Proyecto

X-Kating

Grupo

WASTED HORCHATA

Documento de diseño de toma de decisión: espacio y métodos

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. Introducción

En X-Kating tendremos NPCs durante las partidas. Serán otros jugadores con las mismas mecánicas de movimiento y acción que el player. Para una mejor experiencia de juego deben comportarse lo más realista y competitiva posible.

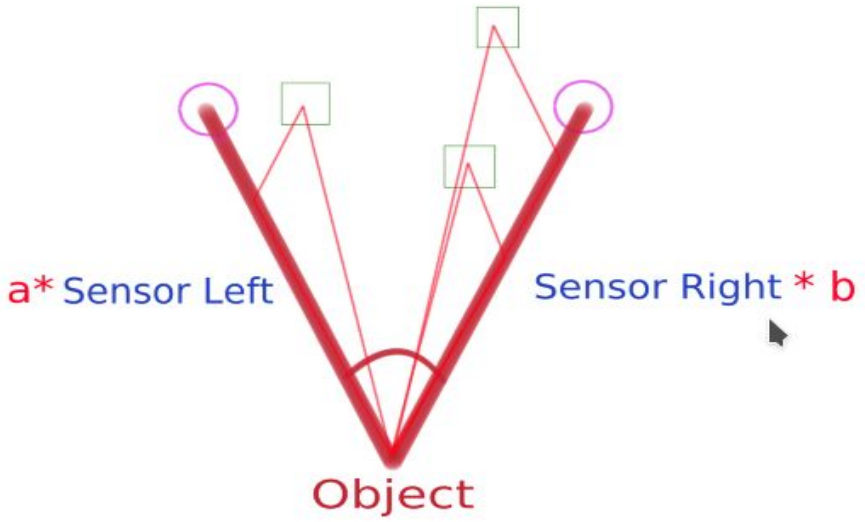
2. Espacio de decisión

a. Sensores

A los NPCs se les acoplarán **Sensores.** Los sensores obtienen su información del **PhysicsManager,** una clase que maneja las físicas y **devuelve** información del **mapa actual**. Las funciones que el Sensor debe de hacer son:

- **Ver en** el **cono** de visión **todos los objetos (colisiones, rampas, cajas y otros jugadores).**
- Devuelve un array de objetos de tipo **VObject,** compuesto por:
  - **Vector3** de la posición del objeto.
  - **Valores float a y b**, correspondientes a la composición vectorial del punto respecto a los límites laterales del sensor unitarios.
  - **Radio** del objeto desde su centro, para tener en cuenta su colisión en el radio y no en el centro.
  - **Length** del objeto, para tener en cuenta su colisión en 3D.
  - **Tipo** del objeto al que corresponde el **VObject**. Estos pueden ser:
    - Colisión.
    - Pared.
    - Rampa.
    - Caja.
- Al visualizar un punto que esquivar o al que ir, devuelve dos valores **A y B** que determinan la posición relativa de esos

objetos con **respecto** al **gameObject que contiene** al **sensor**.



