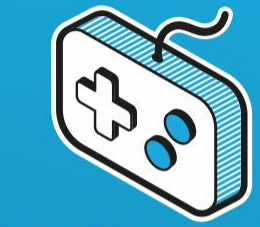


LEVELUP

GAME DEV HUB

PROGRAMACIÓN AVANZADA CON UNREAL ENGINE

Bloque 3.1 - Inteligencia Artificial

Inteligencia Artificial en UE

Inteligencia Artificial

Cuando hablamos de las clases de juego, normalmente nos referimos a GameMode, PlayerController, etc..

En Unreal Engine, la IA se define siguiendo un patrón muy similar al que seguimos con los personajes jugadores

- En el caso del jugador, hablamos del PlayerController
- En el caso de los NPC, hacemos uso también de un tipo de controlador llamado AIController

Inteligencia Artificial

- El AIController es generalmente el encargado de centralizar toda la lógica de gestión del personaje
  - Define qué sistema de control vamos a usar (BehaviorTrees, etc...)
  - Hace de punto central para transmitir información entre agente y sistema
  - Puede ser la referencia empleada para el sistema de percepción

Podemos entenderlo como un elemento controlado por el propio juego

IMPORTANTE:

El AIController es un ACTOR

Inteligencia Artificial

Para trabajar con AI en C++, debemos incluir el AIModule

```
PrivateDependencyModuleNames .AddRange(new string[]
{
  "AIModule"
});
```

Inteligencia Artificial

La clase de la que partimos para crear un controlador es el AAIController

```
class AAIControllerBase : public AAIController
```

Es conveniente que tengamos presente que AAIController es hijo de AController, por lo que comparte bastante con el propio APlayerController

Inteligencia Artificial

- Los dos métodos quizás más relevantes de un AAIController son las relativas a la toma de control del personaje

```
virtual void OnPossess(APawn* InPawn) override;
virtual void OnUnPossess() override;
```

- Es común que toda la configuración de inicio del personaje se gestione en el momento en el que el controlador tome **posesión** del agente.

Nota: Generalmente a las entidades controladas por Inteligencia Artificial se las suele denominar Agentes

Inteligencia Artificial

Al tratarse de funciones virtuales, las podemos sobreescribir.

Mucho cuidado de no olvidar la llamada al padre

```
void AAIControllerBase::OnPossess(APawn* InPawn)
{
  Super::OnPossess(InPawn);
}
```

```
void AAIController::OnPossess(APawn* InPawn)
{
    // don't even try possessing pending-kill pawns
    if (InPawn != nullptr && !IsValid(InPawn))
    {
        return;
    }

    Super::OnPossess(InPawn);

    if (GetPawn() == nullptr || InPawn == nullptr)
    {
        return;
    }

    // not calling UpdateNavigationComponents() anymore
    // is now observing newly possessed pawns (via OnNe

    if (PathFollowingComponent)
    {
        PathFollowingComponent->Initialize();
    }

    if (bWantsPlayerState)
    {
        ChangeState(NAME_Playing);
    }
```

Inteligencia Artificial

A diferencia del PlayerController, que definimos la clase desde el GameMode, hay que definir el AIController desde la clase de cada agente.

Generalmente en el constructor del Pawn:

- La clase de controlador que vamos a Spawnear

```
AIControllerClass = AAIControllerBase::StaticClass();
```

- En qué caso vamos a crearla

```
AutoPossessAI = EAutoPossessAI::PlacedInWorldOrSpawned;     -> siempre
AutoPossessAI = EAutoPossessAI::Spawned;                    -> solo en spawn
AutoPossessAI = EAutoPossessAI::PlacedInWorld;              -> solo si ya existe en el nivel
AutoPossessAI = EAutoPossessAI::Disabled;                   -> nunca
```

Inteligencia Artificial

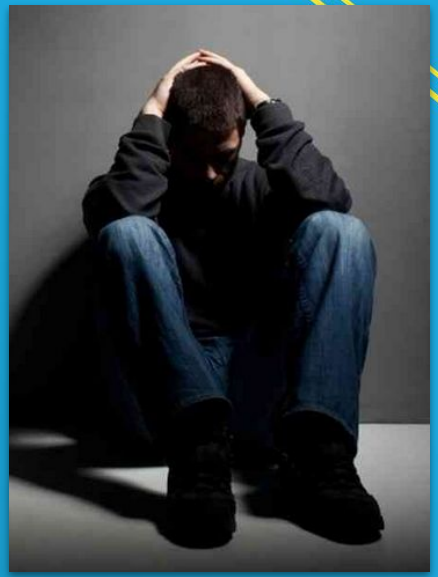
IMPORTANTE: Una de las frases más usadas cuando trabajamos con AI es "¿¡Por qué no se mueve!?"

Uno de los principales motivos es porque el AIController:

- No se ha creado
- No ha poseido al agente

Es tentador que creemos un Character a partir de nuestro personaje. Debemos asegurarnos siempre de configurar

- `AIControllerClass`
- `AutoPossessAI`



No quieres acabar como este pobre programador de IA, ¿no?

Percepción

Inteligencia Artificial

Perception component

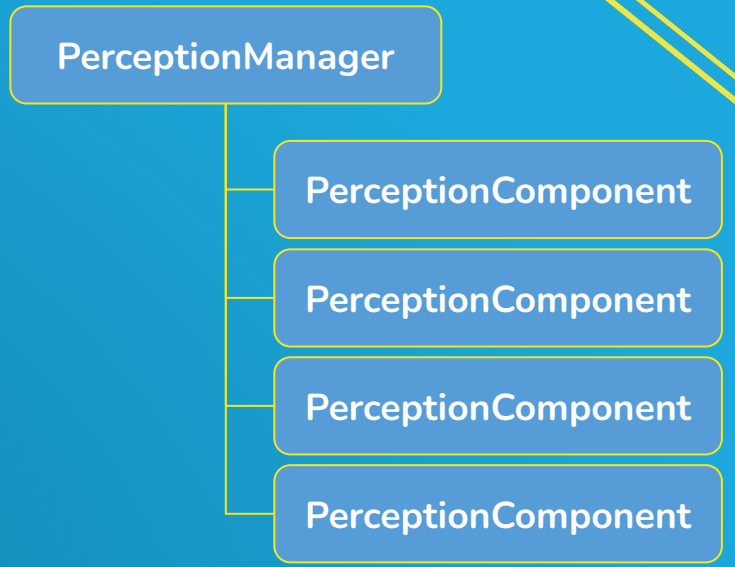
- Cuando hablamos de IA, hablamos de comportamientos y tareas
- Estas ejecuciones se realizan en base a criterios definidos
- La percepción es una de las pistas que nos permite definir cómo actúan los personajes

NOTA: El sistema de Percepción que define Unreal Engine no es estrictamente necesario. Es posible que creemos nuestro propio sistema sin recurrir a este.

Inteligencia Artificial

El sistema de percepción se basa en el uso de componentes que implementamos sobre el AIController

- Un componente por entidad que gestiona la recepción de estímulos. Eso dispara diferentes eventos a los que podemos engancharnos.
- Diferentes sentidos que representamos por medio de objetos
- Estímulos asociados a estos sentidos
- Un perception manager que se encarga de coordinar todo



Inteligencia Artificial

Nota: Realmente no es necesario bajo ningún concepto usar el PerceptionSystem de Unreal. Puede ser interesante que implementemos un sistema más ajustado a nuestras necesidades.

