# Proyecto

X-Kating

## Grupo

WASTED HORCHATA

Documento de diseño técnico del motor de IA

Hito: 1
Fecha entrega: 22-12-2017
Versión: 2

Componentes:

- Luis González Aracil
- Laura Hernández Rielo
- Adrián Francés Lillo
- Pablo López Iborra
- Alexei Jilinskiy

# 1. Descripción general del motor

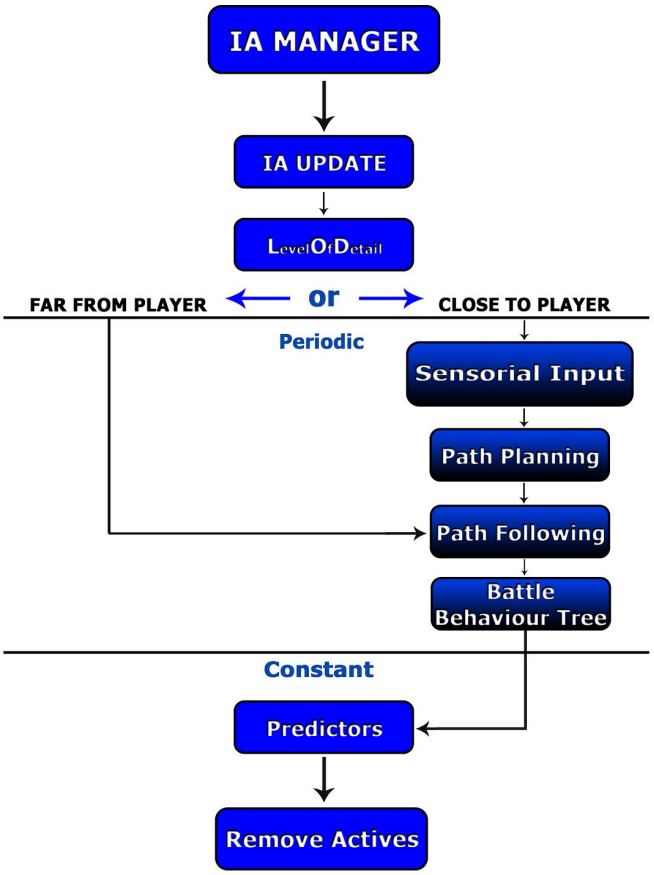
El **motor de IA** se encargará de mover y definir el comportamiento de una serie de adversarios para el jugador. Debe de intentar cumplir los siguientes objetos:

- Los adversarios deberán de **responder** a los estímulos externos **de una manera realista.**
- El motor debe ocupar **el menor tiempo de CPU posible.** Como primera meta al menos el **5% del total.**
- Su objetivo es plantear un **enfrentamiento justo pero** que suponga un **reto** al jugador.

Para conseguir eso haremos uso de **una arquitectura** que tenga **categorías prioritarias** sobre tareas, además de algunas técnicas de **balanceo de carga.**

El elemento principal para la gestión del tiempo de ejecución de la IA será el **gestor de recursos**, que **asignará tiempo de proceso** a cada elemento (Scheduling). Con ello consigue **dividir** el **tiempo** de ejecución de **forma equitativa** entre cada iteración y **evita** grandes **sobrecargas** en puntos donde deban tomarse varias decisiones de golpe.

## 2. Estructura general del motor




## 3. Descripción del diagrama

El **IA Manager** se encarga de gestionar el **funcionamiento del sistema**, y es el que da paso a cada Update de la IA. Cada **Update** de la IA lleva la información relativa del gestor de recursos y se **gestiona** la cantidad de **elementos** que **deben actualizarse, el tiempo de proceso** para **cada proceso "largo" y** en qué **orden deben de ejecutarse**.

Luego el **Level of Detail** (**LoD**), dependiendo del tiempo de ejecución y otras variables, **decide el nivel de complejidad** que se aplicará al update de la IA **de cada NPC**. La **principal decisión** será **en base** a una **distancia** prefijada **relativa al player**, que se verá determinada **por el quadtree** que divida la escena y clasifique a los objetos en diferentes nodos.

