DESARROLLO DE UN MOTOR DE EVENTOS PARA VIDEOJUEGOS

Héctor Agustín Lana

Miguel Garde Vallés

Contenido

1.-Introduccion 3
2.-Estructura de un motor de Eventos 3
3.-Implementación de un motor de Eventos para ordenador 7
3.1 Núcleo 7
3.2 Entrada 8
3.3 Eventos 9
3.4 Estados 10
3.5 Otros Subsistemas 11
4.-Desarrollo de una demo para ordenador utilizando el motor de Eventos y Ogre3D 13
4.1 Definiendo los requisitos de la demo 13
4.2 Actores, Vistas y Listeners 14
4.3 Estados 16
4.4 Eventos 17
5.-Inteligencia Artificial 22
5.1.-Introduccion 22
5.2.-Estructura del motor de Inteligencia Artificial 23
5.3.-Interaccion del motor de IA con el resto del Juego 24
5.4.-Percepcion 26
5.5.-Toma de decisión 27
5.6.-Movimiento (Steering Behaviors) 30

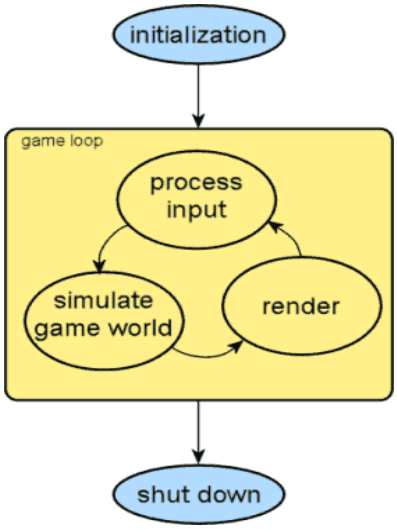
1.-Introduccion

Con este proyecto se pretende realizar un motor de eventos para videojuegos de ordenador, completamente independiente de la parte física, lógica y gráfica, que permita la creación y escalabilidad de cualquier tipo de videojuego, desde un Space Invaders hasta un Grand Theft Auto. Así mismo, se desarrollará una demo para demostrar la potencia de un sistema de eventos.

El motor de eventos está dividido en cuatro secciones también independientes entre sí: el núcleo, la parte encargada de preparar el sistema; la entrada, sección pendiente de los dispositivos Input (ratón, teclado, joysticks, etc.); los estados, controladores del flujo de las secciones del juego; y los eventos, interfaces preparadas para crear mensajes entre los diferentes objetos que darán vida al juego.

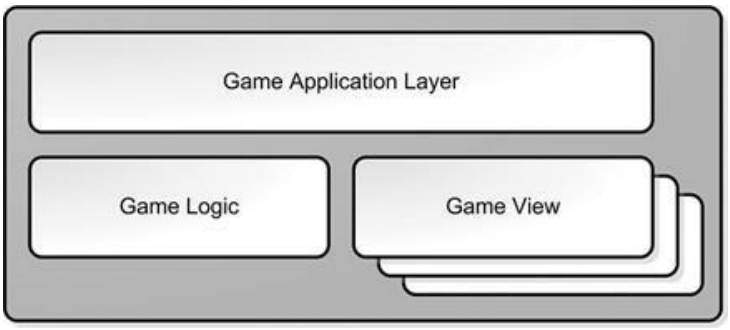
2.-Estructura de un motor de Eventos

Un motor de eventos no es muy diferente de un motor de videojuegos convencional, mantiene su misma estructura, aunque con otra y filosofía que permite maximizar la coherencia y minimizar la cohesión; algo muy valorado en la ingeniería del software.



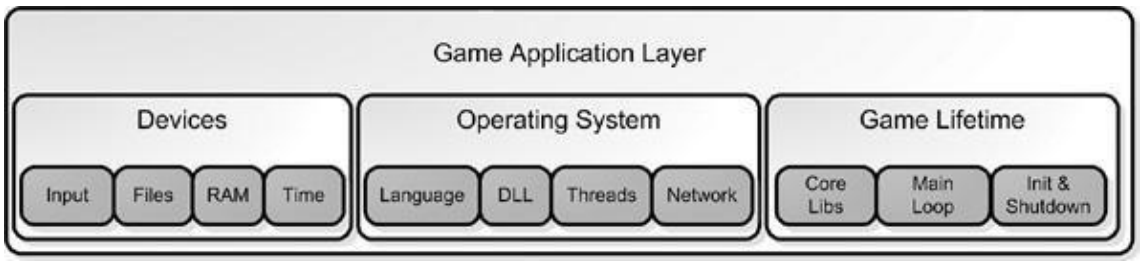
Estructura de un videojuego convencional: tras la inicialización del juego hay un bucle que se repite hasta que el jugador quiere. Finalmente, al acabar el juego se limpia y cierra la aplicación.

El motor de eventos incorpora dentro del bucle una cuarta actividad, que es el procesamiento de los mensajes (eventos) que se transmiten los objetos del juego. Sin embargo, la principal diferencia de este sistema no radica ahí, sino que lo hace en separar el juego de la siguiente forma:



La capa de aplicación (Game Application layer) se encarga de todo lo relacionado con la máquina en la que se está ejecutando el juego: forma de renderizar, tipos de controles de juego, etc. Esta parte es específica para la plataforma y debe rehacerse en cada ocasión. Ejemplo: Si se tiene un videojuego para ordenador que se controla con el ratón y se quiere portar a móvil, en principio bastaría con cambiar esta parte de la aplicación. Se sustituiría el sistema para controlar el ratón por uno para controlar la pantalla táctil del móvil, se cambiarían los sistemas de resolución del juego para adaptarlos a un dispositivo más pequeño, etc.

La capa de aplicación incluiría los procesos de inicialización, entrada, renderizado y cierre de la estructura de videojuegos convencional.

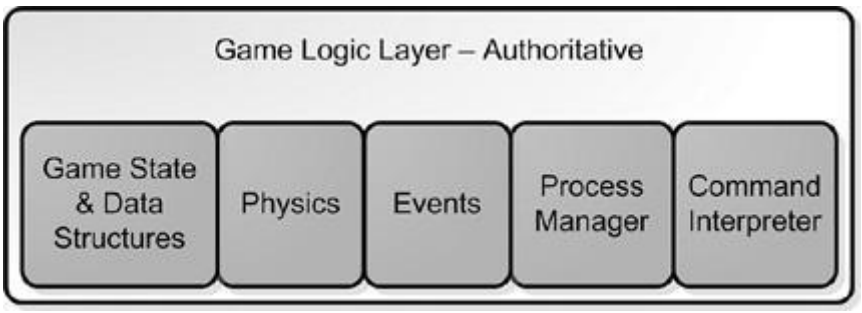


Partes de las que se encarga la capa de aplicación: dispositivos de entrada, de ficheros, de memoria y de tiempo; temas del sistema operativo como hilos y red; y los procesos del propio juego, como ejecutar el bucle principal y la inicialización y cierre.

La capa de la lógica del juego (Game Logic Layer) se corresponde a lo que es el juego en sí. Controla si el jugador está en el menú principal o en plena acción, simula las físicas del

juego, los sonidos, el comportamiento de los personajes no jugadores, y un largo etcétera. Se relaciona con la capa de aplicación recibiendo de ésta las teclas pulsadas por el jugador y diciéndole qué mostrar en la pantalla, qué no y qué sonidos reproducir por los altavoces. Esta parte es independiente de la máquina sobre la que se ejecuta, y en principio bastaría con recompilarla para que funcionase en cualquier plataforma o sistema operativo.

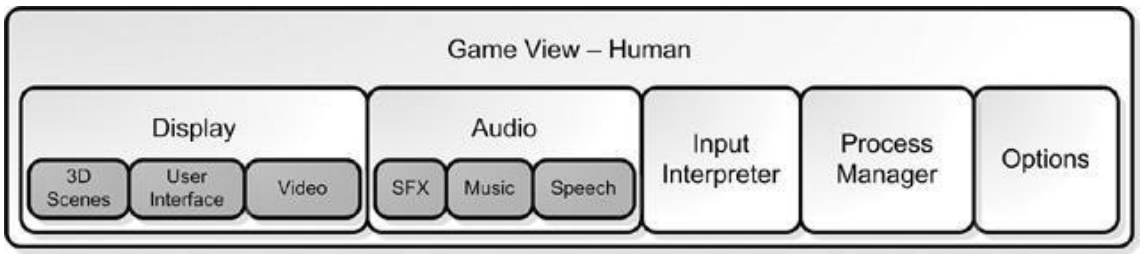
La lógica del juego del motor de eventos es exactamente igual que el proceso de simulación de mundo en un motor convencional.



Partes de las que se encarga la lógica del juego: estructuras de datos y estados de juego, física, eventos, controlar los procesos e interpretar los comandos de las vistas de juego.

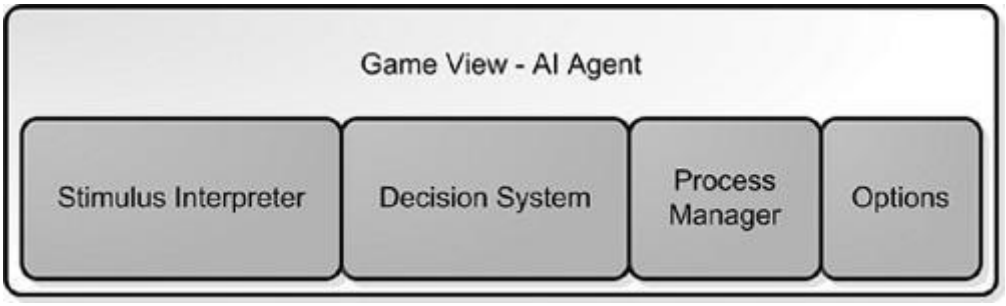
Las vistas del juego (Game Views) son uno de los aspectos diferenciadores de este tipo de motor. El cometido de estos subsistemas (que se pueden crear tantos como se quiera o se requiera) es representar los elementos dentro del juego, interpretar el mundo que les rodea, enviar comandos a la lógica del juego y enviar y recibir mensajes (eventos) a y de otras vistas. La ventaja de las vistas del juego es que pueden tener diferentes implementaciones; unas pueden procesar la entrada del teclado y enviar comandos según se ha pulsado o no determinada tecla, mientras que otras pueden procesar complejos algoritmos de inteligencia artificial utilizando información sobre el entorno y tomar una decisión, con la que deciden qué comando enviar a la lógica del juego. De este modo se consiguen abstraer todos los jugadores junto con los agentes inteligentes. Por esta razón se pueden diferenciar varios tipos de vistas de juego:

Vista de Juego - Humano



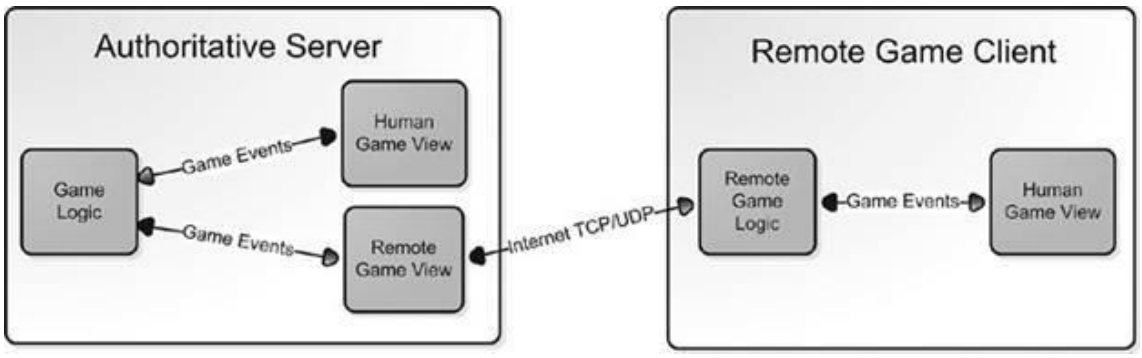
Representa a un jugador humano. El mundo es interpretado a través de la pantalla, con gráficos, y de los altavoces, con sonidos. El jugador toma todas las decisiones y la vista se encarga de interpretar la entrada (teclado, ratón, etc.) como acciones del juego (saltar, avanzar…).