Ejercicio opcional: Implementar un sistema basado en SphereTraces que permita identificar los personajes que hemos detectado.

Inteligencia Artificial

En el PerceptionSystem hay que tener en cuenta solo dos elementos

---

Emisores de estímulos

- Son las fuentes que van a emitir eventos hacia los receptores en base a ciertos sentidos

Receptores de estímulos

- Son los actores en los que vamos a realizar queries sobre los otros agentes o personajes

---

Pueden estar combinados

Inteligencia Artificial

Perception Component: Generalmente se ubica en el AIController.

```
PerceptionComponent = CreateDefaultSubobject<UAIPerceptionComponent>(TEXT("PerceptionComponent"));
```

Nota: PerceptionComponent ya está declarado en el AController, pero no crea automáticamente un Componente

Discusión: ¿Cómo podríamos hacer para que se crease un Perception component de una cierta clase automáticamente? ¿Podríamos parametrizarlo en el AIController que hemos creado?

Inteligencia Artificial

Definición de sentidos

Como todo lo que queremos exponer al Editor, los sentidos se definen como UObjects.

- UAISense_ que es la definición del sentido
- UAISenseConfig_ que es la implementación que le pasamos al PerceptionComponent para personalizar la percepción de este.

Para indicarle al PerceptionComponent un sentido del que estar atento:

```
UAISenseConfig_Sight* SightConfig =
                        CreateDefaultSubobject<UAISenseConfig_Sight>(TEXT("UAISenseConfig_Sight"));
```

Inteligencia Artificial

Definición de sentidos

Cada sentido tiene sus propiedades. Podemos personalizarlas. Esto normalmente ocurre en el constructor del AIController para cada sentido.

Ejemplo:

```
SightConfig->SightRadius = 1500;
SightConfig->LoseSightRadius = 2000;
SightConfig->PeripheralVisionAngleDegrees = 70.f;
SightConfig->AutoSuccessRangeFromLastSeenLocation = FAISystem::InvalidRange;
SightConfig->SetMaxAge(0.f);
```

-> esto indica cuanto tarda el enemigo en “olvidarse” de nosotros cuando nos alejamos

```
SightConfig->DetectionByAffiliation.bDetectEnemies = true;
```

-> tipo de entidades (amigos, enemigos)

```
SightConfig->DetectionByAffiliation.bDetectFriendlies = true;
SightConfig->DetectionByAffiliation.bDetectNeutrals = true;
```

Inteligencia Artificial

Definición de sentidos

- Con los valores configurados, enviamos la información al sistema:

```
PerceptionComponent->ConfigureSense(*SightConfig);
```

Esto se encargará de registrar para este componente en el sistema un sentido. Cuando se cumplan las condiciones (haya un personaje delante del controller) se dispararán los eventos correspondientes.

Inteligencia Artificial

El PerceptionComponent implementa dos Delegates que se disparan cuando un nuevo agente o actor es percibido (es un emisor de estímulos)

- **En el AIController podemos definir dos métodos:**

```
UFUNCTION() void PerceptionUpdatedForMultipleActors (const TArray<AActor*>& Actors);
UFUNCTION() void PerceptionUpdated (AActor* Actor, FAIStimulus Stimulus);
```

- **Registramos estos métodos a los dos delegates**

```
PerceptionComponent->OnTargetPerceptionUpdated.AddDynamic(
                                this, &ThisClass::PerceptionUpdated);

PerceptionComponent->OnPerceptionUpdated.AddDynamic(
                                this, &ThisClass::PerceptionUpdatedForMultipleActors );
```

Inteligencia Artificial

Es rescomendable que se registren los métodos en el OnPossess del AIController

```
void AAIControllerBase::OnPossess(APawn* InPawn)
{
  Super::OnPossess(InPawn);

  PerceptionComponent->OnTargetPerceptionUpdated.AddDynamic(
                          this, &ThisClass::PerceptionUpdated);

  PerceptionComponent->OnPerceptionUpdated.AddDynamic(
                          this, &ThisClass::PerceptionUpdatedForMultipleActors );
}
```

IMPORTANTE: Aunque no es necesario, como alternativa podríamos añadir al delegate de la percepción métodos del PossessedPawn. Sin embargo, tendremos que asegurarnos de “liberar esos métodos” en el OnUnPossess

Inteligencia Artificial

StimuliComponent

- Es un componente que permite definir a un Actor como una fuente detectable.
- Podemos indicar qué sentidos lo perciben
- Todos los Characters están automáticamente dados de alta

Inteligencia Artificial

En un Actor cualquiera:

```
UPROPERTY(EditAnywhere, meta=(AllowPrivateAccess = true))
UAIPerceptionStimuliSourceComponent * StimuliSourceComponent;
```

- Configuramos qué sentidos queremos que se puedan detectar

```
StimuliSourceComponent =
    CreateDefaultSubobject<UAIPerceptionStimuliSourceComponent>(TEXT("StimuliComponent"))
);
```

- Registramos los sentidos (todos los que queramos)

```
StimuliSourceComponent->RegisterForSense(UAISense_Sight::StaticClass());
```

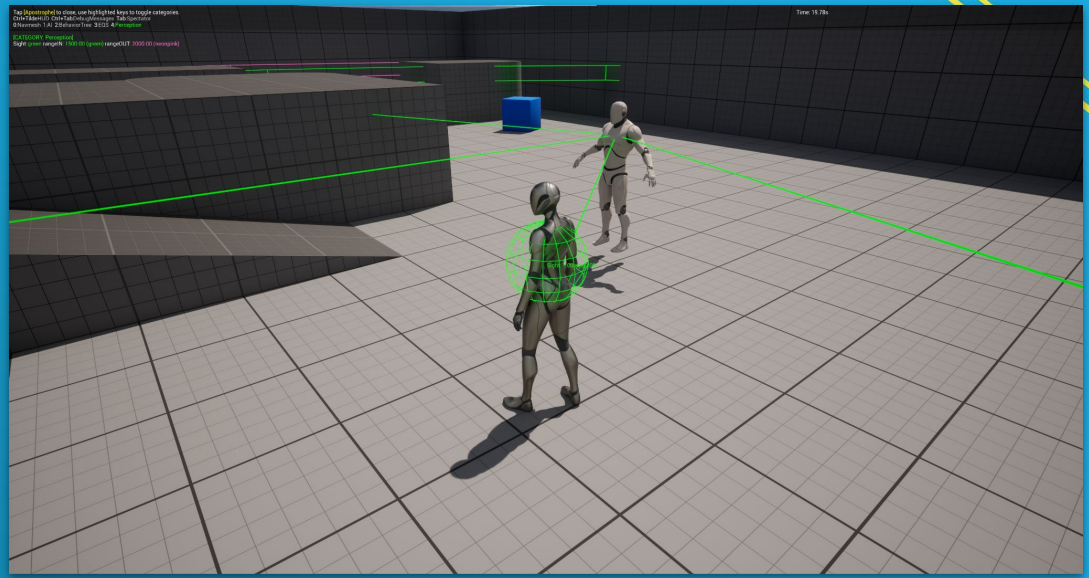
- Si no se registra por defecto, lo tenemos que hacer manualmente (lo que no sea ACharacter)

```
StimuliSourceComponent->RegisterWithPerceptionSystem();
```