Luego se **ejecuta** el **comportamiento de la IA. Según** el **ciclo** en el que nos encontremos, se ejecutan funciones **periódicas y constantes.**

Las funciones **periódicas** se repiten cada tantos updates o tiempos asignados. Consisten principalmente en:

1. **Sensorial inputs:** Los NPCs obtendrán información de:
    a. **Su cono de visión**.
    b. Variables locales de **posición** y **velocidad**.

2. **Path planning:** usando el sistema de **WayPoints y** nuestra información del objeto como **posición y velocidad,** obtenemos los siguientes puntos objetivo de la pista en el camino para determinar hacia donde debe de ir el NPC. Además, el punto seleccionado en el WayPoint será **más lejano cuanto mayor sea la velocidad** del NPC.

3. **Path following:** El NPC **valorará** el **punto** escogido por el path planning y **analizará** los puntos de **colisión** obtenidos de los sensores de visión. En caso de que se encuentre en rumbo de colisión, virará en la dirección adecuada.

    Los resultados de la ponderación, invocarán las acciones de movimiento, giro, etc… a través del **input manager** de la misma manera que si el NPC **usase un joystick.**

4. **Battle Behaviour Tree:** El NPC lanza el objeto que **tenga en el inventario.** Usa un **árbol de comportamiento** que determine **cuándo** lanzar el objeto para obtener el **mayor rendimiento** posible del objeto. **Este árbol recibirá:**
    a. La posición actual del player
    b. La posición cercana de los demás jugadores
    c. Sus posiciones en el marcador

d. Variables sensoriales pertinentes.

El resultado debería de ser un lanzamiento que reflejase una ventaja para el NPC frente a los demás jugadores, siendo **más óptimo cuanto más difícil** sea el nivel de la IA.

Las funciones **constantes** deben de realizarse en cada update. **Consisten** en: **tareas compensatorias y la etapa de exclusión.**

Las **tareas compensatorias**:

1. **Actualizan el comportamiento** de la IA que se ha quedado sin órdenes este frame.
2. **Realizan cálculos sencillos.**
3. Decididas por el Resource Manager o por la periodicidad de otras funciones.
4. **Casos de uso:**
    a. **Interpolación lineal** del movimiento de un NPC.
    b. **Evitar los cálculos** del "Sistema de batalla" cuando un NPC no ha pasado por encima de una caja de objetos.
    c. **Evitar cálculos** de Path planning o following cuando un jugador ha sido noqueado (aunque dure un segundo o dos).

La **etapa de exclusión** quita los componentes a actualizar en el update. **Casos en los que se da esta etapa:**

1. **Objeto** es **destruido** en el juego
2. **Objeto** es **desactivado** durante X tiempo.
3. **Objeto** es **paralizado**.
4. **Objeto** con el **componente** de **IA desactivado** o destruido por un determinado proceso.

## 4. Sistema de input sensorial

El input sensorial se recogerá en dos sensores diferentes: el **VSensor y el MSensor.** El VSensor recibe del entorno lista de componentes de colisión presentes en el mundo. El MSensor recibe las

variables del terreno y devuelve los datos visualizados por el gameObject();

**Los lados de los sensores se construyen de la siguiente manera:**

```
//Inicializando variables
angleInitial=angI;
angleVision=angV * 3.141592653589f / 180.f;

//Cálculo de los sensores
sensorLeft = glm::vec3(cos(angleVision+angleInitial), 0.f, sin(angleVision+angleInitial));

sensorRight = glm::vec3(cos(-angleVision+angleInitial), 0.f, sin(-angleVision+angleInitial));
```

La variable "*angleInitial*" **consiste en** la variable **posición angular inicial** del sensor, que al girar a la misma velocidad que el personaje siempre estará en la posición adecuada. Luego, "*angleVision*" e**s el ángulo de visión** a izquierda y a la derecha que dentro de este rango es donde veremos los objetos de colisión.

**Los sensores** de izquierda y derecha **son** dos **vectores unitarios**, al ser resultado de las operaciones de coseno y seno. Cada punto dentro del rango que abarcan estos dos sensores supone un obstáculo o punto ventajoso al que acercarse.

