos unos cuantos puntos mas a nuestro objeto, esto se ilustra en la siguiente figura.

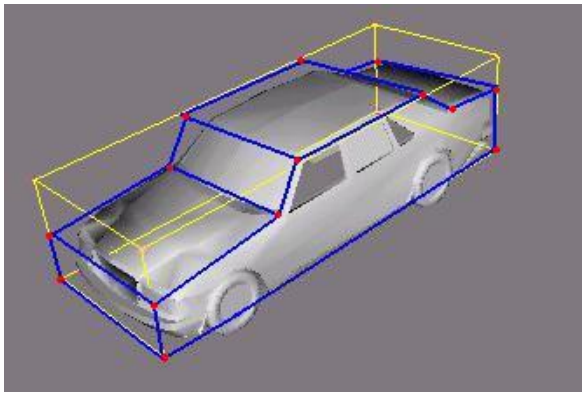


figura 3.4.4.2.2

Ahora la figura que envuelve a nuestro objeto ya no es un prisma rectangular, este nuevo grupo de puntos envuelve mejor al objeto, ya que se acerca a la forma de este.

Entonces al recibir el modelo tridimensional del vehículo, también se recibe la lista de puntos que envuelven al vehículo, estos valores estarán cabiendo durante la ejecución del videojuego cada que el vehículo se mueve, pero es mucho más rápido para la PC calcular la nueva posición de a lo mas 16 vértices a tener que calcular cientos.

Otra forma también muy útil para envolver un vehículo, es con esferas, en la siguiente figura se muestra como el vehículo puede ser envuelto en ellas.

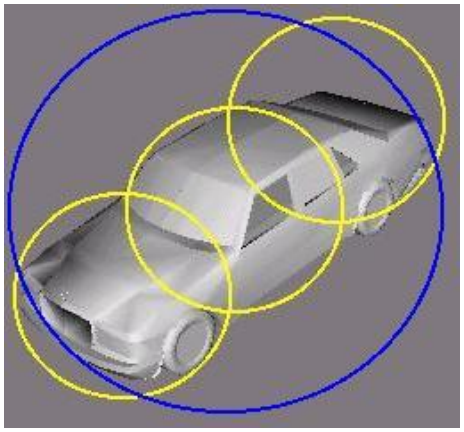


figura 3.4.4.2.3

El envolver el objeto en esferas nos es tan exacto como el envolverlo con puntos, pero mientras calcular las colisiones entre puntos y el escenario es sencillo, calcular colisiones entre objetos utilizando puntos, es bastante complicado, pero ya que no necesitamos de mucha exactitud, el envolver los vehículos con esferas para calcular las colisiones entre ellos es mas sencillo y además el cálculo es muy rápido. Junto con la lista de puntos que envuelven al objeto, también se envían los centros de las esferas que envuelven al objeto así como su radio.

Bueno ahora ya sabemos como se representan tanto el escenario como los vehículos en el sistema, ahora viene la parte matemática de cómo vamos a realizar los cálculos de las colisiones.

3.4.4.3 CÁLCULO DE COLISIONES

Partimos de que tenemos 2 objetos A y B que están a punto de colisionar en un punto P, de tal forma que el vector del centro de masa de cada objeto al punto P son: r^{AP} y r^{BP} , de tal forma que las velocidades de estos 2 puntos son v^{AP} y v^{BP}, entonces la velocidad relativa de entre los objetos A y B que denotamos como v^{AB} es:

$$v^{AB} = v^{AP} - v^{BP}$$

Es importante conocer la dirección que tomara el objeto después de la colisión, entonces necesitamos un vector n que será el vector normal de la colisión, en base a esto definimos la velocidad relativa normal como un componente de la velocidad relativa en dirección de la normal de colisión.

$$v^{AB} \cdot n = \left(v^{AP} - v^{BP}\right) \cdot n$$

Ahora: “Habrá una colisión cuando un punto de un objeto toque otro punto de otro objeto con una velocidad relativa normal negativa”, esto significa que la ecuación anterior deberá ser negativa o no abra colisión en ese punto.

Si la ecuación anterior es mayor que 0, significara que los puntos se están alejando, si es igual a 0, ni se están alejando ni se están acercando y finalmente si es menor que 0 un objeto esta penetrando a otro y debemos detenerlos de que sigan penetrando, de los que se encargara la respuesta o resolución de colisiones.

Este calculo es muy útil para saber cuando hay una colisión con el escenario, de ahí la importancia de que tuviéramos que ordenar las paredes del escenario, con el quad tree en lugar de realizar este calculo con todas las líneas del escenario, solo localizamos donde se encuentran los vértices del objeto en el escenario me refiero a los nodos del quad tree y una vez localizado el nodo en el que se encuentra el punto, basta realizar el calculo de colisiones con las paredes que se encuentren en ese nodo en lugar de hacerlo con todas las paredes.

Si queremos calcular cuando una vehículo tiene colisión con otro vehículo o queremos saber cuando este a sido impactado por una bala, nos conviene mas realizar el calculo de colisiones con esferas.

Tenemos un Objeto A que posiblemente tenga colisión con un objeto B, denotamos el radio de ambos como r^A y r^B si la distancia de los centros de masa de los dos objetos son menores que sus radios entonces tal vez haya colisión.