Hay **dos** tipos de **sensores:**

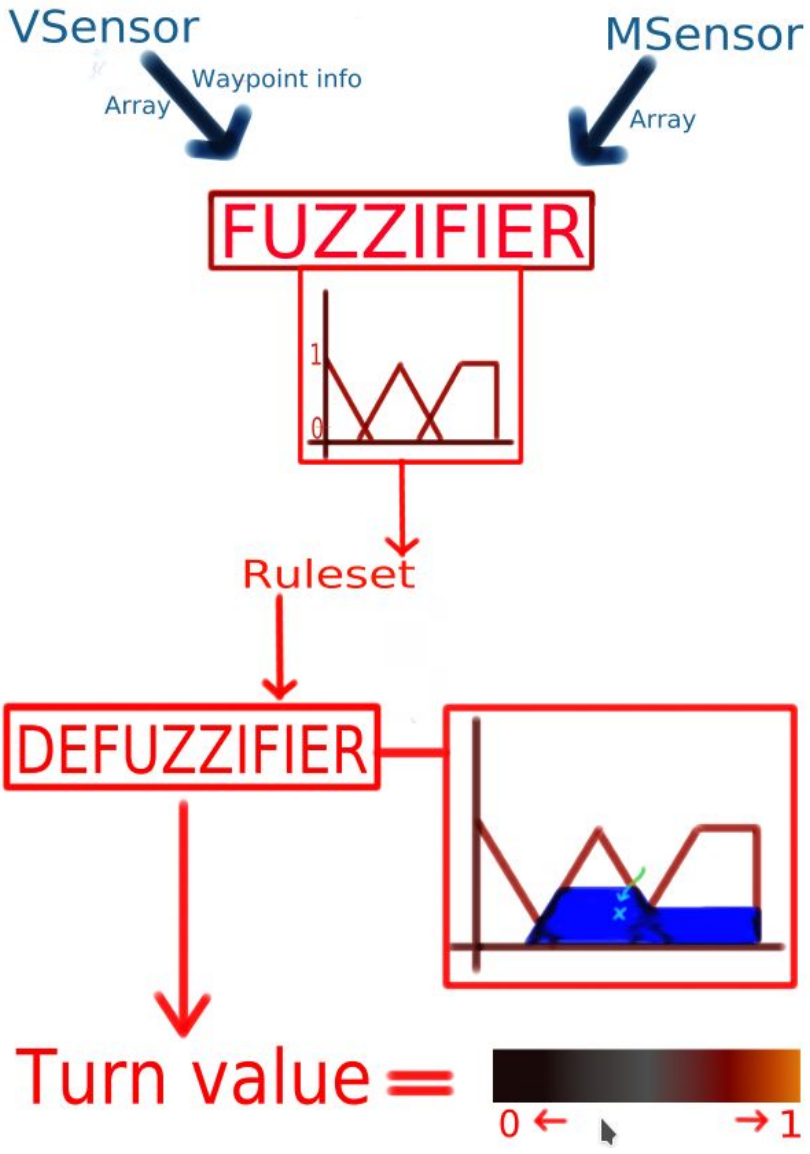
- **VSensor** o **Visual Sensor**: Recoge la información de todos los objetos del mapa que **tengan componente colisión.** Recibe del PhysicsManager la lista de objetos que tengan componentes de colisión y **devuelve** una ristra de **VObjects** con esos datos. Accediendo al GameObject se puede obtener la información del resto de componentes que tenga para discernir

- **MSensor o Map Sensor: Recoge** la **información del terreno** como el terreno **en el que se encuentra y** los **límites** del mismo, y **calcula** los puntos de **colisión** con el terreno usando **dos antenas**. Estas antenas son **dos vectores** separados por un **pequeño ángulo enfrente** del **jugador**, que le permiten discernir si ir a izquierda o derecha.

b. Lógica difusa

Para mover al NPC, le daremos **inputs al sistema de control** para que se dirija al punto escogido por el sistema de planificación, usando un sistema de lógica difusa.

El esquema general del sistema es el siguiente:



La lógica difusa de los NPCs se **utilizará** para las acciones de **giro** manteniendo una **aceleración constante.** El sistema decidirá en qué grados ejecutar estas acciones, devolviendo **una decisión para el input** con cierto **grado de pertenencia** hacia uno de ellos.

Necesitaremos como entradas del sistema las variables el array de **VObjects** de los objetos **colisionables** que devuelve el **VSensor,** el array de VObjects con la información de puntos colisionables del terreno devuelto por el **MSensor** y el **vector del próximo waypoint, con su A y su B** devueltos por la composición con los sensores de VSensor**.**

Antes de nada **trataremos las variables con el fuzzificador.** Se adaptan para **pasar por las reglas de inferencia,** que les dan valores **entre 0 y 1.** De ellas obtendremos reglas o condiciones lógicas como “Cerca”, “Medio”, “Lejos”, “Más a la derecha”, “Más a la izquierda”, “Centrado”, como ejemplo, y las usaremos en los **sets de reglas** para dar forma a cada una de las posibles salidas.

Las reglas funcionan como los condicionales normales, con los operadores AND y OR, salvo que en este caso cuando dos reglas debieran ser un AND, usaremos el valor mínimo entre ambas; cuando debieran ser un OR, usaremos el valor máximo entre ellas, y el NOT es el inverso sobre 1 (1 - valor).

A partir de las reglas se obtienen valores para girar a la **izquierda, derecha** o mantenerse en el **centro.** Luego para tomar la decisión final usaremos el **cálculo de Mandani,** por el cual **obtenemos** los **centroides** del conjunto de las decisiones aplicadas el proceso **inverso** al **fuzzificador,** es decir, **defuzzificaremos las variables.**

Una vez se determina el valor de giro final, se multiplica por el valor máximo de giro y se aplica.

c. WayPoints

Tendremos una clase **WaypointManager** que manejará todos los waypoints del mapa, los cuales servirán a los NPC para seguir una ruta óptima.

WaypointManager está compuesto por:

- **Vector listSubNodes:** Contiene todos los Waypoints del mapa.