Para obtener información de este rango, se calculan las variables A y B relativas a la posición del NPC que contiene los sensores. Estos valores A y B son dos **floats** que al **multiplicarse** por el **sensor izquierdo y derecho**, resultan en la posición del punto objetivo.

*a * sensorLeft + b * sensorIzq = PosiciónObjeto - PosiciónActual ;*

Al obtener la información, **si** los valores **A y B** son **positivos,** significa que el objeto está **dentro del rango de visión** de estos vectores.

Una vez obtenidos estos valores, **se guardan en** un **VObject** (Tipo de dato que contiene información relativa a las posiciones de un objeto para ser tratada por la IA) y se meten dentro de una **lista de objetos vistos.** Esta lista luego puede ser recogida para ser consultada por otros componentes.

**El MSensor** no utiliza el área que comprenden ambos vectores, sino que **calcula la colisión de ambos sensores con la pared** a la que está mirando el objeto. Para ello solicita al PhysicsManager la información del terreno sobre el que se encuentra el GameObject, y **calcula con cada pared** la colisión de la extensión del sensor con esa pared. Para ello lo que se hace es resolver esta ecuación y se calculan los términos A y B:

```
(position + a*sensorLeft) - (p_B + b* vAB) = 0 ->
    position + a*sensorLeft - p_B - b* vAB = 0 ->
        a * sensorLeft + b * (-vAB) = p_B - position;
```

donde el punto p_B corresponde a una de las puntas del terreno y el vector vAB el que va desde ese punto del terreno hasta la otra. **Si el resultado da entre 0 y 1 para la B,** significa que el punto es “b” veces el vector vAB, y está en la pared. Una vez determinado ello, se calcula con el otro sensor y se guarda en el vector de objetos vistos.

## 5. Sistema de Lógica difusa

La **lógica difusa** es el **sistema** por el cual el NPC decidirá hacia dónde **dirigirse** en base a la información que reciba del exterior. Los valores de entrada para **alimentar** este sistema **se los da** el **VSensor y el MSensor**: una lista de **VObjects** por parte de ambos, un vector apuntando a un waypoint y sus valores A y B.

Se aplica una función de **arcotangente** a los valores **A y B** para obtener el giro relativo respecto a la orientación del objeto de dónde se encuentran estos objetos. Para el WayPoint se determina su **grado de pertenencia a** unas variables de giro **(izq, centro, dech)** con una **interpolación lineal en los extremos y triangular en el centro. Para** los **obstáculos** y paredes se aplica la misma operación salvo que en ellos el valor **se acumula** en obs_izq, obs_centro y obs_dech, **dividiendo entre** el **total** de elementos. Así se consigue ponderar el punto medio donde se concentran los obstáculos y esquivar ese punto.

```
//Limits: -0.75 - 0.25 right, 0.25 center, 0.25 to 1.25 left
float wp_left   = inferL(atan_w     ,0.25f  ,0.8f   ,0      );
float wp_center = inferT(atan_w     ,0.23f  ,0.25f  ,0.27f  );
float wp_right  = inferL(atan_w     ,-0.25f ,0.25f  ,1      );
```

Una vez obtenidas estas variables, **se usan reglas para** determinar el valor **aproximado de 0 a 1** de cuánto debería de girar el objeto a un lado o cuánto debería de estar quieto, en nuestro caso se llaman **steeringLeft, steeringNone y steeringRight.** Aplicando las reglas lógicas de AND, OR y NOT en lógica difusa, las aplicaremos como el **Máximo, Mínimo y (1 - valor) respectivamente.** Entonces anidando estas reglas podremos obtener este ruleset para definir los valores de giro.

```
steeringLeft  = glm::min( glm::max(wp_left, wp_center), glm::max(obs_center, obs_right) );
steeringNone  = glm::min( glm::max(1-wp_left, wp_center, 1-wp_right), glm::max(obs_left, obs_right) );
steeringRight = glm::min( glm::max(wp_right, wp_center), glm::max(obs_center,obs_left) );
```

**Finalmente** se le da un **valor concreto** a las reglas **defuzzificándolas**. Es decir, **cogeremos** el conjunto de **valores de 0 a 1** y las **meteremos en un set** como si hiciéramos el proceso inverso a fuzzificar. Usando el **cálculo de Maldani,** obtendremos el punto de mayor densidad de la decisión. **El valor de 0 a 1** pertenece a un triángulo o trapezoide, que **al llegar a cierta altura formará** un **trapezoide** menor con un área concreta. Hallando el punto el **centroide** de estos trapezoides **y** obteniendo **el punto medio entre estos,** encontraremos la decisión óptima de la lógica difusa.