Denotamos d^{AB} como la distancia entre rA^{CM} y rB^{CM} entonces:

Si $r^A + r^B < d^{AB}$ existirá posiblemente colisión.

Esta sencilla operación se repetirá para cada una de las esferas que envuelve al objeto, pero con la gran ventaja de que si con la primer esfera (la más grande) no hay colisión, con el resto de las esferas tampoco la habrá (ver figura 3.4.4.2.3).

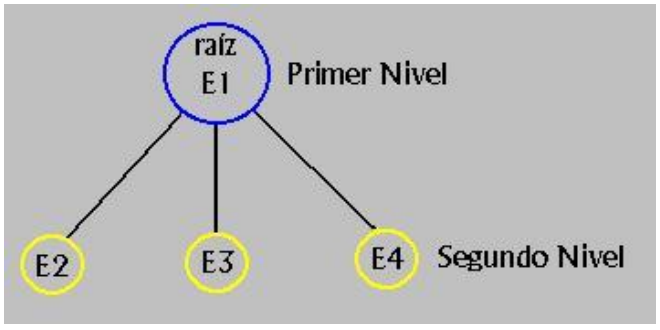


figura 3.4.4.2.4

En esta figura se ilustra que el primer nivel es ocupado por la esfera más grande que envuelve por completo al objeto, si en este nivel se detecta una colisión, para mejorar la exactitud se agrega un segundo nivel, cada una de estas esferas envuelve una parte del

objeto, se realiza el calculo entonces para cada una de las esferas del segundo nivel, si se detecta una colisión en alguna de ellas, entonces el resultado será que el objeto tuvo una colisión, el calculo se puede mejora si se requiere de mayor exactitud, agregando mas niveles, de tal forma que si se encontró una colisión en E3 de la figura 3.4.4.2.4, pasaremos a un tercer nivel en donde realizaremos el calculo con las esferas que envuelvan una parte mas pequeña del objeto y que además estén dentro de la esfera E3.

3.4.4.4 RESOLVER COLISIONES

Cuando hay una colisión, debemos aplicar una fuerza a los objetos que tuvieron la colisión, pero esto no evitara que continúen colisionando, por que la fuerza no puede cambiar la velocidad de los objetos instantáneamente, entonces utilizaremos una nueva cantidad el **Impulso**.

Un impulso puede cambiar las velocidades instantáneamente, aunque realmente si ahí un periodo de tiempo, pero es muy pequeño. Entonces mientras la fuerza cambia el momento con respecto al tiempo (la derivada), el impulso lo cambia instantáneamente, lo que provoca que la velocidad también cambie.

Para nuestros cálculos en esta parte, todas la fuerza que no provengan de una colisión quedaran fuera, como la gravedad. Necesitamos también una nueva cantidad el **coeficiente de restitución e** el cual modela la compresión y restitución del objeto después de una colisión con una valor escalar, el valor de este va de 0 a 1 si e =1 la colisión es completamente elástica, si e = 0, será completamente inelástica.

Este coeficiente relaciona a los puntos colisionando con las velocidades relativas normales de los puntos antes del choque.

$$v_2^{AB} \cdot n = -e v_1^{AB} \cdot n$$

El impulso generado por la colisión ira en la dirección de las normales de colisión, impuso es j por la normal, jn. El impulso que sentirá el objeto A será jn mientras que el objeto B sentirá –jn, lo dice la tercera ley de newton.

Las primeras ecuaciones que escribimos relacionan a las velocidades de los centros de masa de los objetos con el impulso.

$$v_2^A = v_1^A + \frac{j}{M^A} n$$

$$v_2^B = v_1^B - \frac{j}{M^B} n$$

hasta aquí, los objetos no podrán rotar, entonces la velocidad del centro de masa es la misma que la de todos los puntos, entonces reemplazamos a v^{AP} con v^{A} y lo mismo para el objeto B.

Recordando la velocidad relativa $v^{AB} = v^{AP} - v^{BP}$

$\left(v_2^A - v_2^B\right) \cdot n = -e\left(v_1^A - v_1^B\right) \cdot n$ sustituyendo v_2^A y v_2^B en esta ecuación obtenemos:

$$v_1^A \cdot n + \frac{j}{M^A} n \cdot n - v_1^B \cdot n + \frac{j}{M^B} n \cdot n = -e v_1^{AB} \cdot n$$

La ecuación anterior se puede simplificar para ponerse en términos del impulso entonces:

$$j = \frac{-(1+e) v_1^{AB} \cdot n}{n \cdot n\left(\frac{1}{M^A} + \frac{1}{M^B}\right)}$$

ahora que tenemos el impulso, podemos colocarla en las ecuaciones

$v_2^A = v_1^A + \frac{j}{M^A} n$ y su similar para B y podremos encontrar las nuevas velocidades lineales para nuestros objetos.