Vista de Juego – Agente inteligente (Inteligencia Artificial)



Representa a un jugador no humano, es decir, controlado por el ordenador mediante inteligencia artificial. El mundo es interpretado a partir de un número limitado de estímulos que llegan desde la lógica del juego. La vista utiliza esta información para ejecutar diferentes algoritmos de decisión y, dependiendo de los resultados del algoritmo, se envían a la lógica del juego diferentes acciones (saltar, avanzar…).

Vista de Juego – Remota



Representa a un jugador humano, pero su finalidad es enviar y recibir los mensajes del usuario a través de una conexión.

3.-Implementación de un motor de Eventos para ordenador

Como ya se ha comentado anteriormente, la finalidad de este proyecto es realizar un motor de eventos para videojuegos de ordenador y una demo. El primer paso es implementar la capa de aplicación, que como también se ha dicho ya, es específica para cada plataforma. En este caso se ha decidido aprovechar las características y funcionalidades que ofrece el motor gráfico Ogre3D, tanto a nivel de estructuras de datos como a nivel de renderizado, dadas las facilidades que aporta.

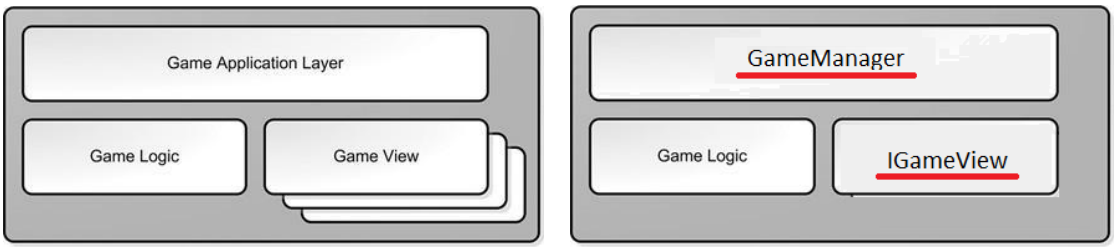
El proceso de implementación del motor se ha dividido en cuatro apartados: Núcleo, Entrada, Eventos y Estados. Los dos primeros se corresponden a la capa de aplicación, mientras que los otros dos entrarían más en la capa de lógica del juego; pero dada las generalidades que tienen estos últimos, cualquier lógica debe implementar el sistema de una forma similar. Es por ello que se ha considerado incluirlos dentro del motor.

3.1 Núcleo

El núcleo estará dividido en dos partes, GameManager y IGameView. La primera parte es una interfaz cuya finalidad es generalizar y aunar todas las vistas de un videojuego. La segunda parte del Núcleo es implementable en una única clase y se encargará, ayudada por Ogre3D, de inicializar la aplicación, crear una ventana de juego, ejecutar el bucle principal actualizando el estado actual en el que se encuentre el juego, renderizar y cerrar el sistema.

Las dependencias disminuirán de tal manera que el programa principal quedará muy simplificado:

```
int main( int argc, char **argv ) {

        // creamos el GameManager

        GameManager   *gameManager  =  new GameManager ();

        //creamos un vector con los estados que tendrá el juego

        std::vector<GameState*> estados;

        estados.push_back(new EstadoJugando());

        estados.push_back(new EstadoMenu());

        estados.push_back(new EstadoCreditos());

        //comenzamos el juego diciendo cuál es el estado inicial

        gameManager->startGame(estados," EstadoMenu");

}
```



La capa de aplicación se encapsula (en parte) dentro del GameManager y todas las vistas se aúnan en IGameView.

3.2 Entrada

La entrada se encarga de todo lo relacionado con los dispositivos que tiene el usuario para dar órdenes al ordenador, en este caso teclado y ratón. Se implementan dos clases diferentes; la primera, llamada GameAction, representa cualquier acción que pueda transcurrir dentro del juego, ya sea saltar, disparar, salir del juego, etc. y se asocia a una o varias entradas (teclas o botones del ratón). La segunda clase se llama InputManager, y es específica para la plataforma, ya que se encarga precisamente de leer los dispositivos de entrada y hacer de puente entre ellos y las GameAction.

Ejemplo:

```
//comienzo de la aplicación, creamos los objetos y asociamos la acción con la tecla espacio
InputManager*   inputManager =new InputManager();
GameAction*  accionSaltar = new  GameAction(GameAction::DETECT_INITAL_PRESS_ONLY);
//la librería OIS de C++ contiene identificadores para todas las teclas del teclado
inputManager->mapToKey(accionSaltar,OIS::KC_SPACE);
    …
//bucle principal
inputManager->update();//lee las teclas pulsadas y avisa a las GameAction que corresponda
…
If (accionSaltar->isPressed()) {
    //código para que el personaje salte
}
```

3.3 Eventos

Con lo que se ha visto hasta ahora, el motor que se podría implementar prácticamente no diferiría de un motor de juegos convencional. Sin embargo, este apartado marca de manera obvia la diferencia caracterizadora de un motor de eventos.

Los eventos son el mecanismo a través del cual los subsistemas del juego se comunican, perciben los cambios en el universo para luego reaccionar a ellos. Por ejemplo, si se tiene en un escenario una radio encendida, el sistema gráfico necesitará crear polígonos y texturas para que se pueda ver, el sistema de audio creará efectos de sonido para que se pueda oír, y posiblemente la inteligencia artificial responderá a la presencia del objeto. Todos estos sistemas deben saber que la radio existe, y qué está haciendo; esta información se le notifica a los sistemas mediante eventos. Igual que una aplicación de Windows escucha eventos de ratón (OnMouseMove, OnMouseEnter, etc.) los sistemas de un juego pueden tener listeners que escuchen eventos del propio juego (entradas de teclado, cargado y guardado de niveles, etc.).

Para llevar a cabo este modelo o sistema de eventos se deben definir cada uno de esos eventos y las estructuras de datos que contendrán. Cada subsistema se registra, a través de un controlador de eventos, a escuchar ciertos eventos, y cuando un evento es disparado el controlador se encarga de distribuirlo a los subsistemas correspondientes. Un buen ejemplo podría ser el subsistema de sonido: se puede registrar a eventos de colisión, de modo que cuando dos objetos choquen y el sistema de física dispare el evento correspondiente, el sistema de sonido lo recibirá y podrá reproducir el efecto de sonoro adecuado. De este modo se consigue que el sistema de física no se tenga que encargar de avisar directamente a todos y cada uno de los sistemas que reaccionarán a una colisión, y a la vez se consigue que el sistema de sonido no tenga que comprobar constantemente si dos objetos han chocado.

Es importante destacar que la definición de los eventos y los comportamientos de los subsistemas varían en cada juego, por lo que entrarían dentro de la capa de lógica del juego. Sin embargo, al igual que con las vistas (GameView) anteriormente mencionadas, hay varios aspectos que se pueden generalizar; por tanto, es posible crear un sistema de eventos abstracto reutilizable para todo tipo de juegos. Esto será o que se implementará.

El apartado de Eventos que se ha implementado incluye una clase abstracta Event, que será la que se utilice para transmitir los mensajes; una interfaz llamada IEventListener, que podrá escuchar eventos; y un controlador, EventManager, encargado de registrar a los listeners y notificarles cuando se dispare un evento.

Ejemplo:

```
//inicio del juego

EventManager *evtMgr = new EventManager();

ClassA* classA = new ClassA(); //clase que implementa la intefaz IEventListener

evtMgr->addListener(classA,Evento_Colision);
```

```
…

//en algún punto del juego

evtMgr->queueEvent(new Evento_Colision(objeto1, objeto2));

…

//en la clase ClassA, donde se ha implementado el método heredado 'HandleEvent'

bool HandleEvent(Event & evento ) {

        //código para tartar los eventos

}
```

3.4 Estados

En este punto ya se podría parar, pues una vez implementado el sistema de eventos el motor ya estaría completo. Sin embargo, se ha considerado ventajoso ir un paso más allá y hacer, al igual que con el sistema de eventos, una aproximación abstracta a los diferentes estados de juego. Aunque varíen de aplicación a aplicación, siempre habrá al menos un estado de juego; y siempre se necesitarás realizar acciones básicas como la carga y descarga de recursos, la actualización continua del universo del juego iteración tras iteración, el renderizado de una escena, etcétera.

El sistema de estados estará compuesto por dos elementos; uno ya ha salido anteriormente hablando del Núcleo del juego, es el GameState. Esta clase abstracta sirve de base y proporciona un interfaz para los diferentes estados que habrá en cualquier juego. Por otro lado está el controlador o gestor de estados, GameStateManager, que se encarga de controlar cuál de todos los estados debe ejecutarse así como de cambiar de un estado a otro. Así mismo, este gestor será el que detecte cuándo el usuario quiere dejar de jugar y detener el bucle principal del juego.