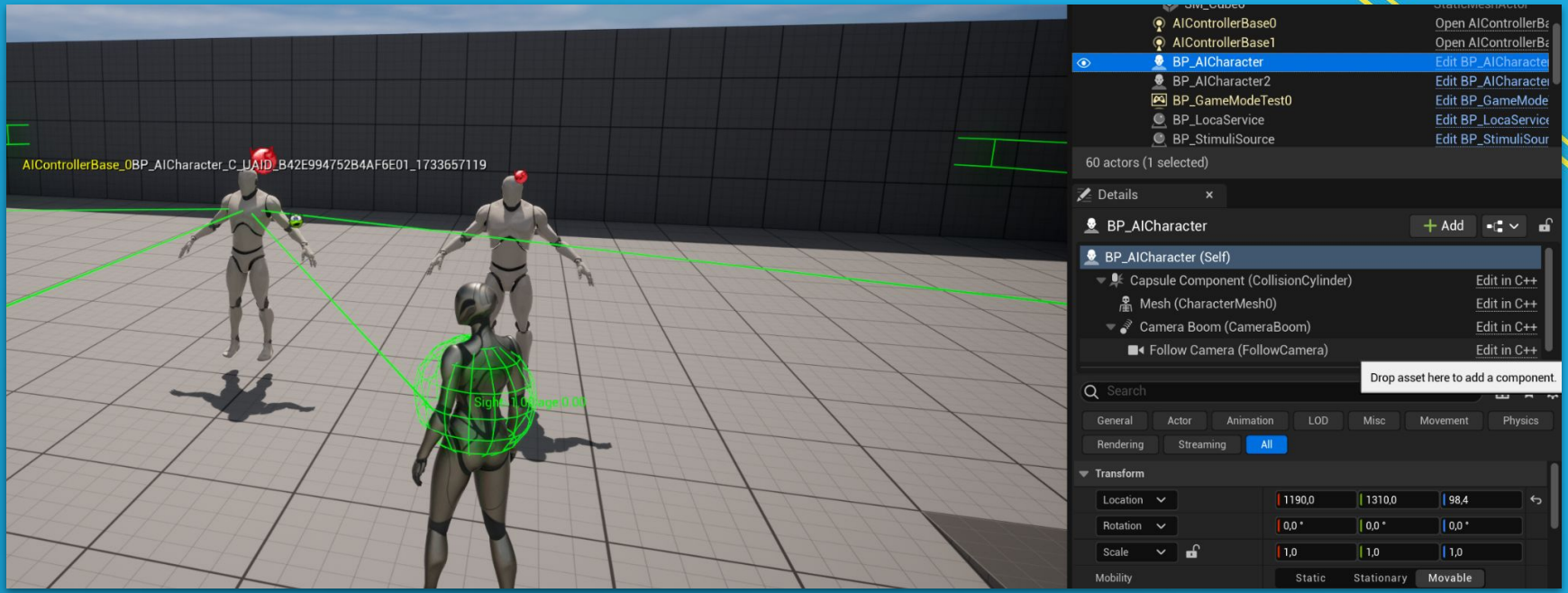
Inteligencia Artificial

Una vez hecha la configuración de los sentidos, podemos ver el resultado por medio del menú de Debug

- Pulsar ‘ para desplegarlo
- Solo es válido cuando estamos en el juego
- Para escapar de la perspectiva del personaje, Tab



Inteligencia Artificial



- Es posible alternar entre varios AICharacters simplemente seleccionándolos en el WorldOutliner

Inteligencia Artificial

- **Equipos**

Es posible definir a partir del PerceptionComponent cómo interactúan los agentes dependiendo del equipo al que pertenecen.

Esto se define definiendo las interacciones entre cada una de las facciones

Inteligencia Artificial

El AIController implementa la interfaz IGenericTeamAgentInterface

- Internamente tiene un ID que podemos asociarle y que está vinculado a un equipo

```
FGenericTeamId TeamID;
```

- Podemos usar estos métodos para acceder a esta información

```
virtual void SetGenericTeamId(const FGenericTeamId& NewTeamID) override;
virtual FGenericTeamId GetGenericTeamId() const override { return TeamID; }
```

Es posible inicializar este valor caso a caso. Por ejemplo, podríamos asignarlo a partir del tipo del Pawn que controla el AIController

Nota: Estos métodos ya están definidos por lo que no hace falta sobreescribirlos si queremos operar con TeamID

Inteligencia Artificial

Discusión

Comúnmente veremos en documentación cosas como:

```
TeamID = FGenericTeamId(1);
```

¿Es esto buena idea? ¿Qué podemos hacer alternativamente para no ir dejando literales por ahí sin ton ni son?

Inteligencia Artificial

Es muy común definir un Enum para este fin y asociarle a cada entrada un valor.

Nota: Podemos asignar este valor donde más convenga. Un lugar en el que tiene sentido hacer esto es el el OnPossess, extrayendo el equipo actual del Pawn al que controlamos.

```
UENUM()
enum class ETeamsType : uint8
{
  None,
  Player = 1,
  RedFaction = 2,
  BlueFaction = 3,
};
```

```
FGenericTeamId Id =
            static_cast<uint8>(ETeamsType::BlueFaction);

SetGenericTeamId(Id);
```

Inteligencia Artificial

El paso más importante es establecer cómo se relacionan los equipos

- **El AIController implementa este método**

```
virtual ETeamAttitude::Type GetTeamAttitudeTowards(const AActor& Other) const override;
```

Permite devolver una actitud: Enemigo, Amigo o Neutral. Esto es lo que permite definir cómo interactúa alguien con esa relación con respecto a los sentidos del Controlador

```
SightConfig->DetectionByAffiliation.bDetectEnemies = false;
SightConfig->DetectionByAffiliation.bDetectFriendlies = true;
SightConfig->DetectionByAffiliation.bDetectNeutrals = false;
```

Inteligencia Artificial

Cuando se perciben los actores, el sistema evalúa por medio de `GetTeamAttitudeTowards` si es un el tipo de interacción que tenemos.

Se devuelven solo aquellos actores que hayamos definido en el sentido

```
DetectionByAffiliation.bDetectEnemies = false;
DetectionByAffiliation.bDetectFriendlies = true;
DetectionByAffiliation.bDetectNeutrals = false;
```

Inteligencia Artificial

Ejemplo:

```
ETeamAttitude::Type AAIControllerBase::GetTeamAttitudeTowards(const AActor& Other) const
{
  if(const APawn* OtherPawn = Cast<APawn>(&Other))
  {
     if(const AAIController* OtherController = Cast<AAIController>(OtherPawn->GetController()))
     {
        if(GetGenericTeamId() == OtherController->GetGenericTeamId())
        {
           return ETeamAttitude::Friendly;
        }
     }
  }

  return ETeamAttitude::Hostile;
}
```

Discusión

¿Es buena idea que definamos las relaciones por medio de esta función? ¿Qué ocurre si tenemos dos AIController con implementaciones diferentes?

Inteligencia Artificial

Si queremos implementar algo así para un sistema con varias facciones, lo más común es diseñar una matriz de correlaciones entre estas.

Un ejemplo puede ser la imagen de la derecha donde se relacionan todas las facciones de *Warhammer*

For the Emperor!!