Ahora podemos calcular también las ecuaciones pero incluyendo ahora las velocidades angulares, necesitamos entonces la ecuación re un punto rotando y trasladándose a la vez.

$$v_2^{AP} = v_2^A + \omega_2^A r_\perp^{AP}$$

Ahora de la misma forma que se resolvió el problema para las velocidades lineales, de la misma forma se resolverá para las angulares.

$$\omega_2^A = \omega_1^A + \frac{r_\perp^{AP} \cdot jn}{I^A}$$

Esta ecuación es el resultado de aplicar el impulso jn a un punto P en el cuerpo A, de tal forma que el ultimo termino convierte de un impulso lineal un impulso angular de la misma forma en que se convertía una fuerza lineal a un momento de torsión, usando e producto punto al punto de aplicación. Como el impulso cambiara el momento angular se divide por el momento de inercia para convertir la ecuación a velocidades angulares.

De la misma forma que sustituimos v^{AB} en las ecuaciones lineales, sustituimos a la velocidad angular. Al terminar la sustitución y la simplificación y al final el despeje del impulso j, obtenemos:

$$j = \frac{-(1+e)v_1^{AB} \cdot n}{n \cdot n\left(\frac{1}{M^A} + \frac{1}{M^B}\right) + \frac{\left(r_\perp^{AP} \cdot n\right)^2}{I^A} + \frac{\left(r_\perp^{BP} \cdot n\right)^2}{I^B}}$$

con esto ya tenemos el calculo del impulso afectado tanto por efectos lineales como angulares, solo agregamos el resultado de esta ecuación a las ecuaciones de la velocidad lineal y al de la velocidad angular y las modificaciones provocadas por la colisión quedaran terminadas.

Concluyendo, necesitamos conocer que objetos que participan en la colisión, los puntos que colisionaron y la normal de la colisión. Todas esta ecuaciones algunas con otras ligeras variaciones, son mas que suficientes para obtener un simulador que nos devuelva resultado no muy lejanos a la realidad.

3.4.5. IMPLEMENTACIÓN

Para realizar los cálculos en nuestro simulador, necesitamos realizar integraciones analíticas, pero estas pueden ser implementadas en un programa de computadora utilizando integradores numéricos.

Las ecuaciones que hemos manejado hasta ahora, son ecuaciones diferenciales de primer y segundo orden, y la mayoría dependen de valores que se reciben como parámetros que provienen talvez de una función previa o de los movimientos que realice el usuario, en ecuaciones diferenciales a este tipo de problemas se les conoce como problemas de valor inicial, ya que nuestras ecuaciones están sujetas a condiciones preescritas, que son las condiciones que se imponen a una función y(x) o a sus derivadas.

Ejemplo de una función de primer orden sujeta a condiciones iniciales:

$$\frac{dy}{dx} = f(x, y) \quad \text{sujeto a} \quad y(xo) = yo$$

3.4.5.1 MÉTODO DE EULER

El método de euler es una de las técnicas más sencillas para aproximar soluciones del problema de valor inicial.

La derivada de una función y(x), evaluada en un punto xo, es la pendiente de la tangente de la grafica de y(x) en este punto. Como el problema de valor inicial establece el valor de la derivad de la solución en (xo, yo), la pendiente de la tangente a la curva de solución en este punto es f(xo, yo), si recorremos una distancia corta por la línea tangente obtendremos una aproximación a un punto cercano de la curva o grafica de la función de solución. Después se repite el proceso en el nuevo punto. Para formalizar este proceso se emplea la linealización, la figura 3.4.5.1.1 ilustra lo que se acaba de mencionar.

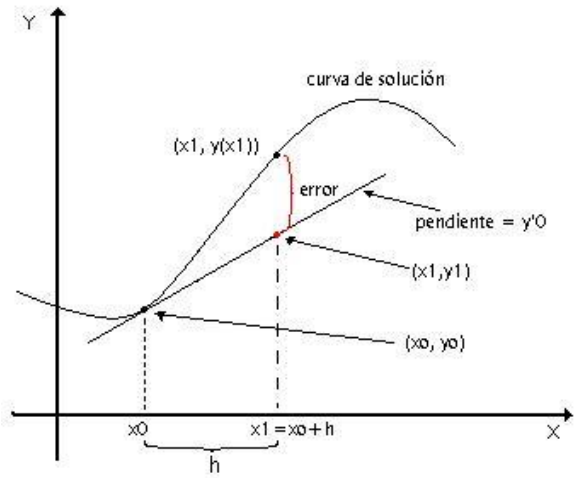


Figura 3.4.5.1.1

Linealización de y(x) en x = x0

$$L(x) = y'(x_0)(x - x_0) + y_0$$

h se define como el incremento positivo en x.

Reemplazando x con $x_1 = x_0 + h$
En la ecuación de linealización, obtenemos:

$$L(x_1) = y'(x_0)(x_0 + h - x_0) + y_0 = y_0 + hy'(x_0) = y_0 + hy_0'$$

$y_0' = y'(x_0) = f(x_0, y_0)$ sustituyendo en la ecuación de $L(x_1)$ tenemos

$$y_1 = y_0 + hf(x_0, y_0) \quad \text{donde } y_1 = L_1(x)$$

El punto (x_1, y_1) sobre la tangente es una aproximación al punto $(x_1, y(x_1))$ en la curva de solución, por tanto.