3.5 Otros Subsistemas

Aunque se han mostrado estos cuatro pilares base de un motor de eventos, existen otros subsistemas adicionales sobre los que debe sostenerse cualquier videojuego. Sin embargo, son tan complejos y existen ya tantos que no merecía la pena meterse con ellos, al menos sin un equipo de desarrollo más numeroso y con más tiempo. Estos subsistemas son:

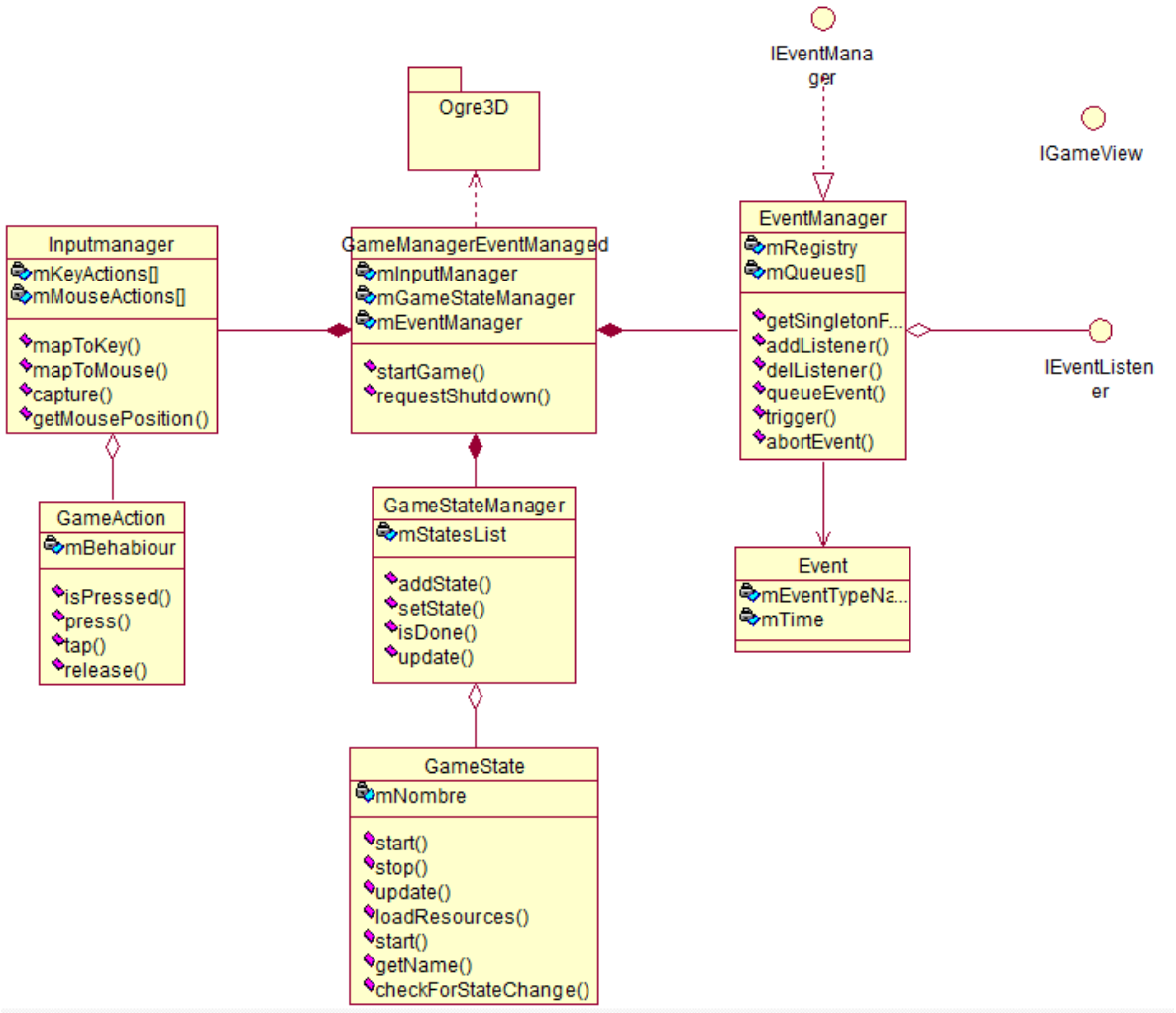
· Gestor gráfico: encargado de renderizar el universo, disponer las resoluciones de pantalla, los frames por segundo (fps) el modo ventana o pantalla completa, etc. Para cubrir la falta de este gestor se ha utilizado Ogre3D, que proporciona todas estas utilidades.

· Gestor de sonido: tiene como cometido reproducir los efectos sonoros, controlar el volumen, los canales de audio, el control del estéreo, etc. La falta de un gestor de sonido se ha suplido con la utilización y personalización de una librería de audio de software libre creada por David Saltarés, ganador del V Concurso Universitario de Software Libre (CUSL) y apoyada con la librería de audio STL.

· Gestor de recursos: se encarga de pasar del disco a memoria todo tipo de datos, desde texturas y sonidos hasta escenarios y esqueletos de animaciones; así como de su posterior eliminación. Ogre3D proporciona todo lo necesario para que no haya que preocuparse de nada.

· Gestor de físicas: su papel es simular el comportamiento físico de los objetos, desde aceleraciones gravitatorias o de impactos hasta rebotes, giros y frenadas, pasando por supuesto por la detección y el reporte de colisiones. Éste último apartado es el que más nos interesa, y casualmente David Saltarés, mencionado hace dos párrafos, implementó su propio sistema de colisiones que podremos reutilizar.

Una vez vistas las cuatro partes del motor, se puede dibujar un diagrama de clases completo en el que se comprenderá de manera visual la relación entre los mismos:


IEventMana
ger
Ogre3D
IGameView
Inputmanager
mKeyActions[]
mMouseActions[]
mapToKey()
mapToMouse()
capture()
getMousePosition()
GameManagerEventManaged
mInputManager
mGameStateManager
mEventManager
startGame()
requestShutdown()
EventManager
mRegistry
mQueues[]
getSingletonF...
addListener()
delListener()
queueEvent()
trigger()
abortEvent()
IEventListen
er
GameAction
mBehabiour
isPressed()
press()
tap()
release()
GameStateManager
mStatesList
addState()
setState()
isDone()
update()
Event
mEventTypeNa...
mTime
GameState
mNombre
start()
stop()
update()
loadResources()
start()
getName()
checkForStateChange()

4.-Desarrollo de una demo para ordenador utilizando el motor de Eventos y Ogre3D

Con el motor ya implementado se desarrollará un pequeño juego, una demo, que utilizará diferentes funcionalidades de forma sencilla y poco acoplada. Mediante el prototipo se demostrará la gran potencia que acompaña a este tipo de motores, permitiendo una escalabilidad muy elevada y evitando interrelacionar cada sistema.

4.1 Definiendo los requisitos de la demo

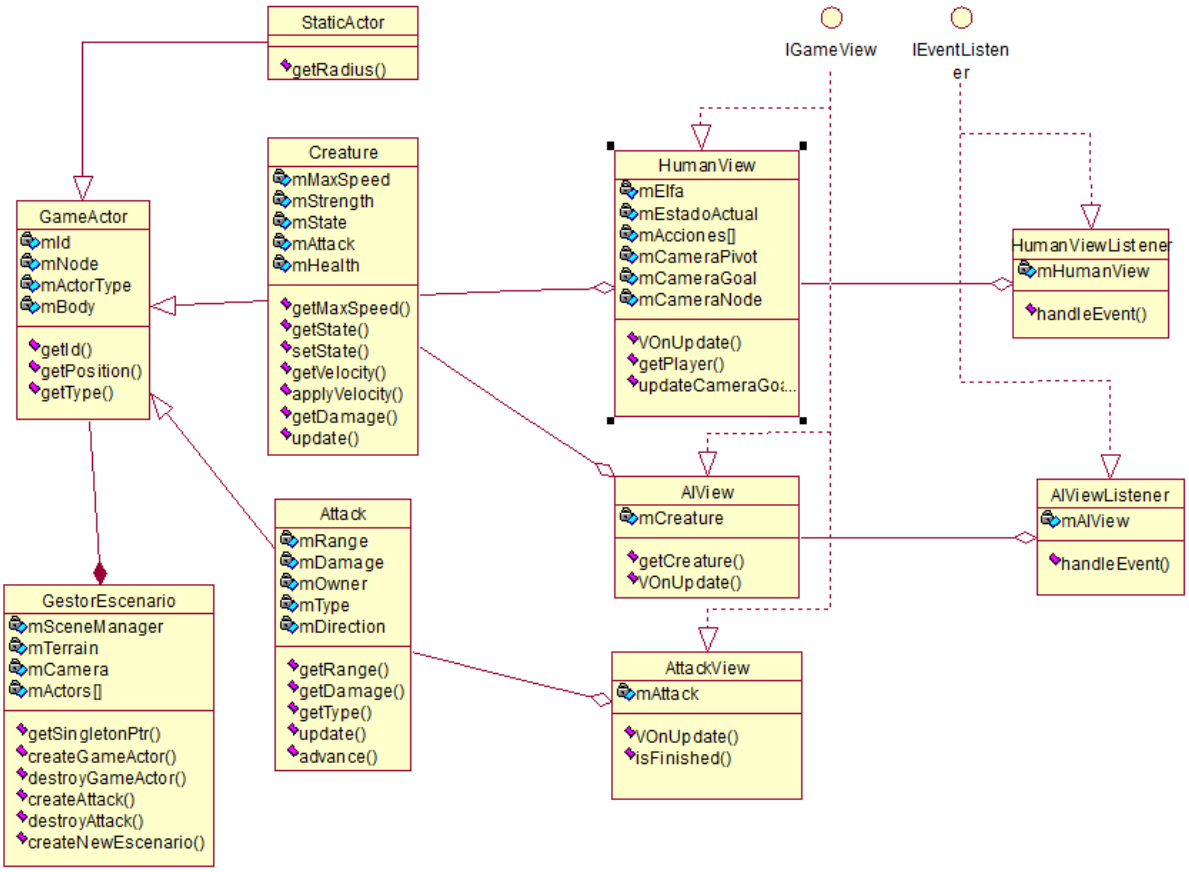
Parte de este proyecto consiste en la realización de un motor de Inteligencia Artificial, de modo que la demo debe adecuarse para mostrar también las funcionalidades del mismo. Se ha considerado que para que la demo sea simple pero completa, el juego consistirá en lo siguiente:

El escenario será un campo abierto, con pendientes, donde estarán situados todos los elementos del juego (distintos tipos de criaturas, árboles, etc.) El personaje principal será una elfa que podrá deambular por el mapa matando animales tanto a distancia como cuerpo a cuerpo, evitando a su vez que los animales salvajes la maten a ella. La cámara se situará en tercera persona y los controles serán mediante combinación de teclado y ratón.

Por otro lado estarán personajes no jugables o PNJs. Habrá ciervos, lobos y dragones (por razones de tiempo no se ha podido poner soldados, que era la idea inicial; por eso se han sustituido por dragones). Cada tipo de criatura se comportará de una forma diferente, los ciervos huyen de cualquier criatura que ven, los lobos persiguen a los ciervos y a la elfa y huyen de los dragones, y los dragones persiguen a todas las criaturas que ven. Los lobos, además, trabajarán en equipo, de modo que cuando uno vea a una criatura aullará para que vengan otros a ayudarle. Una criatura que persiga a otra la atacará hasta matarla.

Todos estos estímulos exteriores que hacen comportarse a las criaturas de una forma u otra se tratarán con eventos, pero eso se explicará más adelante.