De esta clase derivarán otras 2:

- **WaypointComponent:** Componente que controla el **radio** y **nivel** del waypoint.
  - **Float level**: Determina el **orden** en el que los NPC seguirán la **ruta**

- **PathPlanningComponent:** Componente propio de cada NPC que **calcula** el **próximo punto** al que debe ir y, mediante un getter, la lógica difusa obtendrá la posición a la que ir.
  - **Float seconds:** Nivel de **adelanto** qué llevará la IA en los **cálculos**. Con valor 1, tendría la misma precisión que un player.
  - **Float distLastWay: Distancia** al próximo Waypoint.
  - **Unsigned int lastVector: Última posición** guardada del **vector** de waypoints de **WaypointManager**. Con esta posición tendremos **acceso** al último **Waypoint** por el qué ha pasado el NPC.
  - **glm::vec3 nextPos: Próximo punto** en el mapa al que debe ir el **NPC**.

i. Evolución

El sistema de **WayPoints** de nuestro proyecto se desarrollará **en varias iteraciones**.

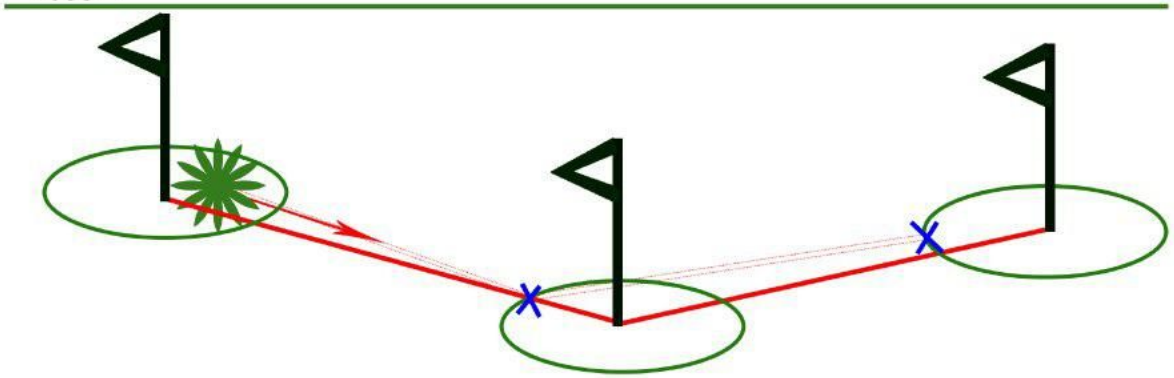
1. **Fase inicial**: **Incluir** un WayPoint en el mapa con p**untos ordenados y conectados** entre sí, y **con** un **rango** de detección. Probamos que el NPC va de un punto al siguiente**, usando lógica difusa** para acelerar y girar.

2. **Segunda fase**: **Esquivar objetos u otros jugadores.** El vector del WayPoint ahora apunta a un punto entre la línea interpolada de los próximos puntos/nodos del WayPoint. Apuntará **más lejos** en el WayPoint cuanto **más rápido** vaya. **Si colisiona, recalcula** el vector del WayPoint, y manda la señal al coche de frenar.

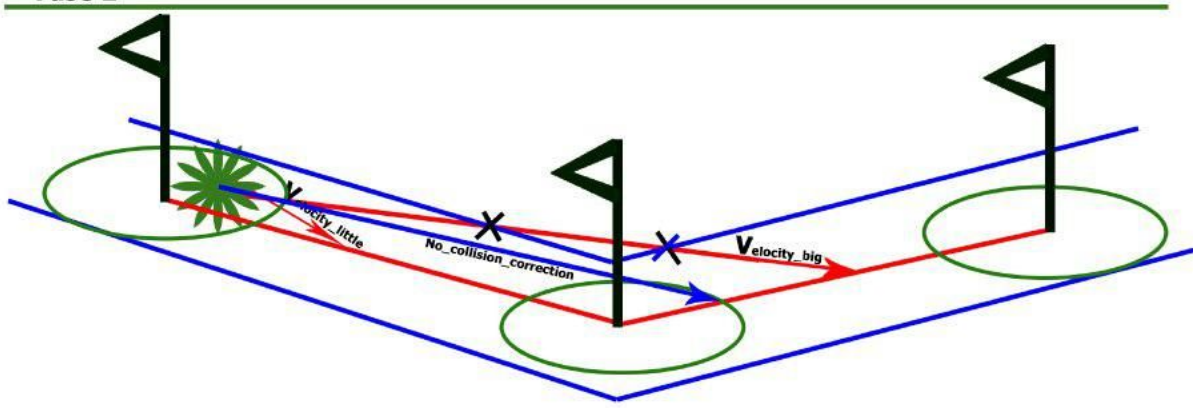
3. **Tercera fase: WayPoints paralelos** en el mismo trozo de ruta. El NPC puede elegir trazados. Se tendrá en cuenta **el derrape** para el giro en curvas **(interpolación circular del movimiento).** Ahora los **WayPoints** no tienen que ser líneas rectas, pueden ser **líneas curvas**.

   Esto se detalla en el esquema siguiente:

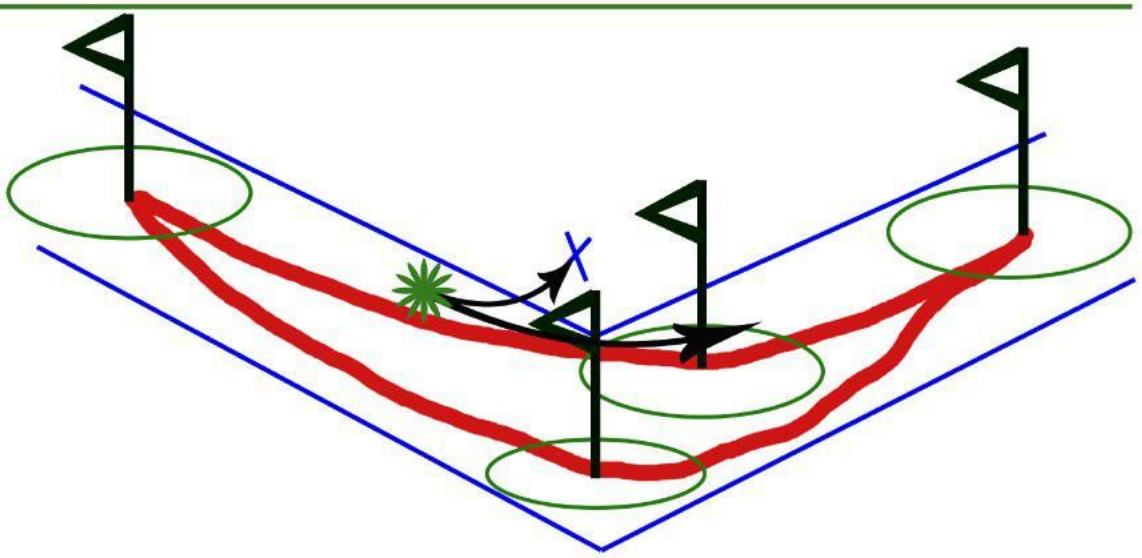
Fase 1



Fase 2



Fase 3



3. Sistema de batalla

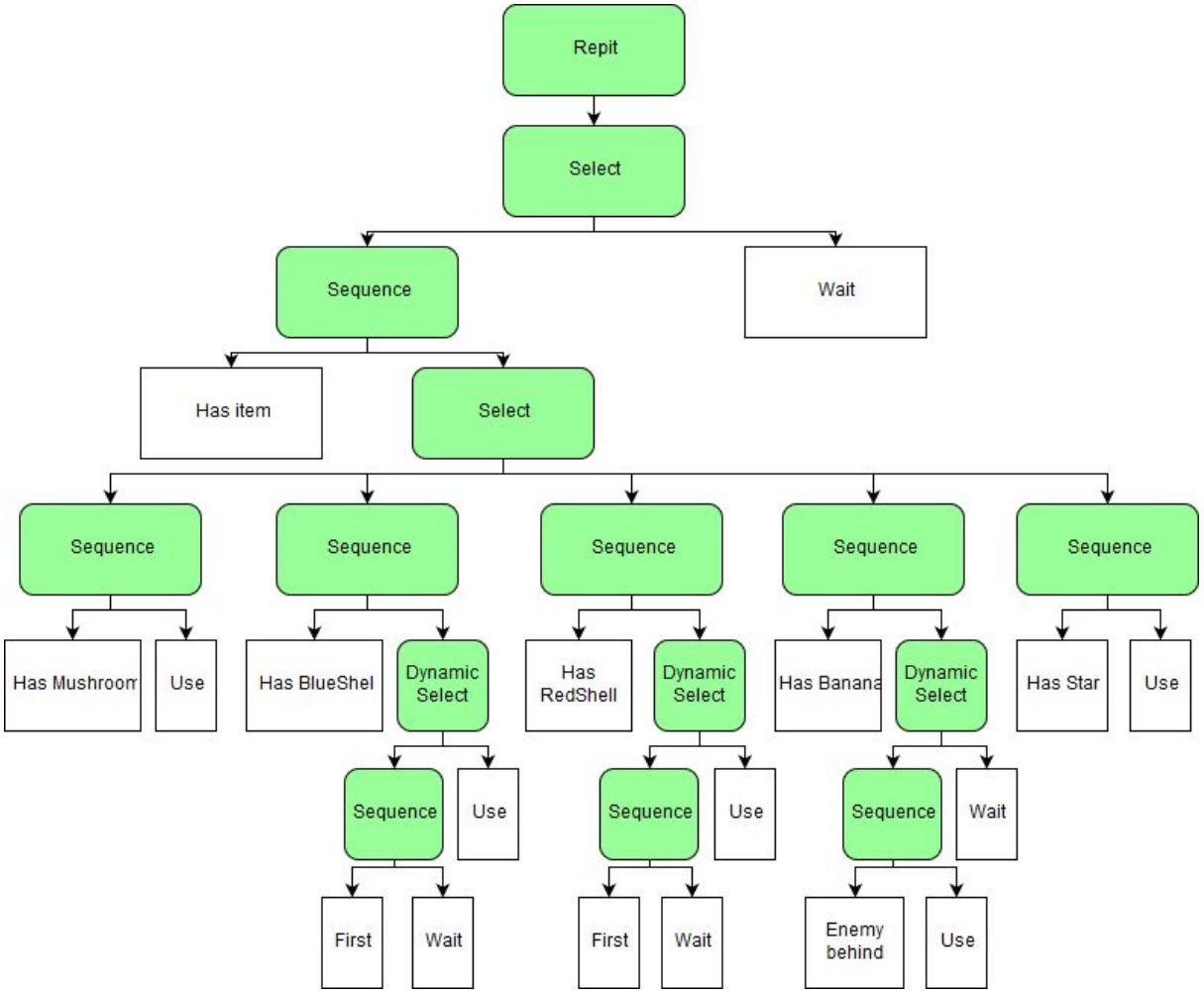
El sistema de batalla estará formado por un behaviour tree. Los posibles estados que puede devolver un nodo del Behaviour Tree son los siguientes:

- **Invalid:** Estado inicial de un nodo inicialmente, antes de haber sido recorrido.
- **Running:** Cuando una acción se encuentra en ejecución.
- **Succeeded:** Cuando una acción se completa cumpliendose tal y como se esperaba, o una condición es válida y se cumple.
- **Failed:** Cuando una acción falla o una condición no se cumple.
- **Aborted:** Cuando una acción en estado Running no se completa.

El sistema estará compuesto por:

- **Repeater:** Reinicia el behaviour tree cuando una de las acciones da como resultado SUCCEEDED o FAILED.
- **Selector:** Estará compuesto por ramas de forma que irá comprobando una a una, cuando alguna de SUCCEEDED o RUNNING el selector parará y lo devolverá como resultado. En el caso de que ninguno devuelva SUCCEEDED, dará como resultado FAILED.
- **Dynamic Selector:** Funciona de manera parecida al selector, solo que en caso de quedar en estado RUNNING volverá a evaluar en cada Tick.
- **Sequence:** Intentará ejecutar todos sus hijos en orden consecutivo, devolviendo SUCCEEDED si lo consigue, RUNNING si algún hijo queda en dicho estado, o FAILED si alguno de sus hijos falla.

El Behaviour Tree en su totalidad se encuentra definido en el diagrama a continuación:



En este diagrama vienen diferenciados, **por un lado**, los **nodos que representan acciones** que son los que pueden devolver un estado determinante para cada Tick, representados con **fondo blanco,** y por otro lado los **nodo**s que forman parte **de la estructura** del Behaviour Tree para su correcto funcionamiento, representados con **fondo verde.**

La estructura básica del Behaviour Tree actualmente es simple. En primer lugar, distinguimos entre si tenemos ítem o no, ya que el sistema únicamente puede actuar en caso afirmativo. En caso de tenerlo, se pasa al select principal de acción, donde en función de que item se tiene se accede al subárbol de acción de cada ítem.

Actualmente, se ha definido que para dos ítems concretos (Mushroom y Star) la acción más apropiada sea usarlos nada más tenerlos. Para el resto de casos, se define un Dynamic Selector que reevalue constantemente que acción debe hacer el enemigo en función de las condiciones de su entorno, teniendo en cuenta, por supuesto, que la decisión final del Behaviour será binaria, pues solo se puede decidir si usar el Item o no.

En el caso de los dos caparazones, ya que su uso es perjudicial en el caso de ir primero, se evita usarlo si dicha condición se cumple, mientras que en el caso del plátano, se usa únicamente cuando tienen algún enemigo detrás dentro de un rango determinado.