$L(x_1) \approx y(x_1)$, o $y_1 \approx y(x_1)$ la exactitud depende del tamaño de h.

El método de euler se puede definir entonces como:

$$y_{n+1} = y_n + hf(x_n, y_n) \text{ con } x_n = x_0 + nh$$

por ejemplo queremos calcular la velocidad final del objeto en términos del método de euler:

$y_{n+1} \approx y_n + h\frac{dy_n}{dx}$,
cambiamos las funciones a las de velocidad y tiempo:

$$v_{n+1} \approx v_n + h\frac{dv_n}{dt} \Rightarrow v_{n+1} \approx v_n + h\dot{v}_n \Rightarrow v_{n+1} \approx v_n + h\frac{F_n}{M}$$

$\dot{v}_n$ es la aceleración, si conocemos la fuerza y la Masa, sustituimos estos dos parámetros en la ecuación y ya tendremos una solución.

El simulador implementado en el videojuego, esta dividido en 7 etapas básicas que son controladas por una octava función "Simulador" que realiza los cálculos de los tiempos, los cuales servirán como parámetros de las demás funciones para utilizado el método de euler obtener los resultados deseados al evaluar las ecuaciones. Las 7 etapas del simulador se enumeran a continuación:

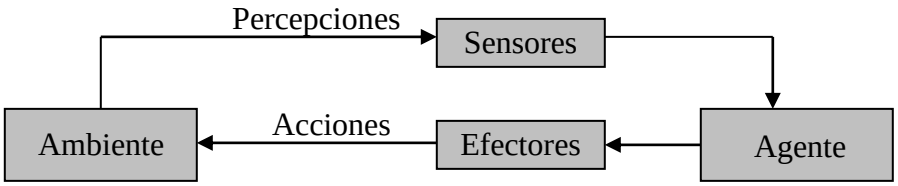
1.- Inicializar Objetos.
2.- Cálculo de dirección.
3.- Cálculo de fuerzas.
4.- Integración.
5.- Cálculo de Vértices.
6.- Calculo de Colisiones.
7.- Resolver Colisiones.

Estas funciones se encuentran en el diagrama 2..7.1 de la sección 2 de análisis.

INTELIGENCIA ARTIFICIAL

AGENTE

Un agente es todo aquello que puede considerarse que percibe su ambiente mediante sensores y que responde o actúa en tal ambiente por medio de efectores.



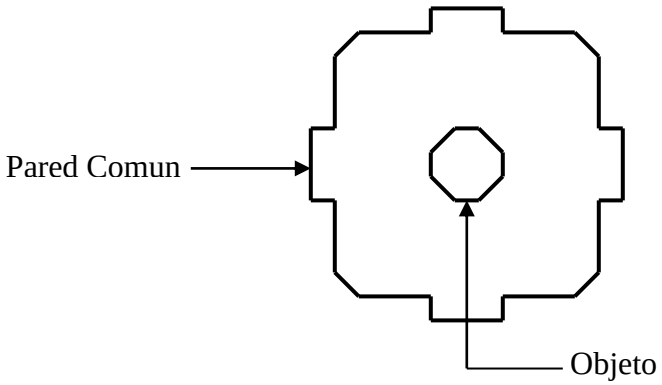
Para el desarrollo del video juego, cada Jugador controlado por la computadora, es considerado como un agente inteligente, ya que actúa de acuerdo a las percepciones de su medio ambiente.

Al inicio de la sesión de juego, el agente inteligente estará vagando en el escenario, y no será sacado de este estado a menos que exista una acción que lo haga cambiar de su estado original a un nuevo estado.

REPRESENTACIÓN DEL MEDIO:

Para que el agente inteligente se desarrolle de forma adecuada en los escenarios del video juego, es necesario una definición exacta del medio que lo rodea, ya que sin esta representación interna, simplemente no simularía la forma de juego de una persona.

Los escenarios del Video Juego están representados con líneas en el Plano Cartesiano, estas líneas son paredes y el conjunto de paredes conforman objetos, como se muestra en la figura.



Cada una de estas líneas esta representada por dos puntos en el plano, y por Geometría Analítica, podemos derivar sus propiedades:

- Pendiente
- Tamaño
- Angulo
- Ecuación

Una vez que se inicia una sesión de juego, el programa obtiene la información del escenario a través de un archivo de texto, en el cual están definidas los 2 puntos de cada línea y estas son guardadas en una clase que contiene toda la información referente al escenario:

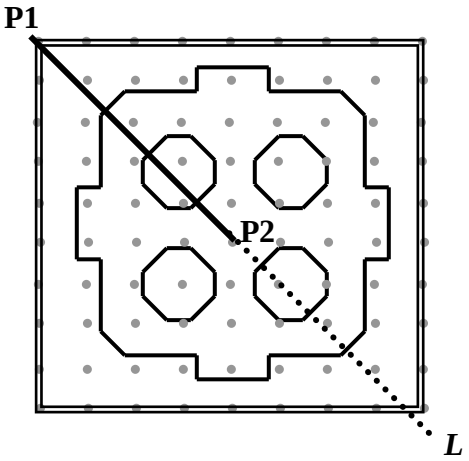
Además de las líneas del Escenario, son definidas 4 líneas mas, que posteriormente servirán como auxiliares en la definición de la estructura de datos. Al final de cada archivo son insertadas estas líneas y son estrictamente indispensables tanto para el desarrollo de la inteligencia artificial, como para el desarrollo del CuadTree para colisiones.

```
Class Recta
{
        .
        .
        float tamaño;
        float ecuación[3];
        float pendiente;
        .
        .
public:
        .                .
        void Inicializar(float x1,float y1,float x2,float y2);
        void Pendiente(void);
        void Ecuación(void);
        .
        .
}Escenario[TOTAL_RECTAS];
```

Después de obtener las ecuaciones de las rectas del escenario, es definido un Grid, el cual servirá de auxiliar para la construcción de la estructura de datos (malla) con la siguiente información en cada nodo:

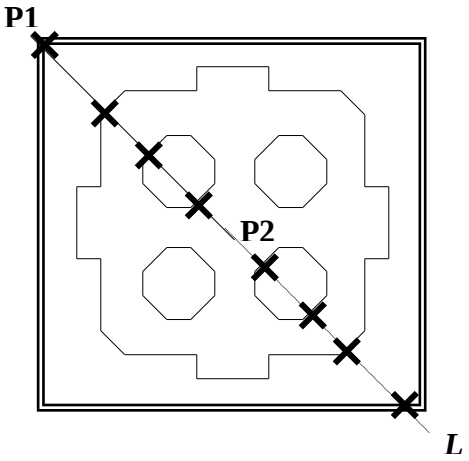
```
struct malla
{
        .
        .
        malla * apuntadores[4];
        float posición_x,posición_y;
        bool valido;
        .
        .
}*Malla;
```

Este Grid traza una línea de cada uno de sus puntos al origen del escenario, como se muestra en la figura para el primer elemento del Grid:



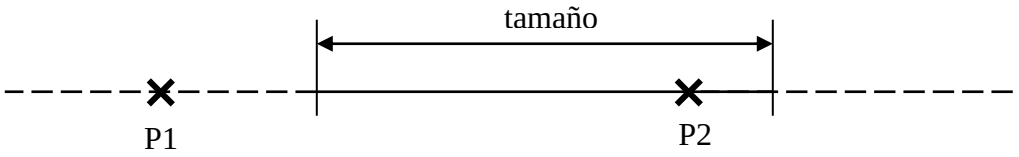
Al obtener la línea L, se pueden obtener sus propiedades simplemente recurriendo a las funciones miembro de su clase, por lo que la ecuación de la recta ya es conocida.

Posteriormente se hace una comparación de esta línea con todas las rectas del Escenario, para poder determinar con que pared del escenario existe intersección, esto se hace igualando ambas ecuaciones y obteniendo el punto de intersección (x, y). La siguiente figura muestra como se hace este proceso para el primer punto del Grid:



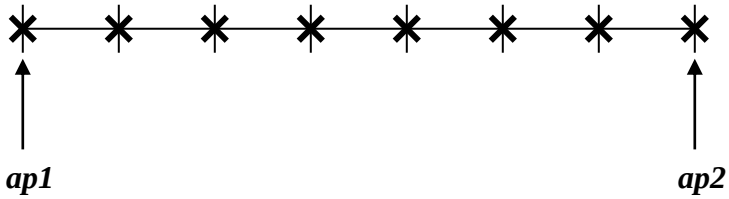
Estas intersecciones son guardadas en una lista doblemente ligada ligada, para poder determinar si el punto del Grid que se esta analizando esta fuera o dentro de un objeto. La elección de una lista doblemente ligada fue por que las operaciones de obtener el elemento anterior son necesarias.

Es necesario hacer notar que las paredes son estrictamente segmentos de línea, y a la hora de buscar las intersecciones por medio de sus respectivas ecuaciones, estas son tomadas como líneas infinitas, por lo que es necesaria la determinación de si un punto pertenece, o no a un segmento dado. La siguiente figura muestra el proceso para determinar si un punto pertenece a un segmento.

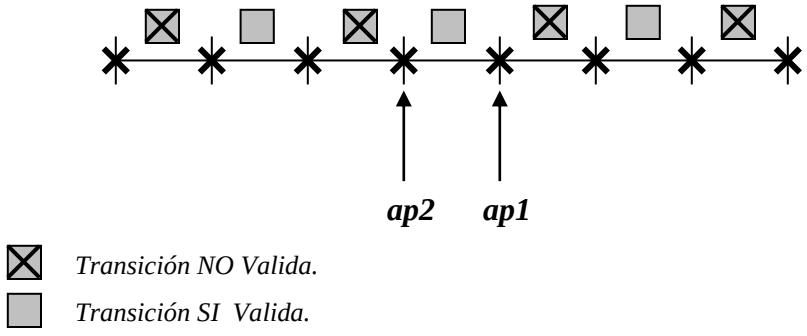


Para determinar si el punto pertenece a la recta, simplemente se comparan la distancias del punto al centro de la recta con la distancia del centro de la recta a cualquiera de sus extremos. Lo anterior se puede obtener fácilmente recurriendo a la variable miembro "tamaño" de la clase recta calculado previamente al inicializar la sesión de juego.

El siguiente paso es el análisis de nuestra lista doblemente ligada para determinar si el punto del Grid es un punto valido o no valido. Este análisis consiste en recorrer la lista en ambos sentidos, es decir, dos apuntadores; uno en el nodo cabecera y el otro en el ultimo nodo de la lista, la siguiente figura muestra como quedaría la lista para el ejemplo anterior.



Los apuntadores se van incrementando y decrementando respectivamente, y el proceso se detiene cuando estos se hayan igualado o sean intercambiados. Al inicio del proceso (ap1 en nodo cabecera y ap2 en el ultimo nodo), estos apuntadores están indicando las intersecciones de la línea trazada por el Grid y las líneas auxiliares, por lo que las transiciones al siguiente nodo respectivamente, serán transiciones no validas, es decir que cualquier punto que se encuentre entre estas intersecciones esta fuera del escenario o bien, esta adentro de un objeto. Continuando con el ejemplo anterior, la siguiente figura muestra como quedan las transiciones para el escenario.



Si se compara este ultimo resultado con la figura del escenario al inicio del ejemplo, se puede observar que los espacios ya están definidos con transiciones validas o invalidas, por lo que un vehículo controlado por la maquina solamente podrá acceder a las transiciones validas.

Ahora el siguiente paso consiste en ir creando los nodos de la malla para el análisis de cada punto que conforma el Grid. La siguiente Clase muestra las principales funciones y datos miembro de la malla.

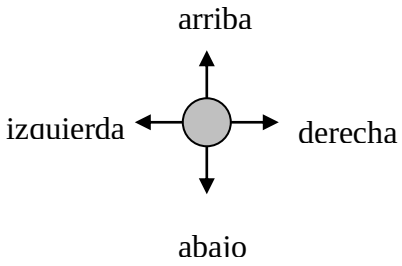
```
class malla
{

   malla * apuntadores[4];
   float x,y;
   bool valido;
   int colu,reng;
   int tot_col,tot_reng;

}*M;
```

Esta clase requiere de mayor análisis, ya que en ella se desarrollara el movimiento del vehículo en el escenario, por lo que a continuación se explicaran brevemente cada uno de sus datos miembro.

- malla * apuntadores[4]: Un arreglo de cuatro apuntadores de tipo malla que servirán para tener acceso a los nodos aledaños al nodo en curso.



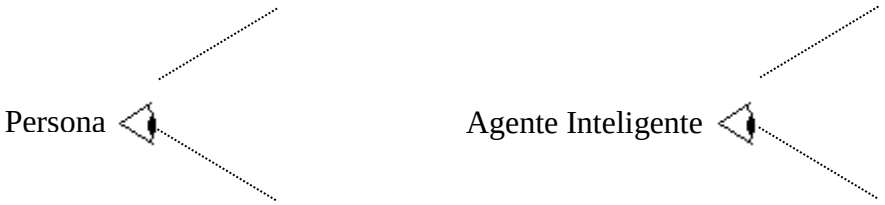
- Dos variables de tipo flotante, que indicarán la posición del nodo en el escenario.
- Una variable booleana que indicara si el nodo es valido o invalido(esto ya se determino previamente en el análisis del Grid).
- Cuatro variables enteras que indicaran en que numero de columna y renglón esta el nodo con respecto a los nodos validos, estas cuatro variables son la esencia de la heurística para la navegación del vehículo en el escenario.

Después de haber considerado otras opciones de representación del escenario, la mejor opción resulto ser la malla. En cuestión del almacenamiento es muy deficiente, pero es estrictamente indispensable que el espacio bidimensional sea discretizado en su totalidad, además una ventaja importante (incluso la que nos llevo a tomar la decisión de escoger este tipo de representación) es que simplifico en su totalidad las operaciones que realizará la computadora en donde esta ejecutándose el juego, por lo que las operaciones heurísticas son muy simples.

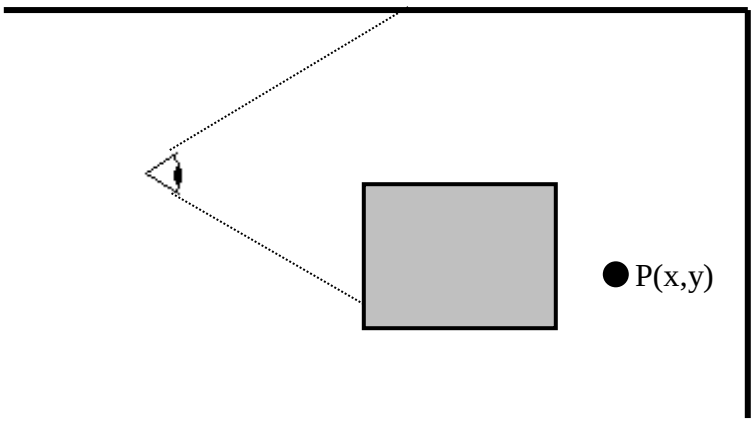
Tal vez (depende del punto de vista) las operaciones realizadas para cada punto del Grid son consideradas como excesivas, pero también hay que hacer notar que estas operaciones son solamente realizadas únicamente al inicio de la sesión de juego, quitando así la carga de operaciones durante la sesión.