4.2 Actores, Vistas y Listeners



Actores

Representan todo tipo de elementos que puedan existir de forma temporal en la partida (el protagonista, los dragones, las bolas de fuego…).

- La clase GameActor, de la que heredan todas las demás, contiene la información necesaria para representar al actor en el universo del juego: tiene un identificador, único por actor, un nodo de la librería Ogre3D con información sobre la posición, orientación, etc., un identificador del tipo de actor que es, y un cuerpo de la librería de físicas que define su tamaño de colisión (BoundingBox).

- La clase StaticActor representa objetos inertes, como árboles, casas o piedras. Añade una funcionalidad más a la clase GameActor, que es el método getRadius(), lo que permitirá a los agentes inteligentes rodear el objeto.

- La clase Creature engloba elementos del universo de juego que tienen capacidad pensante, son los lobos, ciervos, dragones y la elfa. Lo único que diferencia a unos de otros, además del modelo, es sus atributos de velocidad, vida máxima, fuerza, etc. Como se puede observar, al no haber una clase para cada animal, se pueden crear muchas criaturas diferentes simplemente cambiando el modelo y los parámetros.

- La clase Attack tiene como cometido representar los ataques que hay en el juego (flechas, bolas de fuego, zarpazos…). Son unos actores (invisibles en el caso de los ataques cuerpo a cuerpo) que se desplazan en línea recta a velocidad constante hasta una distancia determinada o hasta que chocan contra algo.

- La creación y destrucción de todas estas clases con sus correspondientes recursos se gestiona a través del Singleton GestorEscenario. Éste contiene referencias a clases de Ogre3D que se ocupan de cargar modelos, animaciones, sonidos, etc. Es, en definitiva, una factoría de GameActors. Esta clase contiene también todo lo relacionado con el escenario: mapa de alturas del terreno, intensidad y color de la luz y de la niebla, sombras…

Vistas

Si los actores representan físicamente a los elementos que hay en el juego, las vistas son su representación mental, son las clases que definen cómo se van a comportar las criaturas que habitan el universo.

- La clase HumanView gestiona el comportamiento humano, del jugador, utilizando para ello las entradas de teclado y ratón, con las que controlar los movimientos de la elfa y de la cámara.

- La clase AttackView simula la 'inteligencia' de los ataques. En realidad, lo único que hace es controlar el desplazamiento uniforme del que se ha hablado antes.

- La clase AIView representa el comportamiento de los agentes inteligentes (todas las criaturas que no son la elfa), es la ventana al motor de Inteligencia Artificial; ahí se profundizará mucho más en ella.

Listeners

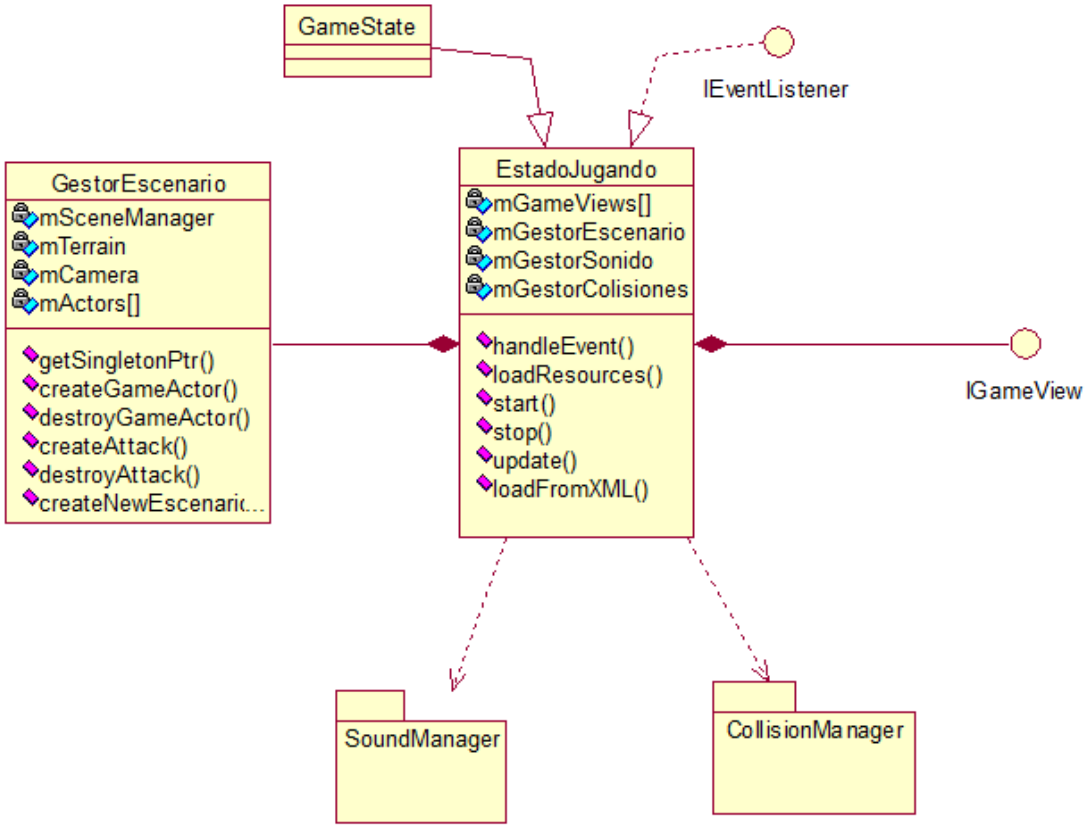
Los listeners son los encargados de captar los eventos que se deseen para luego procesarlos como corresponda. Aunque cualquier subsistema puede tener un listener (audio, física, etc.) los más comunes son los listeners asociados a las vistas, lo que les permite avisarles de lo que ocurre a su alrededor.

- La clase HumanViewListener se encargará de recoger eventos que tengan que ver con la elfa (daño de ataque, etc.) y otros que puedan ser interesantes para el jugador (sonidos, etc.).

- La clase AIViewListener recogerá todos los eventos relacionados con la Inteligencia Artificial (movimiento, visión, etc.). En la parte del motor de IA se explicarán más en detenimiento.

4.3 Estados

Para el desarrollo de esta demo hemos considerado que la creación de diferentes estados no es relevante, así que se ha creado un único estado, llamado EstadoJugando, que representa la partida. En él se crearán el terreno, las criaturas, las vistas, etc. y se gestionarán los eventos relacionados con el escenario.

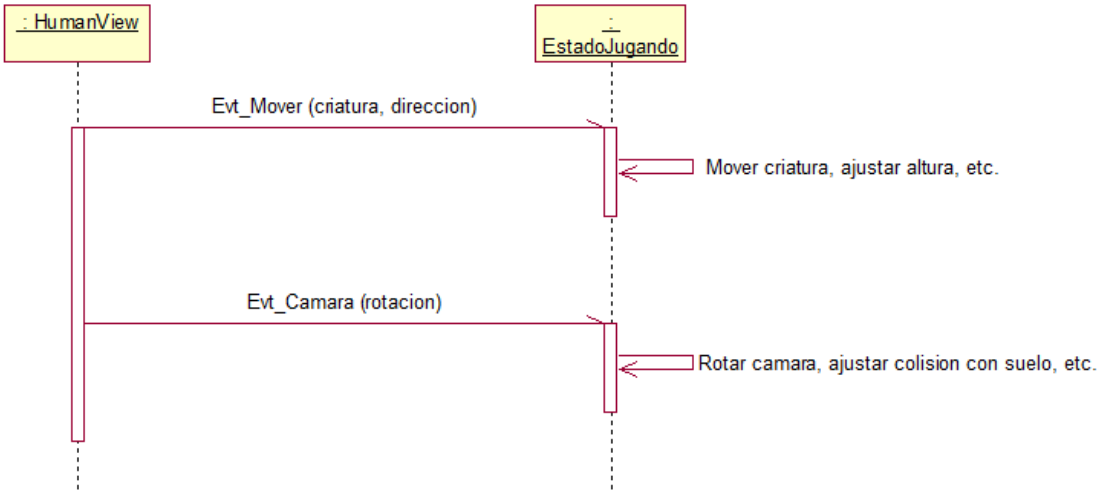


4.4 Eventos

Los eventos, como ya se ha venido diciendo a lo largo de todo este proyecto, son los mensajeros de los acontecimientos del juego, el sistema de comunicación entre todos los subsistemas. Han sido agrupados según sus funcionalidades de la siguiente forma:

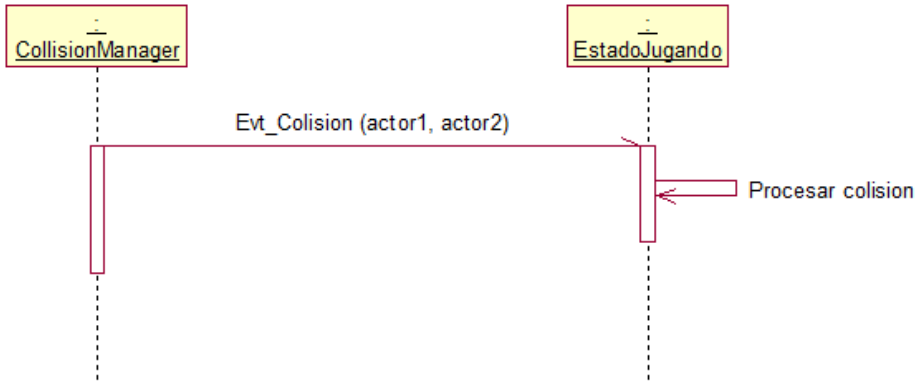
Movimiento

El evento Evt_Mover es disparado en las vistas y se encarga de pedir al motor que aplique el movimiento sobre la criatura. Por otro lado está el evento Evt_Camara, que solicita la transformación, tanto en posición como en rotación, de la cámara que sigue al personaje.



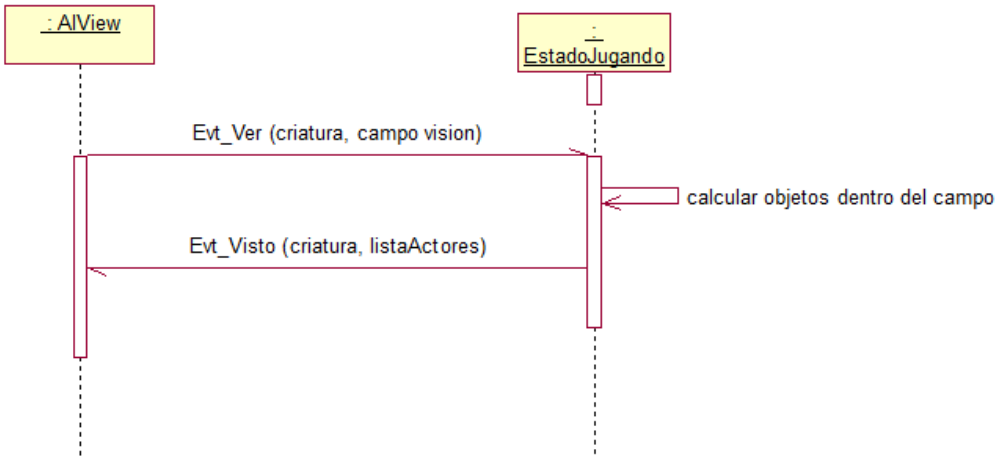
Física

El evento Evt_Colision es el que informa al estado de juego de qué actores han chocado para que éste pueda procesar la colisión.