Esta decisión se llamará sobre el InputManager y luego el moveComponent aplicará la decisión para mover el objeto.

## 6. Sistema de PathPlanning

El **Waypoint Manager** se encarga de gestionar el **vector** de de **waypoints** del mapa, los cuales están ordenados por nivel. Este nivel le **determina** la sucesión y **orden** de los mismos, y le sirve al sistema de **Path Planning** a la hora de realizar los cambios de waypoint. El sistema está pensado para poder tener waypoints del mismo nivel, de manera que el NPC siempre irá al qué más le convenga de ese nivel, sin poder ir a uno ya pasado. Esté manager está compuesto por:

- **Vector waypointComponentList:** Este vector contiene todos los **WaypointComponent** qué han sido creados. Por cada **WaypointComponent** existe un objeto asignado, previamente creado.

- **Vector pathPlanningComponentList:** Vector con todos los **PathPlanningComponent**. De igual manera que el **WaypointComponent** y todos los otros componentes, tiene un objeto previamente asociado.

- **Vector listSubNodes:** Contiene todos los **GameObject** que contienen un **WaypointComponent**. Este vector es el que se utiliza desde **PathPlanningComponent** para realizar el cálculo de próximo punto.

El **Waypoint Component** se encarga de guardar la información individual de cada waypoint, de está manera cada uno de estos, tiene su propia información. Están compuestos por:

- **Float radius:** Rango de acción qué tendrá el waypoint, de tal manera que el NPC al entrar en está área, pasará a marcarse como objetivo el proximo waypoint.

- **Int level:** Nivel del waypoint. Este nivel determina el orden en el que los NPC's recorren los waypoints.

Como último componente encargado del Path Planning, tenemos el **PathPlanningComponent**, el cual obtendrá de forma individual:

- **Vector posición** del NPC.
- **Vector velocidad** del NPC.
- **Modulo de la velocidad** del NPC.

Este componente está formado por:

- **Float seconds:** segundos de procesamiento que realizará la IA. En caso de querer el mismo procesamiento que un Player, el valor debe ser 1.
- **Float distLastWay:** distancia al próximo waypoint.
- **Unsigned int lastVector:** Posición del próximo waypoint en el vector general de **WaypointManager.**
- **glm::vec3 nextPos**: Vector con la próxima posición a la que deberá ir el NPC.

Mediante un setter en **AIManager** fijamos el tiempo de cálculo que tendrá de IA, de manera que asignándole un valor mayor de 1, la IA podrá adelantarse en cálculos al Player. Esto se debe a que la diferencia del cálculo de la distancia a los waypoints, se realiza con el módulo de la velocidad y los segundos asignados.

El update de este componente, se encarga del cálculo de la nueva posición, de manera que todos los cálculos de distancias se realizan con

los cuadrados. Así se consigue evitar realizar raíces cuadradas y ahorrar tiempo de procesamiento.

- En primer lugar **se calcula el cuadrado de mi velocidad por los segundos asignados. Si la distancia** al próximo waypoint **es mayor** que la distancia que recorre con mi velocidad, **el punto** que guardara **será el del próximo waypoint o** un punto **interpolado** entre el waypoint y el NPC.

- En el caso de que tu recorrido vaya a ser mayor que la distancia al próximo waypoint, se calculará un punto interpolado entre los dos siguiente waypoints. Este cálculo se realizará con el resto de recorrido que quedaria despues de superar el waypoint.

- En el caso de que después de realizar el paso anterior, aún superemos la distancia al próximo, el punto de interpolación se calculará con el siguiente waypoint, y así de manera continua hasta obtener un punto fijo.