Referencia: [Inicio | Games Workshop Webstore]

Navegación

Inteligencia Artificial

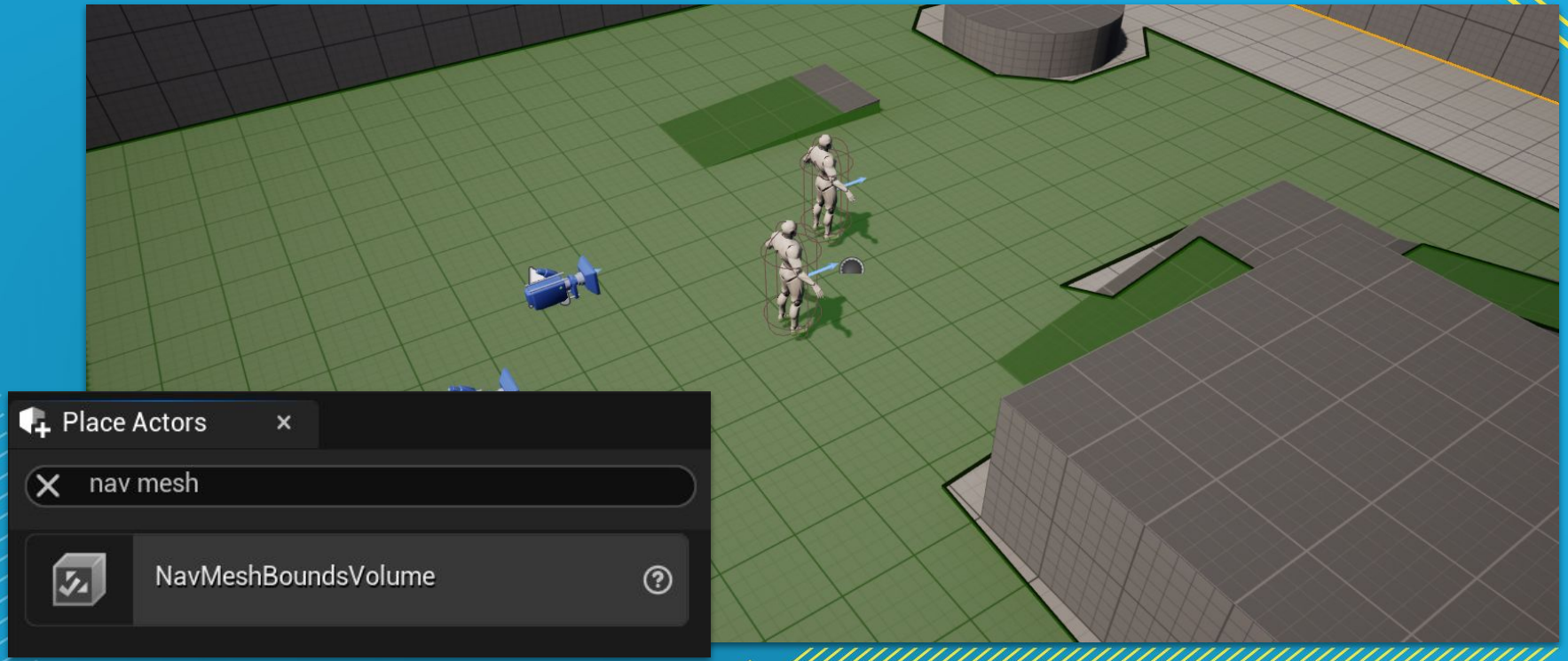
- Generalmente la navegación se define por medio de una rejilla de puntos
- En Unreal Engine, el encargado de gestionar la navegación de una IA a través de esta rejilla es el UPathFollowingComponent

Salvo casos excepcionales (movimiento tridimensional, custom, etc...), no necesitaremos hacer uso directo del este componente

Referencia : Navigation System

Inteligencia Artificial

Esta rejilla se genera por medio del NavMeshBoundsVolume



Inteligencia Artificial

Como ya es costumbre, tenemos que añadir el módulo que queramos usar (NavigationSystem)

```
PrivateDependencyModuleNames .AddRange(new string[]
{
  "Navigation"
});
```

Inteligencia Artificial

Al colocar en el nivel un NavVolume, automáticamente se genera un RecastNavMesh-Default.

Esto es un actor que va a contener la información de malla sobre la que podremos definir el PathFinding.

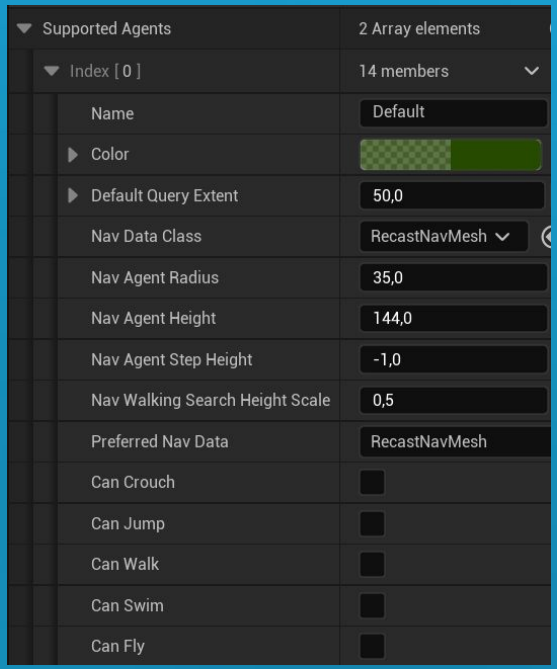
Cada vez que cambiemos la posición o el área abarcada por el NavMeshBoundVolume, estos se regenerarán

Discusión

¿Deberíamos usar la misma rejilla de navegación independientemente del tipo de Agente?