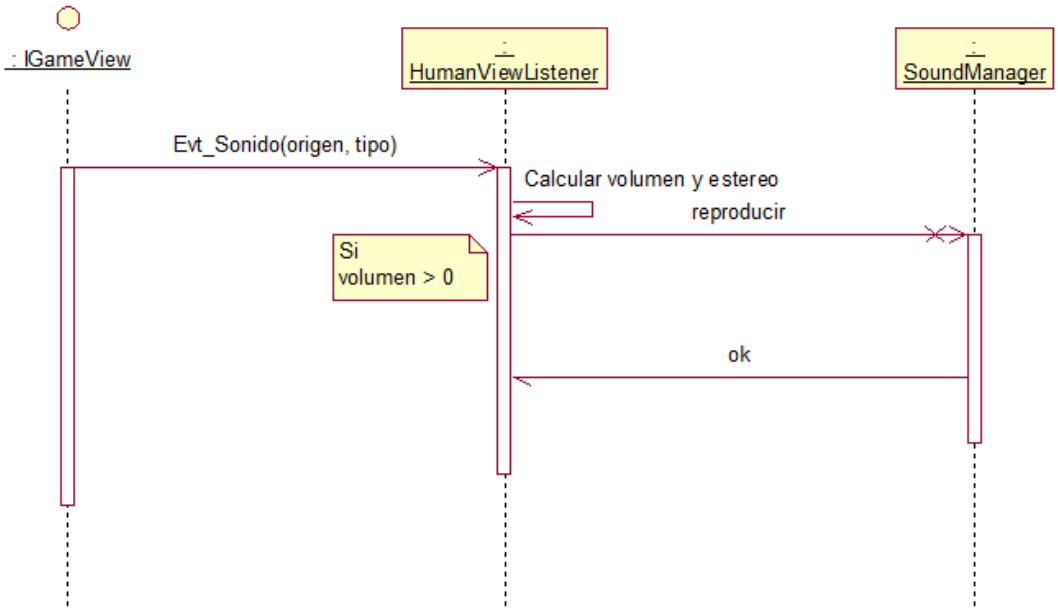
Visión

Se compone de dos partes: Evt_Ver realiza peticiones de visión especificando un campo de visión, y Evt_Visto, disparado en respuesta, indica qué se ha visto. El motor utiliza un sistema de selección volumétrica para saber qué hay dentro del campo de visión.



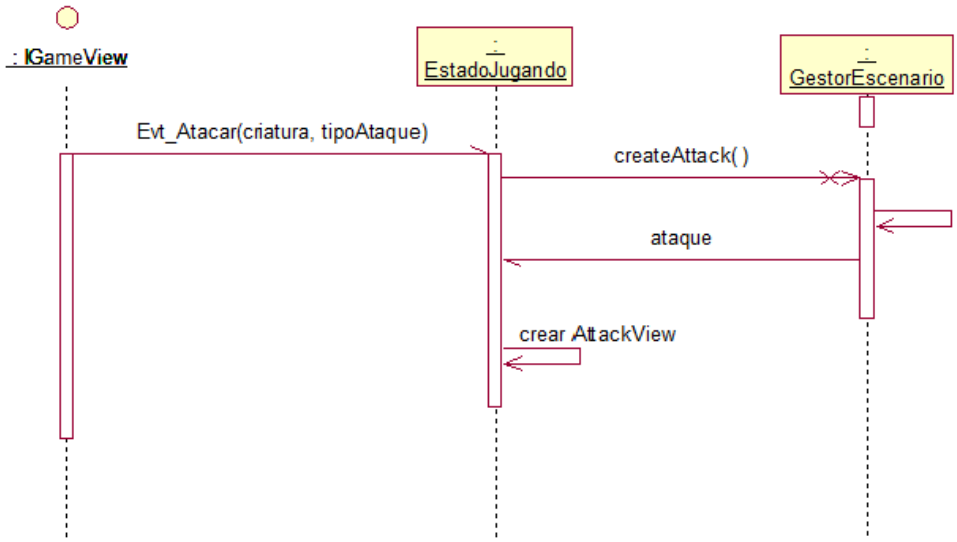
Sonido

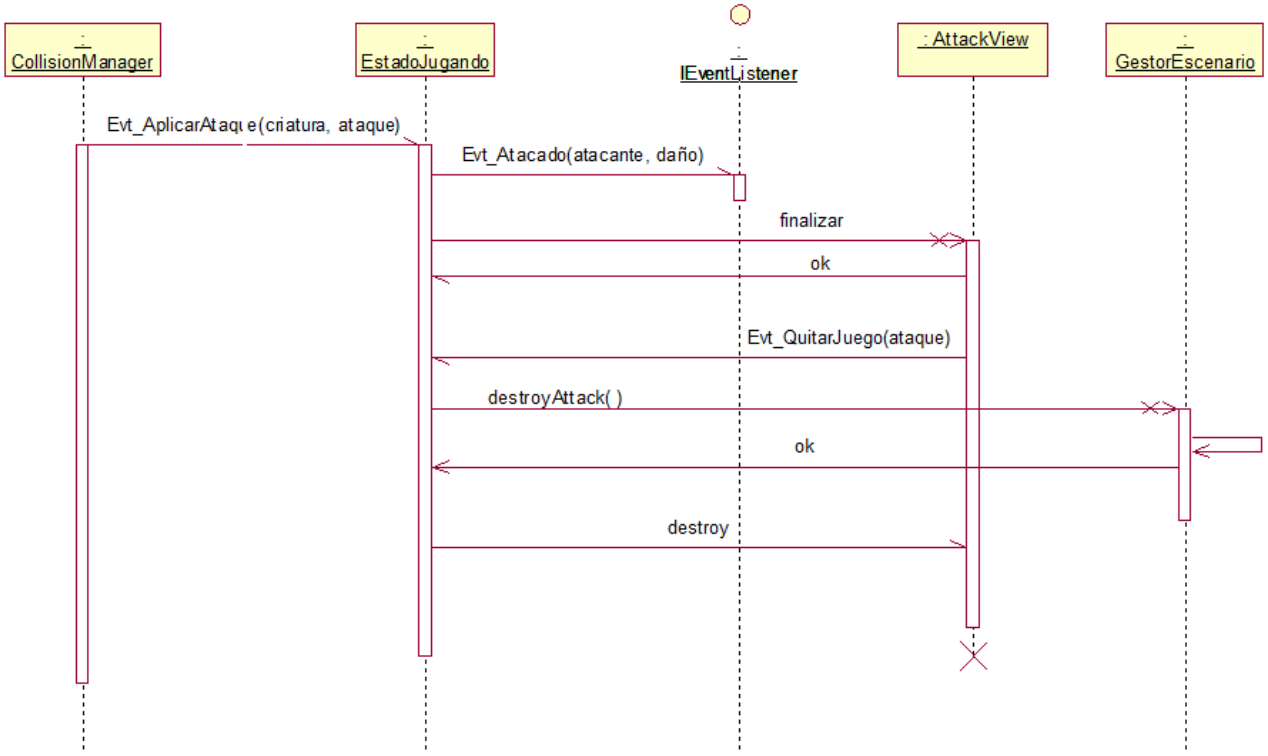
Son disparados a través del evento Evt_Sonido en las diferentes vistas al realizar determinadas acciones (caminar, aullar, atacar, etc.); y recogidos principalmente por el HumanViewListener, que lo renvía al gestor de sonido.



Combate

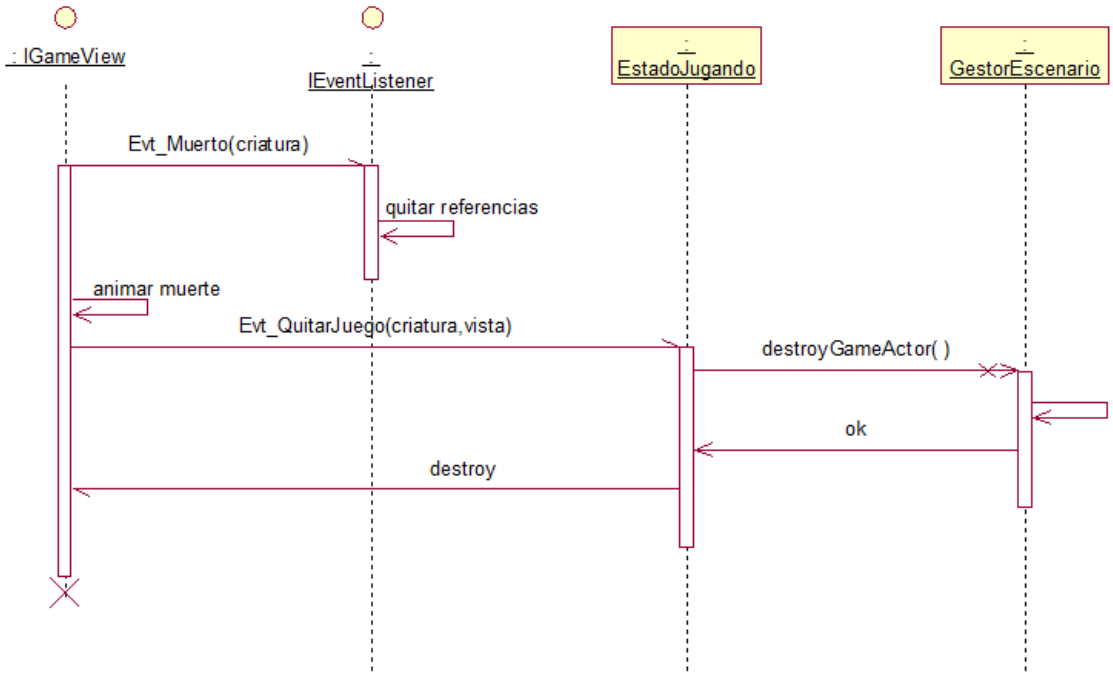
Los eventos de combate tienen tres partes: la declaración del ataque, mediante el evento Evt_Atacar disparado en las vistas; la detección del golpe gracias al motor de colisiones, mediante el evento Evt_AplicarAtaque , y la resolución de daño con el evento Evt_Atacado.