PERCEPTORES

Debido a que las percepciones de una persona en una sesión de juego están restringidas únicamente por lo que puede ver, las percepciones del agente inteligente esta igualmente restringidas por un ángulo de visión previamente definido.



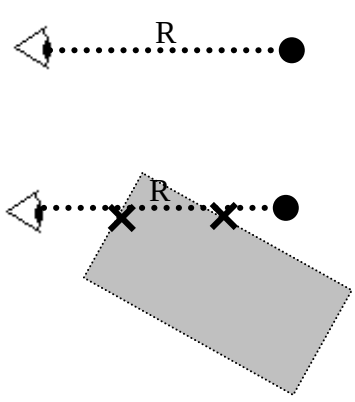
Lo anterior conduce al siguiente problema; si existe un objeto entre el objetivo de visión y el observador



Si únicamente se comparara el ángulo que existe con respecto al eje horizontal, el objeto estaría dentro del ángulo de visión, pero este forma de resolver el problema no contempla el caso en el que existan objetos entre el objetivo y el observador.

Para resolver el problema anterior, se crea una línea imaginaria R desde el objetivo hasta el observador haciendo uso de la clase Recta. Al hacer una instancia de esta clase, se tiene acceso a todas sus funciones miembro (tamaño(),pendiente(),ecuación(),etc.), pero para este caso en particular únicamente se hará uso de la ecuación de la recta.

Al conocer la ecuación de la recta, se hace una comparación de todas las paredes del Escenario con la recta imaginaria R, por lo que se puede conocer si existe una intersección entre los segmentos, si la existe, significa que hay un objeto que se esta interponiendo entre el objetivo y el observador. Si ya se recorrieron todas las paredes del escenario y con ninguna existió una intersección, quiere decir que no hay objetos entre el escenario y el observador, lo anterior se ilustra en la siguiente figura.



No hay ninguna intersección con las paredes del Escenario

Existen dos intersecciones, por lo que el objeto no es visible

Si el resultado fue exitoso (el objetivo es visto), entonces se procede a realizar un par de comparaciones, las cuales determinarán si el objetivo esta dentro del rango de visión.

OPERADORES

Un aspecto importante para el desarrollo del agente inteligente en el escenario virtual es una precisa definición de operadores, estos operadores le permitirán al vehículo controlado por la computadora poder:

- Avanzar
- Rotar
- Disparar
- Desplazamiento Izquierda
- Desplazamiento Derecha.
-

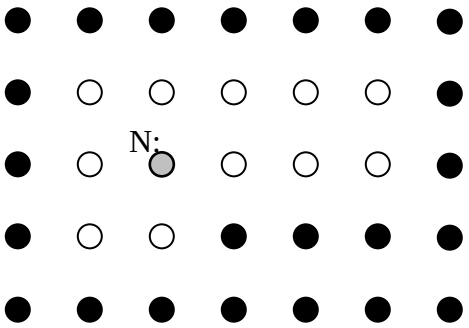
Estos operadores constituyen una sucesión de puntos para que estos sean graficados en la pantalla.

HEURÍSTICA

Navegación:

Recordemos que el objetivo primordial de la Inteligencia Artificial es que la forma de juego de la computadora sea lo mas parecido a la de una persona, por lo que si el vehículo se encuentra chocando sin ningún sentido con las paredes, evidentemente el proceso esta fallando.

Al terminar de definir la representación del escenario (estructura de datos tipo malla), se procede a definir una heurística por cada nodo de la malla. Esta definición está en función del número de columna y número de renglón en el nodo correspondiente donde se encuentra el vehículo.



El nodo N tiene la siguiente información:

Columna: ***2***
Total columnas: ***3***
Renglón: ***2***
Total renglones: ***5***

○ Nodos SI validos
● Nodos NO validos

La enumeración de renglones y columnas se hace de la misma forma, a continuación se explicara la enumeración únicamente para renglones:

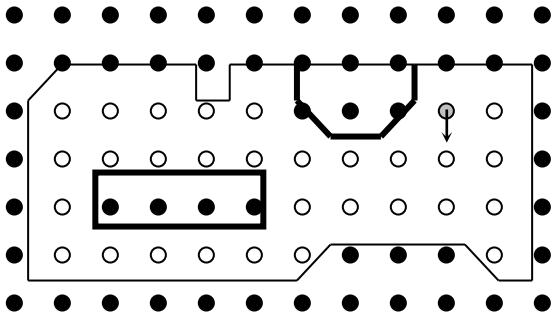
- En cada renglón se cuentan los nodos validos que existen entre dos nodos no validos, para el ejemplo es cinco.
- Se asigna en cada nodo el numero que le corresponde, para el ejemplo, es de dos

La enumeración de renglones y columnas están definidas en dos funciones globales que se ejecutan únicamente al inicio de la sesión de juego, por lo que cada vez que el agente inteligente visite un nodo durante el juego, conocerá previamente en que numero de nodo se encuentra.

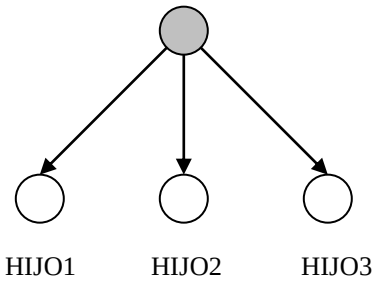
Durante la sesión del juego, el agente inteligente se encontrara en un proceso continúo de la búsqueda de su oponente, por lo que cada vez que se disponga a navegar, accederá a los nodos vecinos al que se encuentra y decidirá cual será la mejor opción para

aplicar el operador correspondiente. Esta función esta implementada con un árbol de decisión ternario. A continuación se muestra un ejemplo de cómo se desarrollaría este tipo de árbol de decisión:

Dado el siguiente escenario, la malla es definida como:



Si suponemos que el punto inicial del análisis es el nodo en color gris con dirección hacia abajo, los elementos que entran para la creación del árbol de navegación son los nodos que se encuentran frente al nodo en curso. La siguiente figura separa estos nodos de los restantes del ejemplo anterior:



Y la decisión es tomada de acuerdo al numero de renglón o columna según, sea el caso para poder así moverse al nodo elegido. La siguiente tabla muestra la heurística para cada una de las tres posibilidades:

	Renglón	Columna	Total renglones	Total columnas
HIJO 1	8	1	10	2
HIJO 2	9	2	10	3
HIJO 3	1	2	10	4

Es claro que la ruta por donde existen mas nodos antes de que se tope con una pared es la de HIJO1, por lo que el agente inteligente tomara la decisión de aplicar el operador de rotación hacia la derecha con respecto a el.

CONCLUSIONES

En ocasiones nos hemos encontrado personas que piensan que realizar un videojuego no tiene mucho chiste, tal vez piensan que durante el desarrollo de este solo vamos a estar jugando y no pondremos en practica lo que se ha aprendido en la escuela. Pues bueno, para nuestro punto de vista, un videojuego es uno de los proyectos más completos que se pueden desarrollar en una escuela cuyo objetivo es entrenar gente especializada en el desarrollo de sistemas ya que para desarrollarlos se requieren conocimientos avanzados de programación, aplicamos directamente lo aprendido en materias como Física, Ecuaciones Diferenciales, Álgebra Lineal, Calculo, Análisis Vectorial, todas las de Programación, redes de computadoras, inteligencia artificial, graficación, ingeniería de software y técnicas de análisis y diseño de sistemas. Talvez solo materias orientadas a Bases de Datos y Electrónica no fueron aplicadas por que quedan fuera del alcance de lo que se necesita para un videojuego. Aunque el detector de colisiones y el agente inteligente implementado en el videojuego tienen la capacidad de recibir cualquier tipo de escenario, con lo que tal vez una extensión de ellas podría permitir a un robot navegar en un escenario real, basándose en los sensores instalados en el robot y con previo conocimiento de la forma del escenario en el que se encuentra.

El desarrollo de este videojuego fue gratificante ya que hicimos algo que nos gusta y además nos permitió reafirmar nuestros conocimientos, y comprender muchas de las cosas que alguna vez aprendimos y que no habíamos entendido bien.

REFERENCIAS

Stuart Russell, Peter Norvig.
Inteligencia artificial avanzada, un enfoque moderno. 1995.
Prentice Hall.

Elaine Rich, Kevin Knight.
Inteligencia Artificial. 1994. Segunda Edición.
Mc. Graw Hill.

Alan Watt.
3D Computer Graphics. 1990.
Addison Wesley.

Hearn & Baked.
Gráficas por computadora. 1995. Segunda Edición.
Prentice Hall.

Ceballos Francisco Javier.
Visual C++ 6. Aplicaciones avanzadas para win32. 1998.
Alfa Omega.

Ferdinand P. Beer & E. Russell Johnston, jr.
Mecánica Vectorial para Ingenieros, Dinámica, quinta edición
Mc Graw Hill

R.C.Hibbler
Mecánica para Ingenieros, Estática, 2da edición
CECSA

Earl W. Swokowsky
Calculo con geometría analítica, 2da edición
Ibero América

Dennis G. Zill
Ecuaciones Diferenciales, 6ª edición
Thomson

Halliday & Resnick
Física parte 1, 1ra edición
CECSA

Tippens
Física conceptos y aplicaciones, tercera edición
Mc Graw Hill

REFERENCIAS

Harvey Gerber
Álgebra Lineal, primera edición
IberoAmérica

Lehmann
Geometría Analítica
Limusa

Shoichiro Nakamura
Métodos numéricos aplicados con software, primera edición
Prentice Hall

Aarón M. Tenenbaum, Langsam y Augenstein
Estructuras de datos con c y c++, segunda edición
Prentice Hall

Documentación del SDK de DirectX 8 y 8.1 para Visual C++.

GameDev.Net – Articles and Resources.
www.gamedev.net/reference

Visual C++ World – Directx Tutorials.
www.vcworld.f2s.com/tutorials/directx.php

Drunken Hyena Direct3D Tutorials.
www.drunkenhyena.com/docs/d3d_tutorial.phtml

NeXe, NeHe DirectX style.
nexe.gamedev.net/tutorials/list.asp