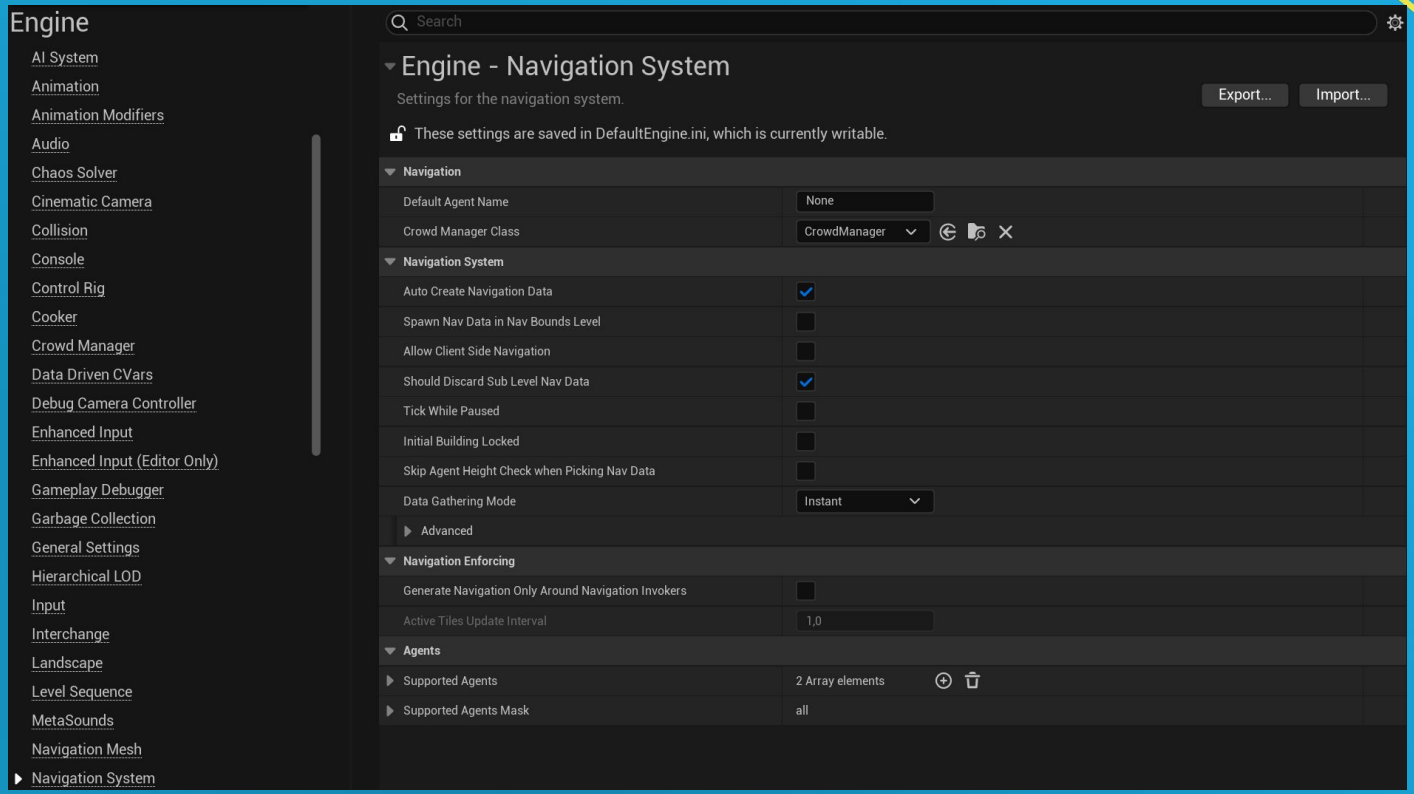
Inteligencia Artificial

Es posible generar diferentes rejillas dependiendo del tipo de agente que tengamos



Referencia : Navigation System

Inteligencia Artificial



Inteligencia Artificial

Esto es solo una introducción muy esencial al tema de navegación. En la documentación se trata en mayor detalle cuestiones como los filtros, múltiples agentes y otros temas similares.

Referencia : Navigation System in Unreal Engine

Inteligencia Artificial

Desde el AIController tenemos un número considerable de utilidades para gestionar el movimiento del agente

```
UFUNCTION(BlueprintCallable, Category = "AI|Navigation", Meta = (AdvancedDisplay = "bStopOnOverlap,bCanStrafe,bAllowPartialPath"))
EPathFollowingRequestResult::Type MoveToActor(AActor* Goal, float AcceptanceRadius = -1, bool bStopOnOverlap = true,  0 blueprint usages
    bool bUsePathfinding = true, bool bCanStrafe = true,
    TSubclassOf<UNavigationQueryFilter> FilterClass = NULL, bool bAllowPartialPath = true);

/** Makes AI go toward specified Dest location, aborts any active path following
 *  @param AcceptanceRadius - finish move if pawn gets close enough
 *  @param bStopOnOverlap - add pawn's radius to AcceptanceRadius
 *  @param bUsePathfinding - use navigation data to calculate path (otherwise it will go in straight line)
 *  @param bProjectDestinationToNavigation - project location on navigation data before using it
 *  @param bCanStrafe - set focus related flag: bAllowStrafe
 *  @param FilterClass - navigation filter for pathfinding adjustments. If none specified DefaultNavigationFilterClass will be used
 *  @param bAllowPartialPath - use incomplete path when goal can't be reached
 *  @note AcceptanceRadius has default value or -1 due to Header Parser not being able to recognize UPathFollowingComponent::DefaultAcceptanceRadius
 */
UFUNCTION(BlueprintCallable, Category = "AI|Navigation", Meta = (AdvancedDisplay = "bStopOnOverlap,bCanStrafe,bAllowPartialPath"))
EPathFollowingRequestResult::Type MoveToLocation(const FVector& Dest, float AcceptanceRadius = -1, bool bStopOnOverlap = true,  0 blueprint usages
    bool bUsePathfinding = true, bool bProjectDestinationToNavigation = false, bool bCanStrafe = true,
    TSubclassOf<UNavigationQueryFilter> FilterClass = NULL, bool bAllowPartialPath = true);

/** Makes AI go toward specified destination
 *  @param MoveRequest - details about move
 *  @param OutPath - optional output param, filled in with assigned path
 *  @return struct holding MoveId and enum code
 */
virtual FPathFollowingRequestResult MoveTo(const FAIMoveRequest& MoveRequest, FNavPathSharedPtr* OutPath = nullptr);

/** Passes move request and path object to path following */
virtual FAIRequestID RequestMove(const FAIMoveRequest& MoveRequest, FNavPathSharedPtr Path);
```

Muchos de estos métodos requieren computar el camino a partir del NavMesh (depende del agente).

- **La versión más sencilla es MoveTo**

Referencia : Navigation System

Inteligencia Artificial

Ejecutar un movimiento (desde el AIController)

- Creamos una orden de movimiento

```
FAIMoveRequest MoveRequest(TargetToMove); ó FAIMoveRequest MoveRequest(LocationToMove);
```

- Definimos las condiciones del movimiento

```
MoveRequest
  .SetAcceptanceRadius(50.f)
  .SetUsePathfinding(true)
  .SetAllowPartialPath(false)
  .SetNavigationFilter(DefaultNavigationFilterClass)
```

a qué distancia es dado por bueno

si queremos usar pathfinding o una línea recta

si no existe un camino completo, no nos desplazamos

filtro de navegación (coste del suelo, qué actores...)

- Emitimos la orden

```
MoveTo(MoveRequest);
```

Inteligencia Artificial

- Cuando emitimos una orden de movimiento, esta nos avisa cuando ha finalizado (cualquiera sea la razón)

```
void OnMoveCompleted(FAIRequestID RequestID, EPathFollowingResult::Type Result)
```

- Es un método del AIController. Normalmente es aquí donde gestionamos qué ocurre al finalizar

```
switch(Result)
{
case EPathFollowingResult::Success: break;
case EPathFollowingResult::Blocked: break;
case EPathFollowingResult::OffPath: break;
case EPathFollowingResult::Aborted: break;
case EPathFollowingResult::Invalid: break;
default: ;
}
```

Inteligencia Artificial

- Alternativamente podemos usar un Delegate para esto

```
UPROPERTY(BlueprintAssignable, meta = (DisplayName = "MoveCompleted"))
FAIMoveCompletedSignature  ReceiveMoveCompleted;
```

Ejemplo(AIBlueprintHelperLibrary.cpp:51 **):**

```
AIController->ReceiveMoveCompleted.AddDynamic(this, &UAIAsyncTaskBlueprintProxy::OnMoveCompleted);
```

Comportamientos

Inteligencia Artificial

Existen muchas maneras de implementar comportamientos para la gestión de Inteligencia Artificial en videojuegos.

Unreal Engine usa de forma nativa el concepto de Behavior Trees

- Árboles de comportamiento que se van ejecutando en base a ciertas condiciones
- Se apoyan en colecciones de información que iremos actualizando desde el AIController
- Controlamos el flujo en base a ciertas condiciones (decoradores)

Esto no quiere decir que estemos limitados a usarlos, pero son un buen punto de partida.

Inteligencia Artificial

Disclaimer: Esto es una opinión, pero puede ser conveniente seguirla

- La parte referente a los comportamientos suele estar más cerca de diseño
- Tiene que permitir un cierto nivel de flexibilidad
- Gestionar esto directamente desde C++ puede complicar mucho el sistema
- Normalmente, si trabajamos con Behavior Tree, se suele definir la estructura de estos desde el propio editor.
- Esta estructura se compone de tareas, decoradores y otras operaciones que sí podemos crear de forma individual desde C++

Hay un equilibrio entre ambas herramientas

Nota: Nuevamente, esto puede ser o no aplicable a otros sistemas como máquinas de estado, GOAP, etc...

Inteligencia Artificial

Hay dos elementos principales:

- **Behavior Tree:** Es el arbol de comportamientos. Se va ejecutando de forma secuencial
- **Blackboard:** Se encarga de almacenar la información de lo que está ocurriendo. Es un conjunto de variables que podemos vincular con el BehaviorTree

Cuando inicializamos el Behavior Tree en el AIController lo hacemos vinculando un Blackboard

Inteligencia Artificial

Normalmente crearemos en nuestro AIController dos variables para alojar una referencia a BehaviorTree y Blackboard

Aunque no es estrictamente necesario, esto facilita a la hora de trabajar con los Asset. Los asignamos desde BP

Nota: Podemos también hacerlo desde C++ en el constructor

– los componentes encargados de gestionar los comportamientos

```
private:

  UPROPERTY(VisibleAnywhere, Category="AI")
  UBehaviorTreeComponent* BehaviorTreeComponent;

  UPROPERTY(VisibleDefaultsOnly, Category="AI")
  UBlackboardComponent* BlackboardComponent;

public:
```

– los asset que vamos a cargar en los componentes

```
  UPROPERTY(EditDefaultsOnly, Category="AI")
  UBehaviorTree* BehaviorTree;

  UPROPERTY(EditDefaultsOnly, Category="AI")
  UBlackboardData* Blackboard;
```

Inteligencia Artificial

```
private:

    /** Component used for moving along a path. */
    UPROPERTY(VisibleDefaultsOnly, Category = AI)
    TObjectPtr<UPathFollowingComponent> PathFollowingComponent;

public:

    /** Component responsible for behaviors. */
    UPROPERTY(BlueprintReadWrite, Category = AI)
    TObjectPtr<UBrainComponent> BrainComponent;  Unchanged in assets

    UPROPERTY(VisibleDefaultsOnly, Category = AI)
    TObjectPtr<UAIPerceptionComponent> PerceptionComponent;

private:
    UPROPERTY(BlueprintReadOnly, Category = AI, meta = (AllowPrivateAccess = "true"))
    TObjectPtr<UPawnActionsComponent> ActionsComp;

protected:
    /** blackboard */
    UPROPERTY(BlueprintReadOnly, Category = AI, meta = (AllowPrivateAccess = "true"))
    TObjectPtr<UBlackboardComponent> Blackboard;
```

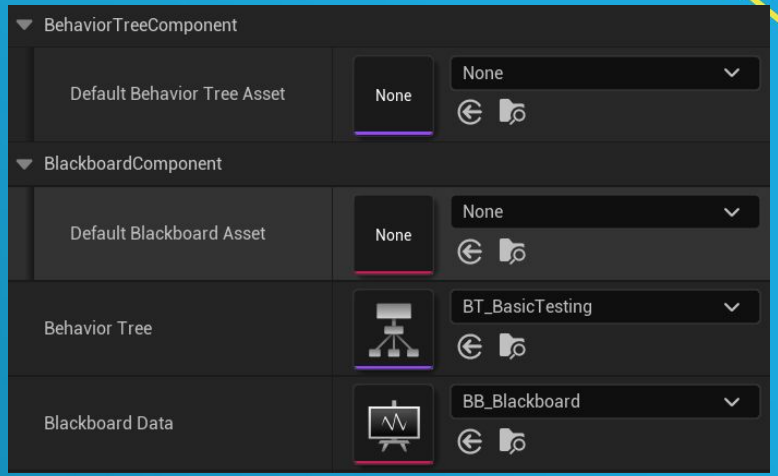
Realmente los AIController ya cuentan con varios de estos componentes

- **Brain Component:** Es una implementación generalista en la que se puede incluir cualquier sistema de IA de toma de decisiones como el BT.
- **Blackboard component:** Es el componente de BB. Podemos usarlo o crear el nuestro.

Inteligencia Artificial

Observación

- Dependiendo del formato de juego, puede tener sentido que los Blackboard y BehaviorTree estén definidos en el personaje que vamos a controlar.



Discusión

¿Qué sería mejor? ¿Dejar los assets en personajes o AIController? ¿Qué ventajas tiene uno u otro?

Inteligencia Artificial

- Una forma de inicializar el Behavior Tree desde el AIController (en OnPossess)

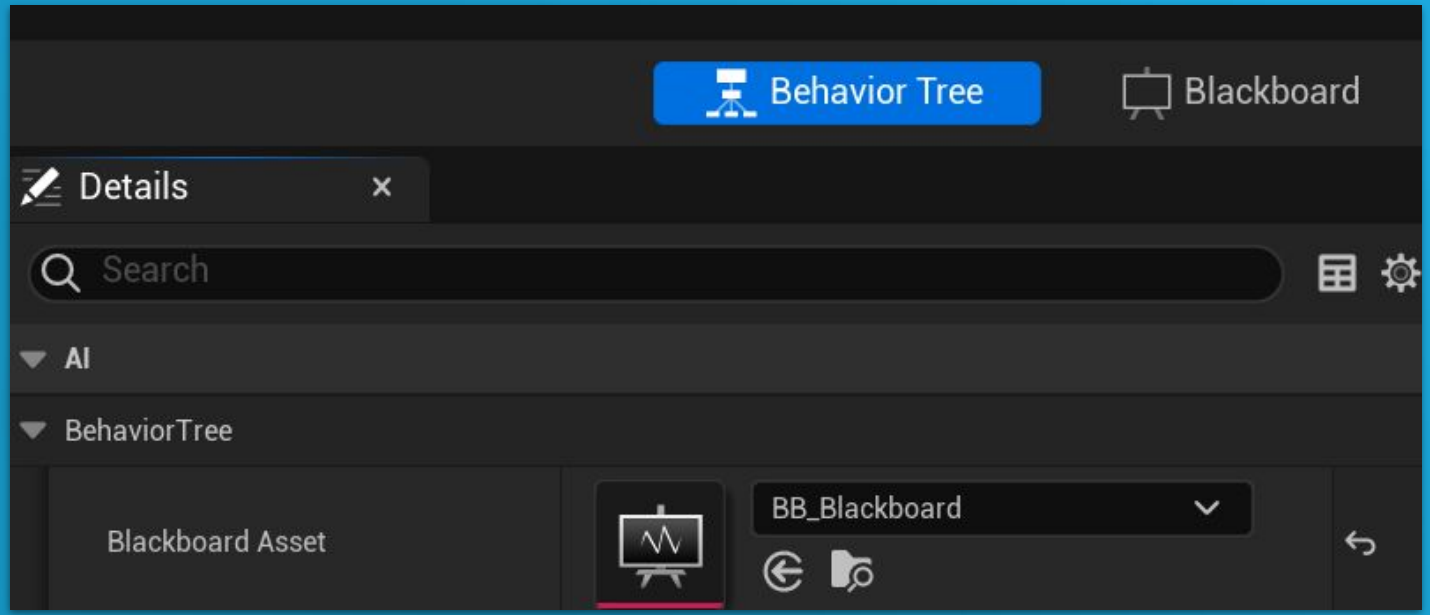
```
if(BlackboardComponent->GetBlackboardAsset())
{
  BlackboardComponent->InitializeBlackboard(*BlackboardComponent->GetBlackboardAsset());
}

if(BehaviorTree)
{
  BehaviorTreeComponent->StartTree(*BehaviorTree);
}
```

- Alternativamente podemos usar RunBehaviorTree. Es una versión simplificada implementada en el AIController

```
RunBehaviorTree(BehaviorTree);
```

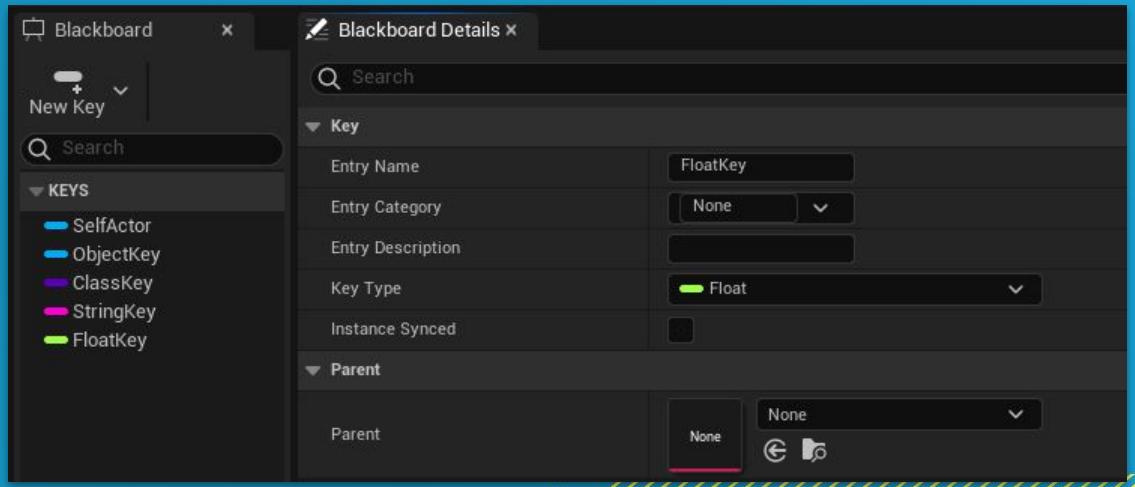
Comúnmente los BT ya tienen asignado un Blackboard que usaremos por defecto



Inteligencia Artificial

Blackboard

- Es una colección de valores que instanciamos para cada entidad (BlackboardKey)
- Almacena información que podremos transmitir al Behavior Tree para decidir los comportamientos

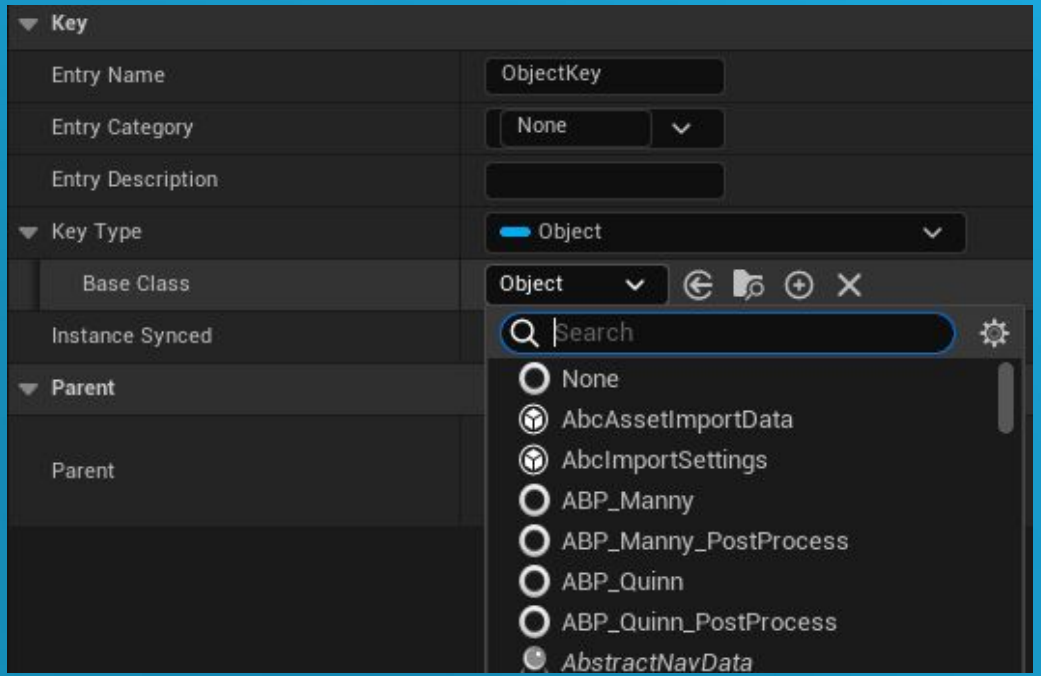


Como veremos más adelante, las entradas del Blackboard son objetos que actúan como *wrapper* de las propiedades

- Está limitado a los tipos básicos que el editor conoce.
- Cada entidad tiene su blackboard, y por tanto los valores de cada blackboard son en principio independientes.
- También podemos definir propiedades que hagan referencia a UObjects.

Inteligencia Artificial

Cómo crear un BlackboardKey que permita almacenar un Objeto de un cierto tipo:



Inteligencia Artificial

Es posible que haya propiedades que se compartan entre todos los Blackboard

Discusión

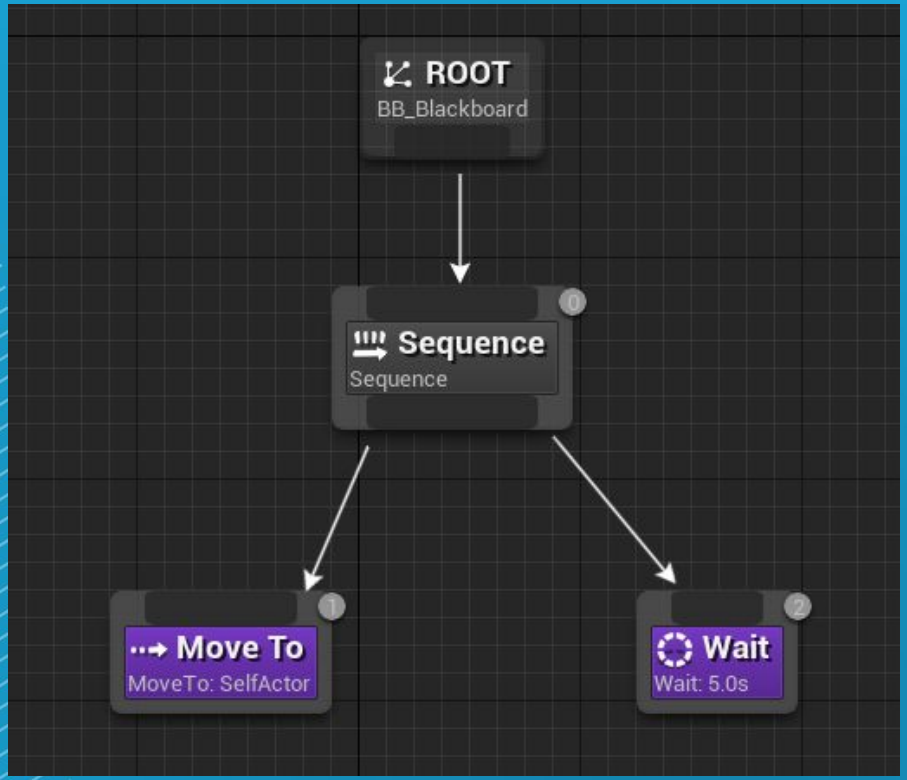
¿En qué casos podría tener esto sentido? Citar algunos ejemplos

Inteligencia Artificial

- Por defecto en todos los árboles existe una propiedad que se llama SelfActor

Es el dueño del arbol, y generalmente se refiere al Pawn que controlamos

Inteligencia Artificial



Behavior Tree

- Es un arbol de nodos
- Los nodos se van ejecutando de izquierda a derecha (la prioridad se indica con un índice)
- Comienzan a partir de un Root

Tipos de nodos

- **Tareas:** Son ejecuciones, se encargan principalmente de hacer que el agente haga algo (desplazamiento, ataques, etc...)
- **Services:** Principalmente pensados para quedarse “observando” elementos del entorno e ir actualizando el Blackboard
- **Decorators:** Definen principalmente condiciones sobre los nodos para saber si podemos o no ejecutarlos o pasar de largo.
- **Control de flujo:** Decide cómo continuar ejecutando una rama en función del resultado de las tareas.