Destrucción

La destrucción de los criaturas consta de dos pasos: en un primer lugar se dispara el evento Evt_Muerto, que activa la animación de muerte y obliga al resto de actores a dejar de referenciar al actor destruido; y posteriormente se lanza el evento Evt_QuitarJuego, que elimina al actor completamente, descarga sus recursos, libera la memoria, etc.



Cabe destacar que se ha puesto la comunicación con los eventos de forma directa entre el disparador y el receptor por simplicidad. Debería ser el Gestor de Eventos el que recoge y distribuye los eventos, pero añadirlo no haría más que aumentar la complejidad de los diagramas y se perdería la idea general que proporcionan.

5.-Inteligencia Artificial

5.1.-Introduccion

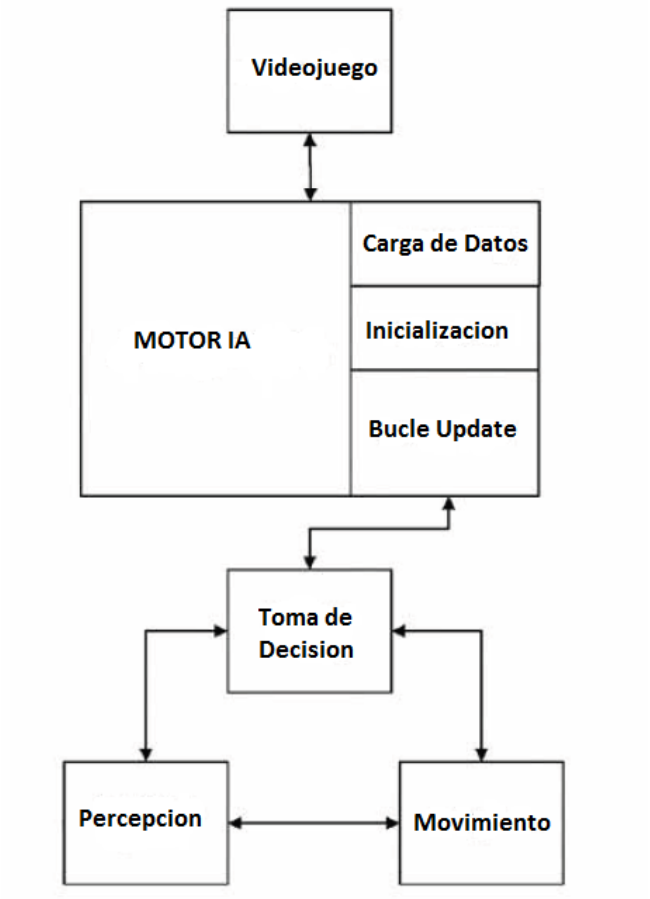
El objetivo de esta parte del proyecto es desarrollar un motor de inteligencia artificial para videojuegos. Este motor debe ser independiente de la lógica del juego, el motor de física y la parte de renderizado del juego. Es decir si un videojuego cambia de plataforma probablemente cambiarán la física y el renderizado pero el motor de Inteligencia Artificial no debe sufrir ninguna modificación. Esto es posible si orientamos nuestros diseños a eventos.

El motor de inteligencia artificial que hemos diseñado consta de tres partes fundamentales:

-Percepción

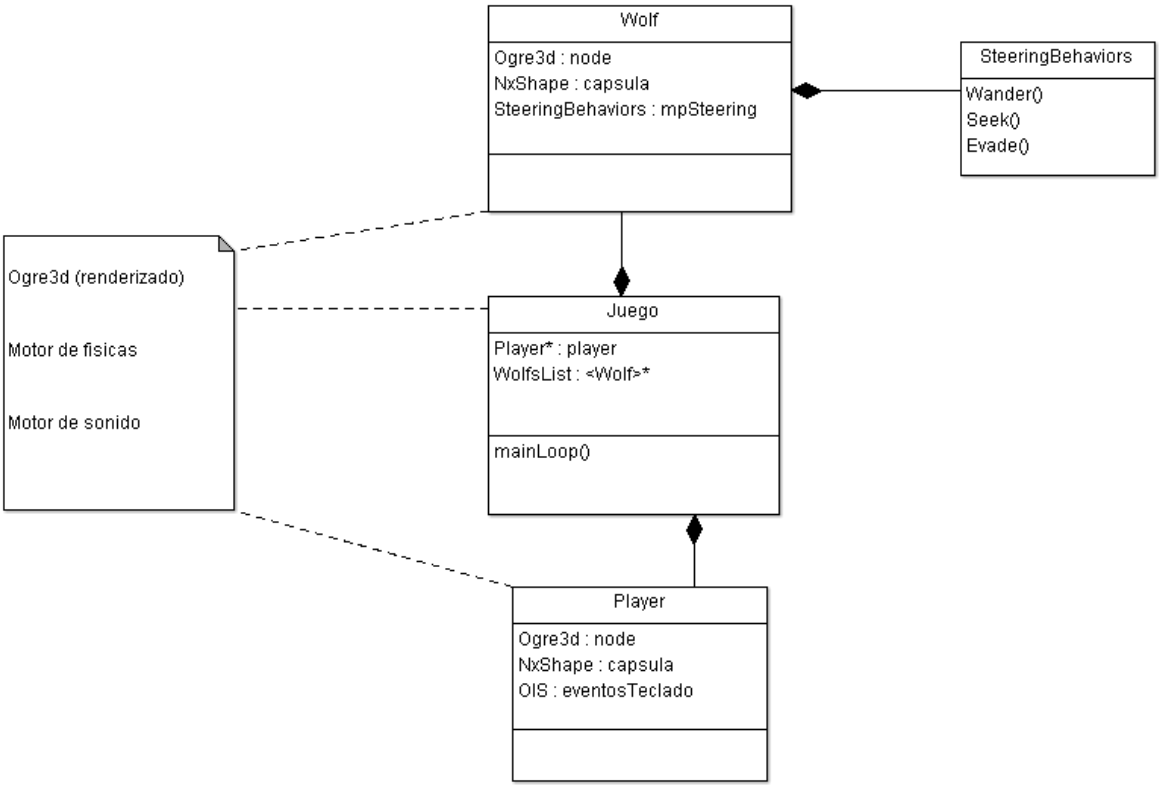
-Toma de decisión.

-Movimiento



5.2.-Estructura del motor de Inteligencia Artificial

Cuando comencé a programar videojuegos no tenía ni idea de como programar un motor de inteligencia artificial así que diseñaba mis juegos con un enfoque orientado a objetos, es decir, cada clase realizaba su función.



Diseño de un juego programado por un principiante

Sin embargo comencé a darme cuenta de que la siguiente arquitectura nos traía los siguientes problemas.

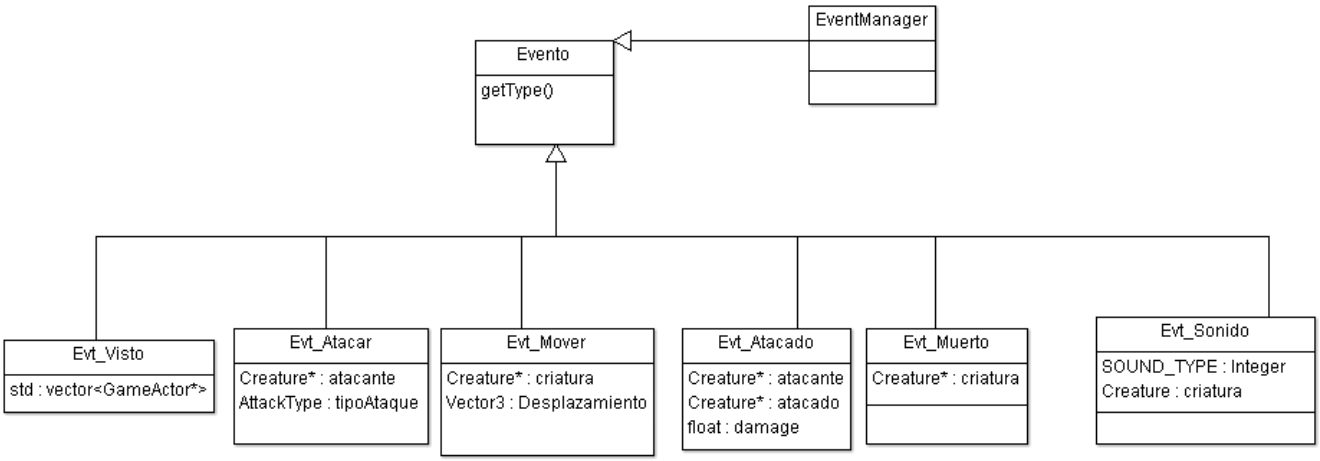
-Todas las clases está fuertemente acopladas con el motor de físicas, el motor de renderizado y el sonido por lo que si queremos usar esa inteligencia artificial en otra plataforma probablemente cambien los motores y tendremos entonces que modificar todas las clases que llevan asociados comportamientos de inteligencia artificial.

-Tratamos a los jugadores humanos y a los personajes de inteligencia artificial de forma diferente. Si queremos extender la funcionalidad de nuestro juego para que sea online deberemos de gestionar todos los personajes de la misma manera.

Ese independencia se conseguirá utilizando vistas de inteligencia artificial y un sistema de eventos.

5.3.-Interaccion del motor de IA con el resto del Juego

La IA se comunica con el resto del juego mediante un sistema de eventos. Cada evento llevará información asociada. Por ejemplo un evento de movimiento llevará un puntero a la criatura que se quiere mover y un vector que contiene el desplazamiento. Sin embargo un evento atacado llevará información del atacante, el atacado y el daño realizado.



Tipos de Eventos

Estos se eventos son gestionados por los listener de cada vista de inteligencia artificial y por el manejador de eventos.

Para explicar el funcionamiento del motor de eventos vamos a poner un ejemplo. Un lobo ve a un ciervo y le comienza a perseguir.