El cálculo de la posición interpolada, se realizada con la siguiente fórmula:

**nextPos = ((tour/dist) * (listNodes[lastPosVector].get()->getTransformData().position - listNodes[posVector].get()->getTransformData().position) + listNodes[posVector].get()->getTransformData().position);**

- **Tour:** Cálculo del procesamiento que realizará la IA. Se calcula como el módulo de la velocidad por los segundos de procesamiento anteriormente asignados.
- **Dist:** Distancia entre el NPC y la posición de dos waypoints después.
- **listNodes[]:** Vector de waypoints, obtenido de **WaypointManager.**

- **posVector:** Posición en el vector del próximo waypoint al qué voy.
- **lastPosVector:** Posición en el vector del waypoint posterior al de **posVector**.
- **getTransformData().position:** Posición, en el mapa, del waypoint pedido.

Para obtener el punto al que se dirigirá la IA, se realizará un getter desde **AIManager**, el cual le servirá a la lógica difusa para realizar sus cálculos.

## 7. Sistema de Batalla

El sistema de batalla, como parte integral de la IA, se encuentra también gestionado desde el **AI Manager**. Se tiene de forma local todos los objetos con una IA de batalla asociada, y desde allí se controla el tiempo de proceso con el que cuenta.

El funcionamiento principal del sistema de batalla viene definido en el **AIBattleComponent**, el cual internamente cuenta con los distintos tipos de nodos implementados para el Behaviour Tree.

La creación de la estructura completa del Behaviour Tree se realiza de la misma manera para cada componente, pues se considera que todos tendrán el mismo comportamiento para la gestión de ítems. Para ello, se llama a un método de inicialización que, partiendo de una variable del componente que indica la raíz del árbol se construye completamente creando todos los nodos necesarios para ello. La estructura del árbol completo está definida en el documento de diseño de espacio de decisión.

Además de este método, existe un update que se encarga de llamar al método Tick de la raíz cada vez que es llamado.

El Behaviour Tree, tendrá como posibles resultados en cada una de sus acciones, los siguientes estados:

- **INVALID:** Acción no valida.
- **RUNNING:** Estado en el que el sistema sigue una acción programada o que se realiza en un curso de tiempo. Podría ser el caso de moverse.
- **SUCCEEDED:** Acción en la que la condición se ha cumplido y puede ser completada. Este estado le sirve al behaviour tree para mandar la información de qué se debe realizar la acción.
- **FAILED:** Proceso que ha fallado y no se ha podido completar.
- **ABORTED:** Estado en el que una acción ha debido ser cortada debido a un posible error.

La estructura interna en la que se basa nuestro Behaviour Tree, está formada principalmente por:

- **Composite:** Contiene los hijos del behaviour, de manera que los va comprobando. Mientras realiza comprobaciones, le devuelve al nodo padre “n” resultado de RUNNING.
- **Decorator:** Únicamente contiene un hijo. Cambia el valor del resultado, a otro asignado, según el tipo de **Decorator** que se utilice.
- **Dynamic Selector:** Realizará comprobaciones en todos sus hijos, de manera que cuando uno de ellos de como resultado SUCCEEDED, cortará ejecución y mandará como valor true la acción, de manera que se realizará. En el caso de que ninguno de los resultados sea SUCCEEDED, repetirá la ejecución y las comprobaciones hasta que uno de los hijos se complete.
- **Repeat:** Realiza repeticiones en el behaviour tree, de manera que si no se completa alguna acción, se seguirán comprobando condiciones y acciones.
- **Selector:** Realiza las mismas comprobaciones que el **Dynamic Selector**, la diferencia es que no repite las comprobaciones desde el propio **Selector**, sino que vuelve al **Repeat.**
- **Sequence:** Comprueba todas las condiciones de la acción, en el caso de que todas den como resultado SUCCEEDED, se

completará la acción. En el caso de que una de ellas diese como resultado FAILURE, RUNNING o ERROR, se cortaria la acción y no se completaria.

A continuación se adjunta el Behaviour Tree utilizado: