Inteligencia Artificial empleada en “Prisionero”

1. Algoritmo: Trigger System

En el juego “Prisionero”, he utilizado este algoritmo de propósito general para registrar y actualizar los distintos eventos que ocurren durante el “gameplay”. La forma de implementar el código ha sido muy similar al algoritmo del libro *AI Game Programming Wisdom*.

Los Triggers se crean en determinado momento del “Game Play”, el sistema los actualiza y los elimina cuando es necesario y lo “Agents” los pueden atender si están activos.

Los archivos del código que completan este algoritmo son: Trigger.h, TriggerSystem.h, TriggerSystem.cpp, Agent.h, Agent.cpp.

Estos son todos los tipos de Triggers que pueden ocurrir durante una partida:

```
kTrig_None          = 0,
kTrig_GiveObject    = (1 << 1),
kTrig_Gunfire       = (1 << 2),
kTrig_KnifeAction   = (1 << 3),
kTrig_TakeObject    = (1 << 4),
kTrig_ReleaseObject = (1 << 5),
kTrig_SmallNoise    = (1 << 6),
kTrig_MediumNoise   = (1 << 7),
kTrig_BigNoise      = (1 << 8),
kTrig_PlayerOnStage = (1 << 9),
kTrig_AgentOnStage  = (1 << 10),
kTrig_ObjectOnStage = (1 << 11),
kTrig_UseKey        = (1 << 12),
kTrig_UseAidKit     = (1 << 13),
```

Trigger.h

Diferencias respecto al libro *“AI Game Programming Wisdom”*

- La estructura Trigger incluye un puntero de tipo “AIEvent” (evento de reputación). Solo contendrá un evento si es el jugador quien dispara el Tigger y si el tipo del Trigger causa algún efecto de reputación. Por ejemplo Trigger de tipo kTrig_TakeObject no tiene vinculado ningún AIEvent, sin embargo kTrig_Gunfire sí.
- El constructor de la estructura Trigger no venía definido en el libro, es una aportación propia.
- La clase TriggerSystem implementa el patrón “Singleton”
- Se han implementado algunos métodos para devolver el primer Trigger del “mapa de Triggers” con determinadas coincidencias:

```
TriggerRecordStruct* GetTriggerBy(unsigned long nTriggerID);
TriggerRecordStruct* GetTriggerBy(rubengine::GameObject* source, EnumTriggerType type);
```

TriggerSystem.h

- Las signatura del método Update del Trigger System es diferente:

```
void CTriggerSystem::Update(const std::vector<CAgent*>& agents, b2World* world)
```

TriggerSystem.h

- A la hora de saber si un Agent atiende un Trigger, en el libro se contempla como última condición que el radio de acción de cada Trigger intersecte con dicho Agent. He utilizado esta técnica para los tirggers que no son visibles, como por ejemplo los Triggers de tipo ruido. Sin embargo no me parecía útil para Triggers "visuales", como por ejemplo "coger un objeto" que ésta fuera la condición última que determinara si el Agent atiende el Trigger. La forma en la que he resuelto esto es que para los Triggers visuales (radio == 0), el Agent en cuestión nos indique si ve o no un Trigger con el sensor de visión que posee:

```
// Is a visible trigger (visible -> radius == 0) ?
if (!pRec->fRadius){
    intersect = pAgent->seeTrigger(pRec, world);
}
// Is other kind of trigger ?
else{
    // Check intersection
    intersect = INTERSECT(pAgent, pRec);
}
```

TriggerSystem.cpp

- En el método Update del Trigger System, el libro refleja que existe una clase llamada CAgent:

```
// HandleTrigger returns true if the
// Agent responded to the trigger.
if( pAgent->HandleTrig(pRec) )
{
    // Listen to highest priority trig at any instant.
        break;
    }
}
```

De este código se entiende que ha de existir una clase que al menos sea capaz de decirnos si está atendiendo un Trigger concreto o no. La clase CAgent no viene definida ni declarada en el libro, solo se nombra. Por lo tanto la clase CAgent es también una aportación propia.

- La clase CAgent la he resuelto como una interfaz pensada para que los NPCs del juego "Prisionero" la implementen. Por lo tanto esta interfaz aúna elementos comunes a todos los NPCs:

```
AIMemory m_AImemory;
Group m_agentGroup;
unsigned long m_triggerFlags;
__int64 m_nextTriggerUpdate;
TriggerRecordStruct* m_handleTrigger;
AIengine::VisibilitySensor m_visibilitySensor;
```

Agent.h

Algoritmo: Dynamic Reputation System Based on Event Knowledge

En el juego “Prisionero”, he utilizado este algoritmo de propósito específico para registrar y manejar los distintos eventos de reputación durante el “gameplay”. La filosofía/fundamentos del diseño han sido muy similares al algoritmo del libro *AI Game Programming Wisdom*, sin embargo la implementación del mismo ha sido en su mayoría de elaboración propia.

Los Eventos de reputación (AIEvent) se crean y se vinculan en el momento del registro de un Trigger (pero no siempre que se crea un Trigger se crea un Evento de reputación). Dependiendo de los eventos que un Agent presencie y del tipo de Agent (Group) que sea, éste reaccionará modificando su reputación hacia el jugador y alterando su estado actual.

Los archivos del código que completan este algoritmo son: AIEvent.h, AIEvent.cpp, ReputationTemplate.h, ReputationTemplate.cpp, AIMemory.h, AIMemory.cpp.

AIEvent

Básicamente es la estructura que contiene los atributos y métodos relevantes de un Evento de reputación único, tales como el ID, el número de Agents que saben de él y los efectos:

```
class AIEvent
{
private:
    EventID*                    m_eventID;
    unsigned int                m_referenceCount;
    std::map<Group, Effect>     m_effects;
```

AIEvent.h

ReputationTemplate

Nos ayuda a “mapear” los distintos eventos de reputación que pueden ocurrir en el mapa. Esto es, lee de un fichero “Assets\Templates\reputation.xml” y mantiene en memoria los datos para que cuando se crea por primera vez un evento de reputación se busque dicho evento y se apliquen los efectos:

```
static void LoadTemplateCollection(const std::string& templatePath);
static std::map<Group, Effect> find(const Verb& verb, const Group& object);
```

ReputationTemplate.h

El archivo Assets\Templates\reputation.xml, tiene una estructura como la siguiente:

```
<template verb="doViolenceTo">

    <object type="policia">

        <effect obj="policia" rep="hate"></effect>
        <effect obj="alguacil" rep="hate"></effect>
        <effect obj="enfermera" rep="hate"></effect>
        <effect obj="limpieza" rep="hate"></effect>
        <effect obj="preso" rep="like"></effect>

    </object>
```

AIMemory

Estructura que sirve para mantener el control de los eventos que ocurren a lo largo de una partida, contar las referencias de los Agents a estos, asignarlos a los Agents, fusionar memorias, modificar la reputación respecto al jugador, etc.

```
std::vector<EventID*>       PerNPCLongTermMemory;
static std::vector<AIEvent*>    MasterEventList;
std::map<Group, int>        PerNPCReputationTable;

bool FLAG_reputation = false;
unsigned int m_eventsToShare;

void    Merge(AIMemory* pAIMemory, Group group);
void    Update(AIEvent* pAIEvent);
bool    AddNewMemoryElement(AIEvent* pAIEvent, Group group);
void    ApplyEffect(EventID* eventID, const Effect& effect);
```

AIMemory.h

Sistemas de percepción de los NPCs

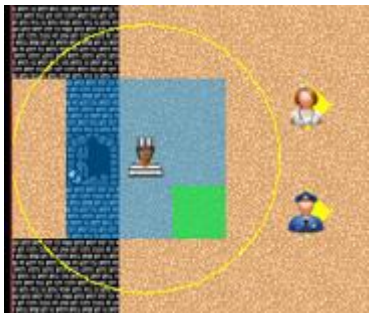
Los sistemas de percepción son una aportación propia.

Los NPCs tienen dos modos de percibir lo que pasa en su entorno. Solamente pueden atender Triggers que están sucediendo en momentos de la partida. Algunos Triggers son de tipo ruido y otros visuales.

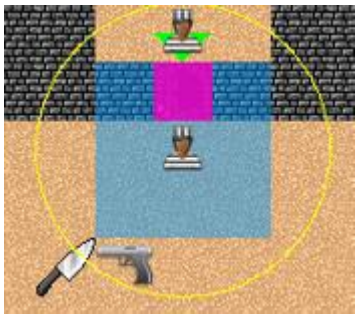
Por ejemplo abrir una celda crea un Trigger de tipo “kTrig_MediumNoise” y también otro “kTrig_UseKey”.

Oído

Un NPC atiende un Trigger de tipo ruido si intersecta con su radio:



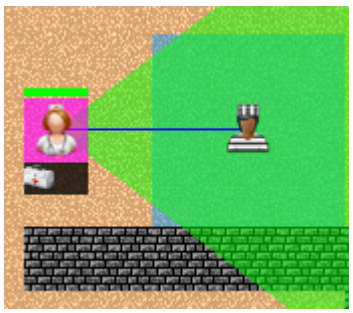
Ninguno de los NPCs nos escucha abrir la puerta



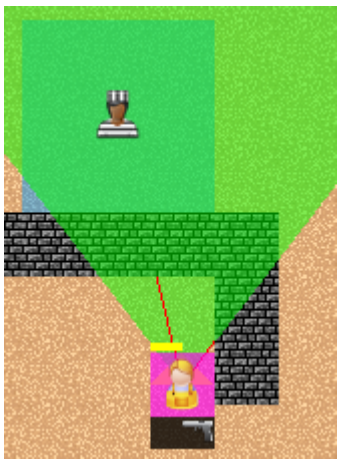
El NPC preso nos escucha abrir la puerta

Vista

Un NPC ve un Trigger si está dentro de su campo de visión y si supera el test de “Ray Cast” (si el rayo que va del centro del Trigger al centro del NPC no intersecta con ningún obstaculo).



El NPC enfermera nos ve



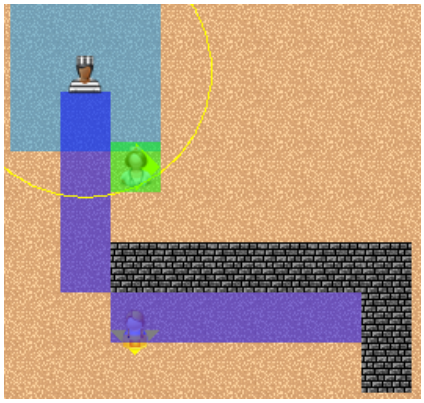
El NPC responsable de limpieza no nos ve

Movimientos de los NPCs

Los NPCs tienen dos estados por defecto posibles para el movimiento. Estos estados son estáticos o siguiendo una ruta preestablecida, esto es una aportación propia.

Tanto los NPCs (cuando cambian su estado) como el jugador cunado tienen como target una posición diferente a la que tienen por defecto, calculan esta nueva ruta a la posición con un algoritmo de Pathfinding A*.

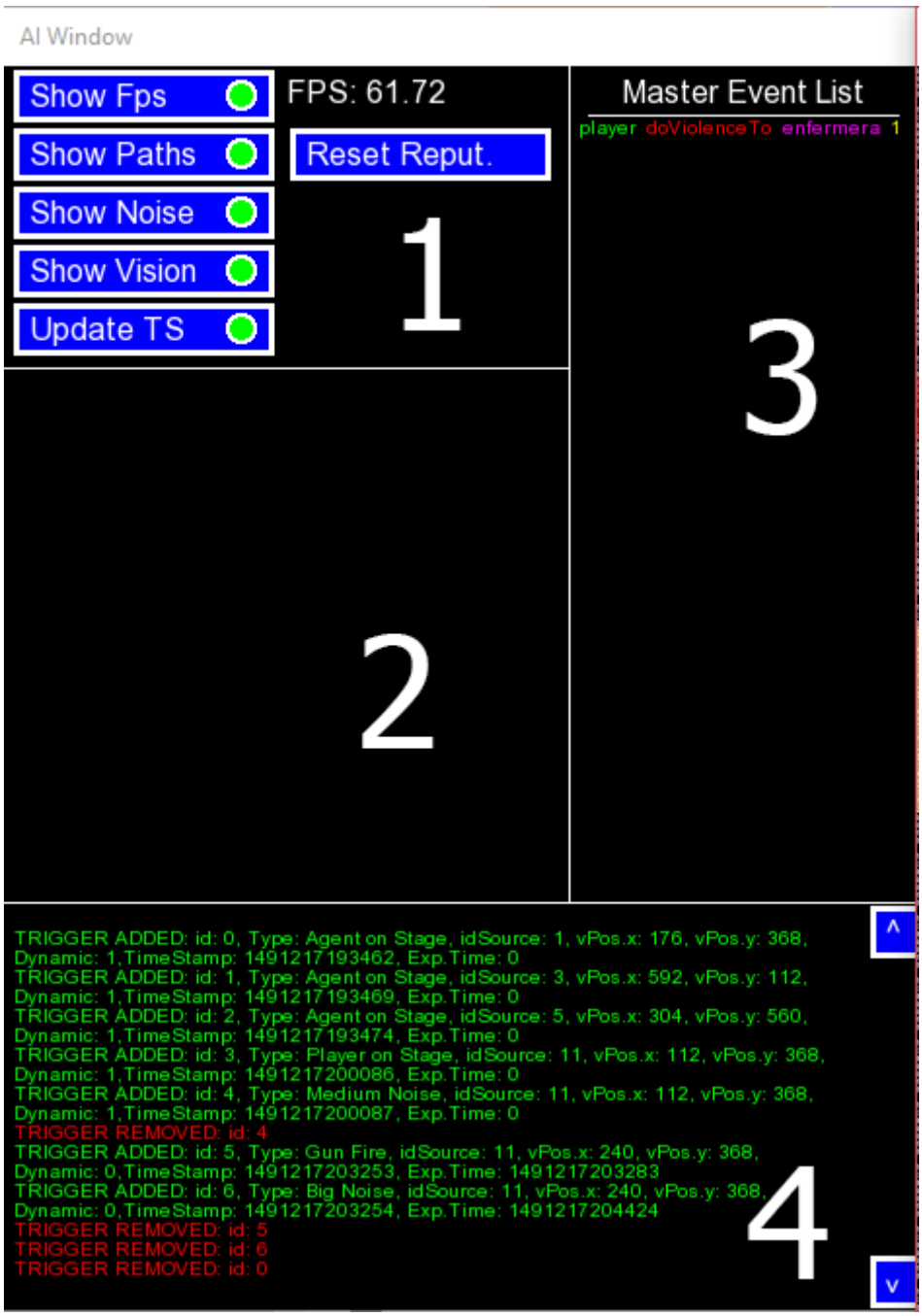
Este algoritmo lo he obtenido del siguiente tutorial (parte 1/4):
https://www.youtube.com/watch?v=NJOf_MYGrYs



He realizado algún retoque mínimo para adaptarlo a las necesidades del juego.

Herramientas de diagnóstico

Las herramientas de diagnóstico son una elaboración propia y las he implementado en forma de ventana de SFML. La he estructurado en 4 zonas bien diferenciadas.



- La zona 1: Es la UI con la que se puede manipular algunos aspectos visuales del juego, resetear los eventos de reputación, elegir actualizar los triggers o no, entre otros…
- La zona 2: Es utilizada para visualizar la información del personaje seleccionado.
- La zona 3: Es utilizada para monitorizar la lista de eventos de reputación ocurridos en la partida (Master Event List) o la lista de eventos presenciados por el personaje seleccionado.
- La zona 4: Es utilizada para monitorizar en orden cronológico la información de los Triggers de la partida, tanto cuando se registran como cuando se eliminan.

Diagrama de clases con abstracción para la IA