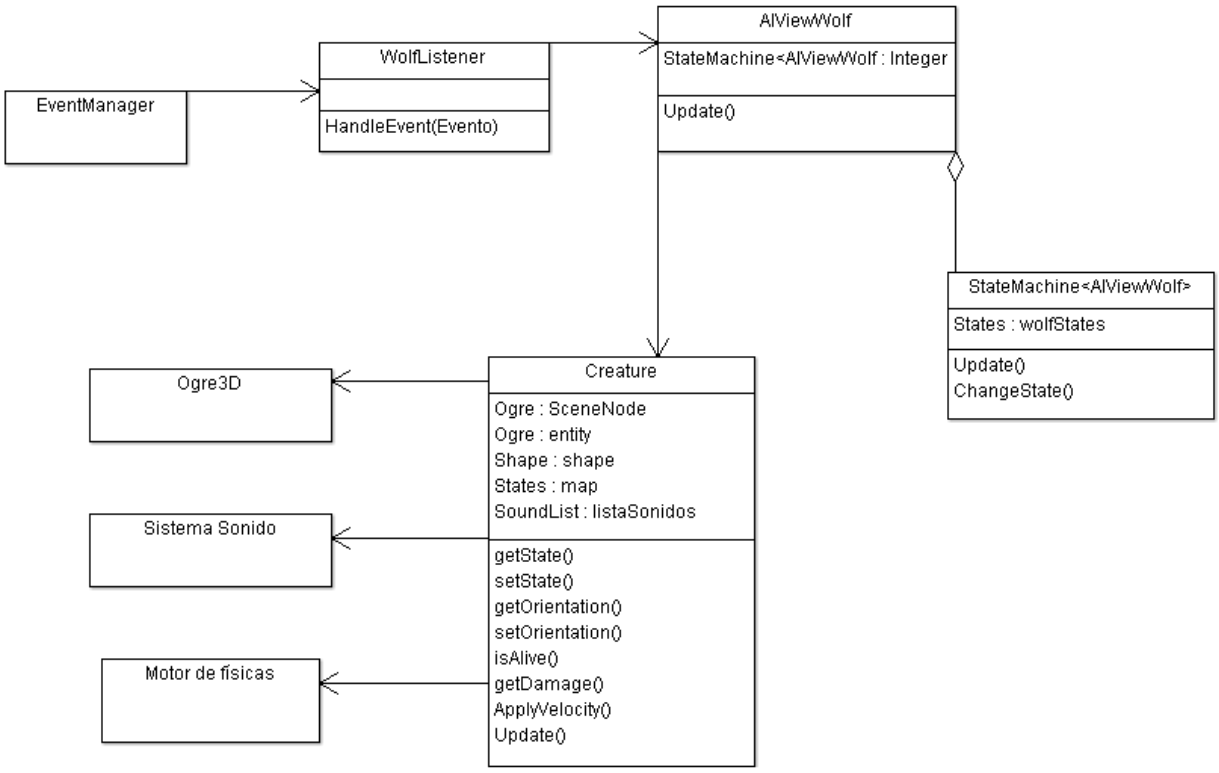
1.- El listener de la clase AIViewWolf recibe un evento de visión con los actores que ve en ese momento, entre ellos un ciervo

2.-La vista del lobo cambia al estado atacar en su maquina de estados.

3.-Se ejecuta el estado atacar y se obtiene el vector de desplazamiento para empezar a perseguir a dicho ciervo.

4.-La vista de inteligencia artificial lanza un evento mover con dicho vector de desplazamiento.

5.-El manejador de eventos recibe el evento mover del lobo y aplica dicho desplazamiento a la



criatura asociada con la vista de inteligencia artificial de dicho lobo.

5.4.-Percepcion

En la clase AIViewListener recibimos los eventos. Con esos eventos tomaremos decisiones.

Por ejemplo en la clase AIViewWolf listener recibimos los siguientes eventos:

Evento Ver

El evento ver nos proporciona la lista de actores que vemos en cada momento. Esa lista contiene tanto actores que son criaturas como obstáculos

Evento Atacado

Bajará la vida del lobo

Evento Aullar

Los lobos aullan cuando ven a un enemigo por primera vez. Con el evento aullar el lobo sabrá que otro lobo ha aullado.

```
bool AIViewWolfListener::HandleEvent( GardeGames::Event & evento ) {
        GardeGames::IEventManager *evtMgr=GardeGames::IEventManager::Get();
         std::vector<GameActor*> obstaculosVistos;
       if (stricmp(evento.getType().getStr(),Evt_Visto::gkName)==0)
         {
                  while (!evt.mResultado.empty())
              {
               GameActor * actualActor=evt.mResultado.back();
                      int tipo =actualActor->getType();
                       if (tipo==Creature)//si es una criatura
                          {
                           //tomar decision
```

```
                        }
                          else if (tipo==Obstaculo)
                          {
                             //tomar decision
                          }
                        evt.mResultado.pop_back();
                }
                   //ACTUALIZAR LA PERCEPCIÓN DE OBSTACULOS
                   UpdateObstaclesPerception(obstaculosVistos)
          }
        else if (stricmp(evento.getType().getStr(),Evt_Atacado::gkName)==0)
         {
            //tomar decision
         }
        else if (stricmp(evento.getType().getStr(),Evt_Aullar::gkName)==0)
        {
          //tomar decision
        }
}
```

5.5.-Toma de decisión

Para la toma de decisión se han utilizado máquinas de estados. Aunque existen otras alternativas como comportamiento orientado a metas me he decantado por la sencillez y flexibilidad de los autómatas

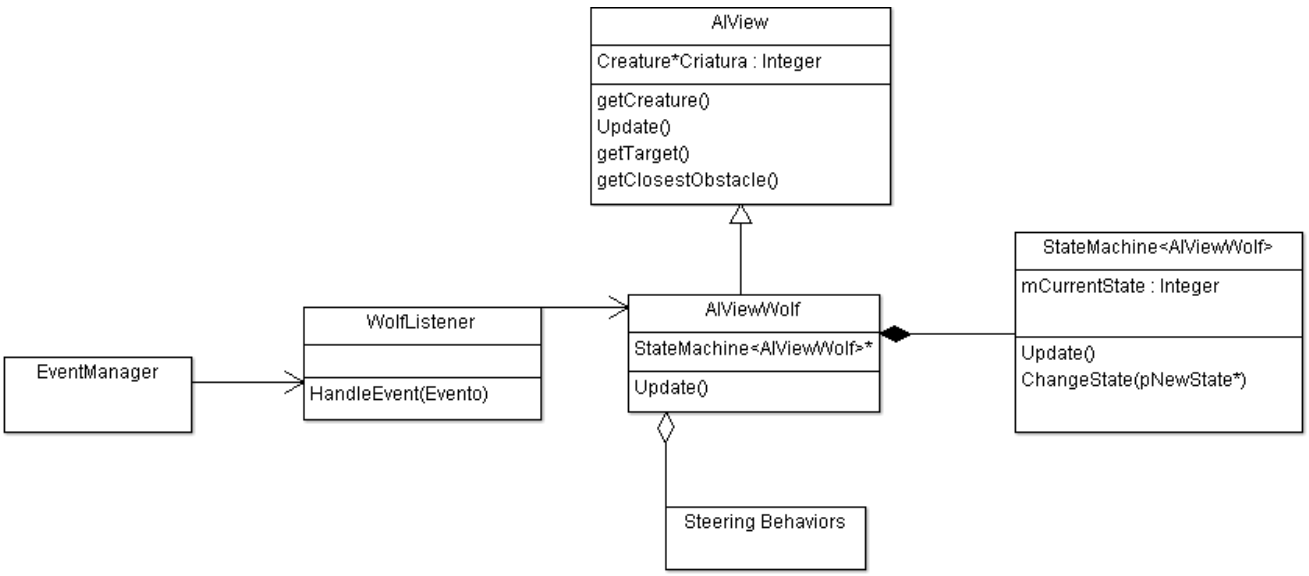
Cada vista de inteligencia artificial tendrá su máquina de estados. El manejador de eventos mandara los eventos a los listeners de las vistas de inteligencia artificial y estos controlarán dicha máquina de estados.

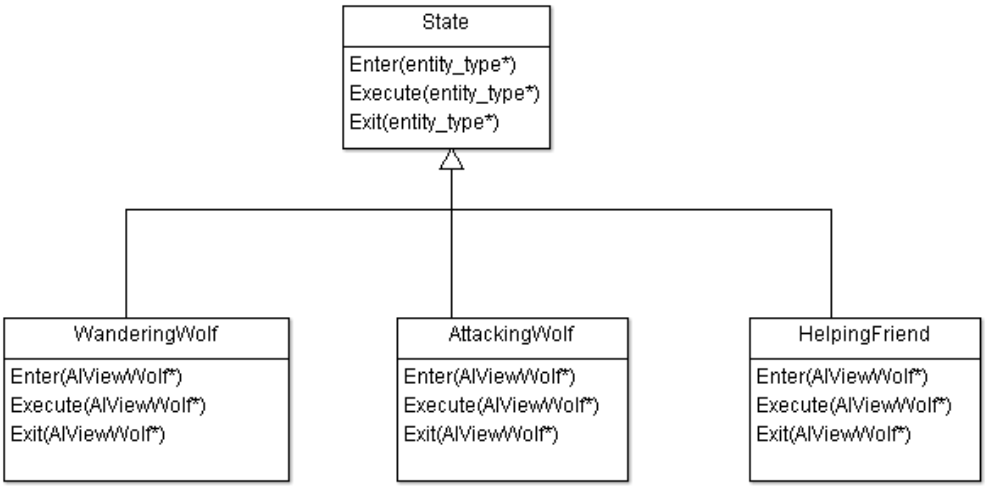
Por ejemplo, los estados del lobo son los siguientes:

Vagar-El lobo ejecuta el algoritmo de vagar y vaga por el escenario.

Atacar- El lobo persigue a una presa. Se activa cuando el lobo ve una presa.

HelpingFriend- El lobo se dirige a una posición concreta para ayudar a otro lobo. Este estado se activa cuando se recibe el evento aullido.





Estados del lobo

Cuando se entra en el estado vagar se activa el algoritmo de wandering. Mientras se sigue en dicho estado se ejecuta dicho algoritmo y se envían eventos de movimiento para que se mueva la criatura correspondiente. Al salir del estado vagar se desactiva el algoritmo correspondiente.

Estado Wandering

```
WanderingWolf* WanderingWolf::Instance()
{
  static WanderingWolf instance;
  return &instance;
}

void WanderingWolf::Enter(AIViewWolf* pAIWiewWolf)
{
        pAIWiewWolf->getSteeringBehaviors()->wanderOn();
}
void WanderingWolf::Execute(AIViewWolf* pAIWiewWolf)
{
     Ogre::Vector2 velocity=pAIWiewWolf->addForce(pAIWiewWolf->getSteeringBehaviors->PrioritiesAlg();
     GardeGames::IEventManager *evtMgr=GardeGames::IEventManager::Get();
     Ogre::Vector3 mov(velocity.x/800,0,Ogre::Math::Abs(velocity.y)/2);
```

```
    evtMgr->queueEvent(GardeGames::EventPtr( new Evt_Mover(pAIWiewWolf->mCreature,mov)));
}
void WanderingWolf::Exit(AIViewWolf* pAIWiewWolf)
{
        pAIWiewWolf->getSteeringBehaviors()->wanderOff();
}
```

5.6.-Movimiento (Steering Behaviors)

Los Steering Behaviors son algoritmos de movimiento dinámico, es decir, tienen en cuenta las propiedades cinemáticas completas del personaje y producen una aceleración para cambiar su velocidad actual. Para ir de un punto a otro, el algoritmo dinámico hace que el personaje acelere hacia dicho punto, cuando se va acercando trata de frenar progresivamente para detenerse completamente en el destino.

6.1Actualización del movimiento

Cada SteeringBehavior devuelve una fuerza (Steering Force)

Utilizando la ley de Newton

$F = m.a$

Despejamos la aceleración

$a = \frac{F}{m}$

Utilizando la formulav $v = v_o + at$ transformamos esa aceleración en velocidad usando el tiempo entre el frame actual y el anterior.

El código que se ejecuta en cada frame es el siguiente:

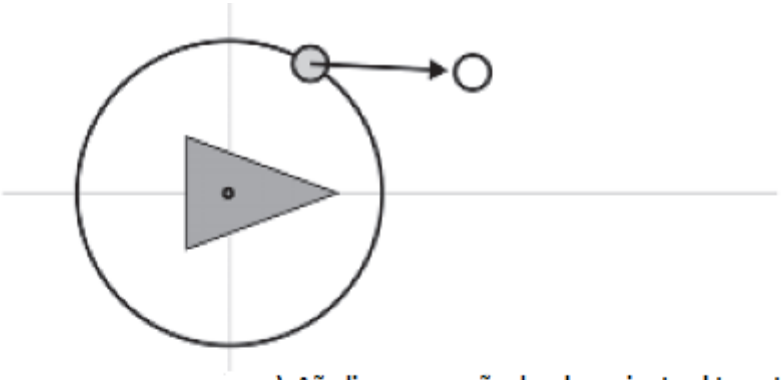
```
Ogre::Vector2 acceleration=steeringForce/masaCriatura;
```

```
double time=getTimeSinceLastFrame();

double vx=getVelocity().x+acceleration.x*getTimeSinceLastFrame();

double vy=getVelocity().y+acceleration.y*getTimeSinceLastFrame();
```

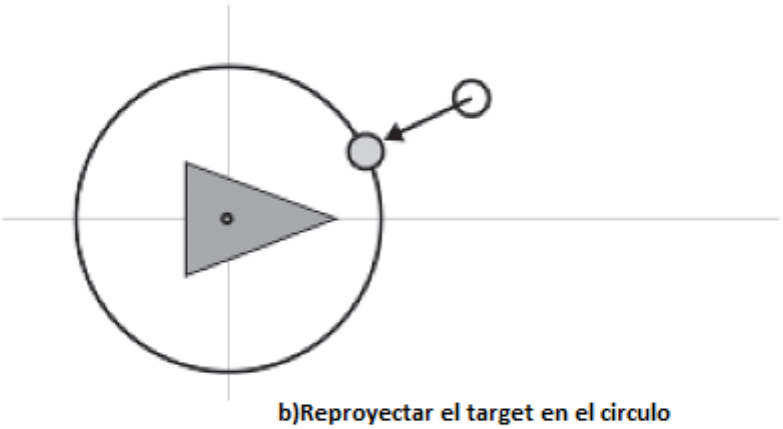
Wandering

Con este comportamiento los personajes vagan aleatoriamente por el escenario.

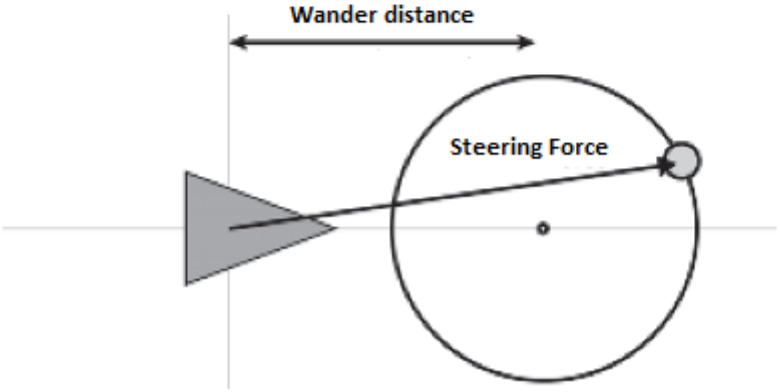
La primera vez que se ejecuta este algoritmo se elige un target aleatoriamente. En cada iteración se va modificando ese target usando proyecciones y desplazamientos mediante un circulo cuyo centro es el personaje que se mueve. El radio de dicho circulo y la distancia de proyección determinarán la cantidad de movimiento que sufre el personaje.



a) Añadir un pequeño desplazamiento al target



b)Reproyectar el target en el circulo



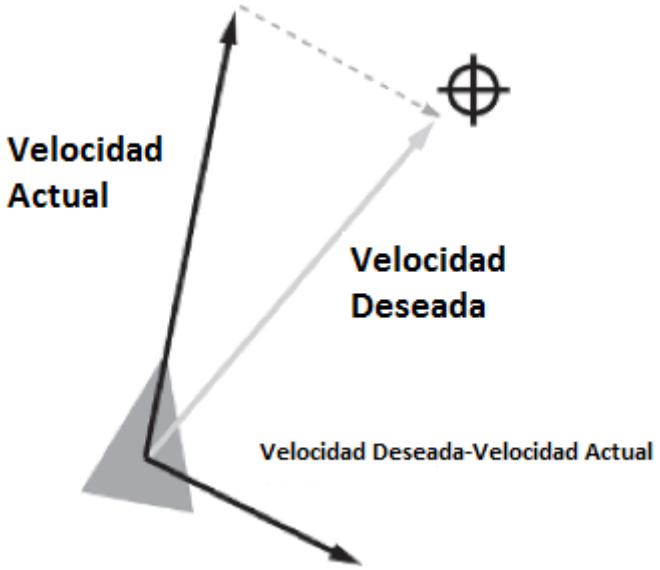
c)Proyectar el target en frente del circulo

```
Vector  SteeringBehaviors::Wander()
{

  //normalizar el tiempo entre frames
  double JitterThisTimeSlice =  WanderJitterPerSec * mAiView-
>getTimeLastFrame
```

```
  // añadir un pequeño vector aleatorio a la posición del target
  m_vWanderTarget +=Ogre::Vector2(RandomClamped() *timeSinceLastFrame,
RandomClamped() *timeSinceLastFrame)

  //reproyectar el vector en un circulo unitario
  m_vWanderTarget.normalise();

  //incrementar la longitud del vector para que sea igual que la del
radio
  //of the wander circle

  m_vWanderTarget *= WanderRad;

  //mover el target enfrente del agente
  Ogre::Vector2 target = m_vWanderTarget +
Ogre::Vector2(0,WanderDist);

  //proyectar el target en el espacio
  Ogre::Vector2 Target = globalPosition(target,mAiView-
>getOrientation(),mAiView- >getPosition());

  //girar

  Vector aux=Ogre::Vector2(-(Target.x-mAiView->getCreature()-
>getPosition().x),Target.y-  mAiView->getCreature()->getPosition().z);

  return aux;
}
```

Seeking

Con este comportamiento lo que se quiere conseguir es perseguir a un objetivo.




```
Vector SteeringBehaviors::Seek(Vector TargetPos)
{
  Vector DesiredVelocity = Normalize(TargetPos - mAiView-
>Pos())*mAiView->MaxSpeed();
   return (DesiredVelocity - mAiView->Velocity());
}
```

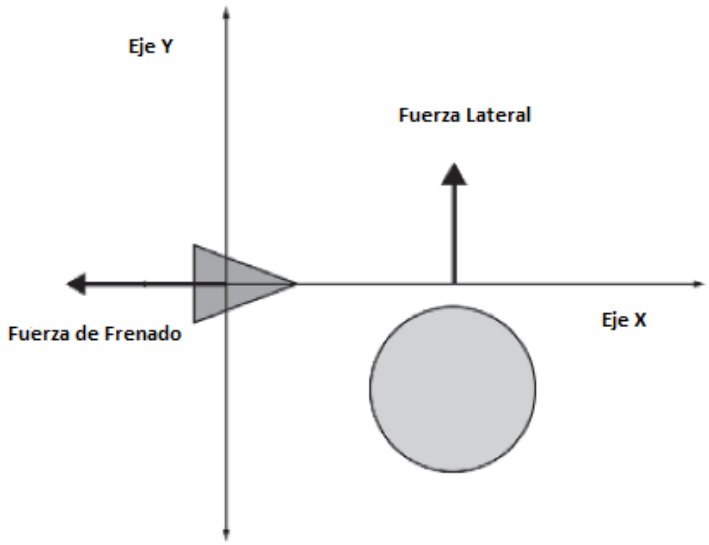
Flee

Este comportamiento sirve para escapar de un enemigo. El algoritmo es el mismo que Seeking cambiando únicamente la velocidad deseada que es de sentido contrario.

```
Vector SteeringBehaviors::Flee(Vector TargetPos)
{
  Vector DesiredVelocity = Normalize(mAiView->Pos()-
TargetPos)*mAiView->MaxSpeed();
   return (DesiredVelocity - mAiView->Velocity());
}
```

Obstacle Avoidance

Mediante este comportamiento conseguimos que nuestros personajes no choquen contra obstáculos.




```
SteeringBehaviors::ObstacleAvoidance(obstaclePosition,double obstacleRadius)
	{
		double multiplier=1;
		Ogre::Vector2 steeringForce;
		//calculate  lateral force
		Ogre::Vector2
	localObstaclePosition=localPosition(obstaclePosition,mAiView-
>getCreature()->getPosition(),mAiView->getCreature()->getOrientation());
		steeringForce.x=(obstacleRadius-
localObstaclePosition.x)*multiplier;

		//calculate the braking force
		double brakingWeight=0.2;

		steeringForce.y=(obstacleRadius-
localObstaclePosition.y)*brakingWeight;

		return  globalPosition(steeringForce,mAiView->getCreature()-
>getOrientation(),mAiView->getCreature()->getPosition());

	}
```

Combinación Steering Behaviors

En ocasiones se puede dar que haya dos comportamientos de movimiento simultáneamente. Por ejemplo cuando estas huyendo de un enemigo y te encuentras un obstáculo en el camino. Para resolver este problema se han intentado hacer muchos algoritmos. Algunos consisten en hacer una media de los vectores Steering Force otros consisten en redes neuronales y algoritmos genéticos. Ninguno de estos enfoques ha proporcionado una solución correcta. La única solución es utilizar un algoritmo de prioridades y realizar un comportamiento sin mezclarlos. En nuestro juego solo hacemos Obstacle Avoidance si tenemos un obstáculo lo suficientemente cerca. En caso contrario aplicamos el algoritmo actual:seek,wander o evade.

```
Ogre::Vector2 SteeringBehaviors::PrioritiesAlgorithm()
       {

          if (mAiView->isClosestObstacle())
            {
               obstacleAvoidanceForce=ObstacleAvoidance(closestObstacle-
              >getPosition()),closestObstacle->getRadius());

                  if (obstacleAvoidanceForce.length()>priorityThreshold)
                     {
                            return obstacleAvoidanceForce;
                     }
                  else
                     {
                            Ogre::Vector2
steeringForce=CalculateSteeringForce();
                            return steeringForce;
                     }
              }
              else
              {
                   return steeringForce;
              }
           }
       }
```