Inteligencia Artificial

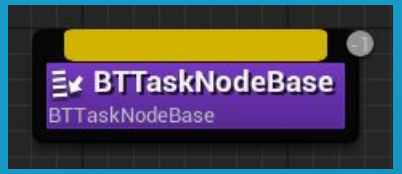
Sequence:

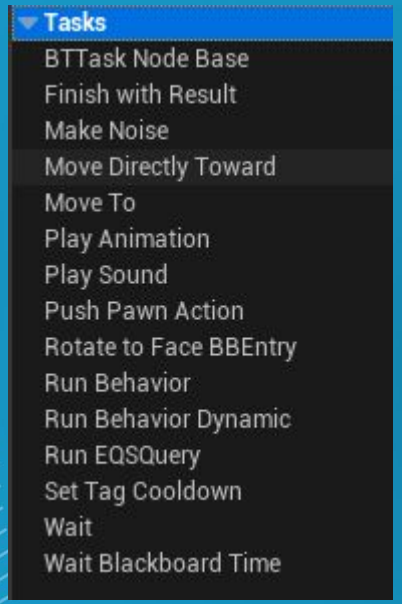
- Lanza un aviso de ejecutar los nodos que parten de este en orden
  - Si uno de los nodos devuelve un fracaso, se detiene y empieza de nuevo
- Si la secuencia se ejecuta por completo devuelve un éxito
- Si la secuencia no termina, devuelve un fracaso

Selector:

- Lanza un aviso de ejecutar los nodos que parten de este en orden
  - Si uno de los nodos devuelve un éxito, se detiene y empieza de nuevo
- Si la secuencia se ejecuta por completo devuelve un fracaso
- Si la secuencia no termina, devuelve un éxito

Inteligencia Artificial

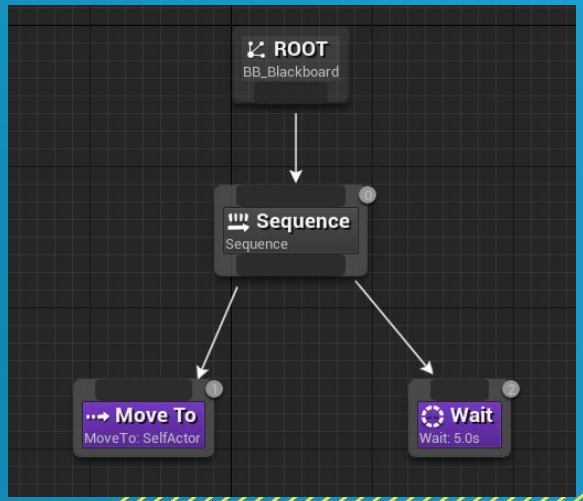




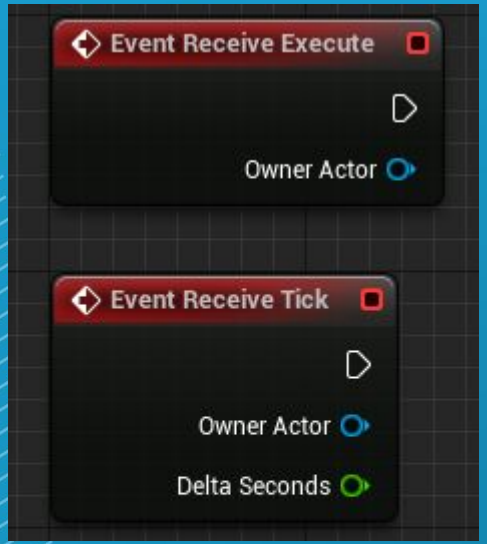
Tasks

- Son tareas que ejecutan comportamientos (atacar, perseguir, moverse...)
- Todas las tareas pueden devolver un éxito o fracaso
- Son el final de una rama

Ejemplo:



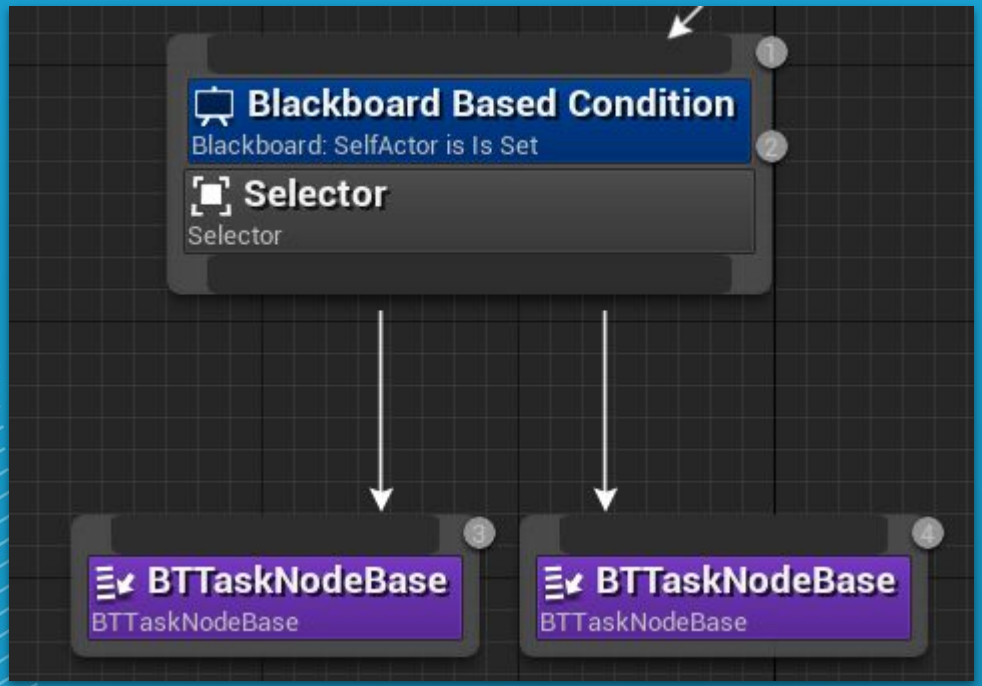
Inteligencia Artificial



Tasks

- Tiene tres eventos que podemos usar
  - **ReceiveExecute**: Se dispara al recibir un execute (por ejemplo estando dentro de un nodo Sequence)
  - **ReceiveTick**: Se ejecuta mientras el BT esté centrado en esa tarea

Inteligencia Artificial

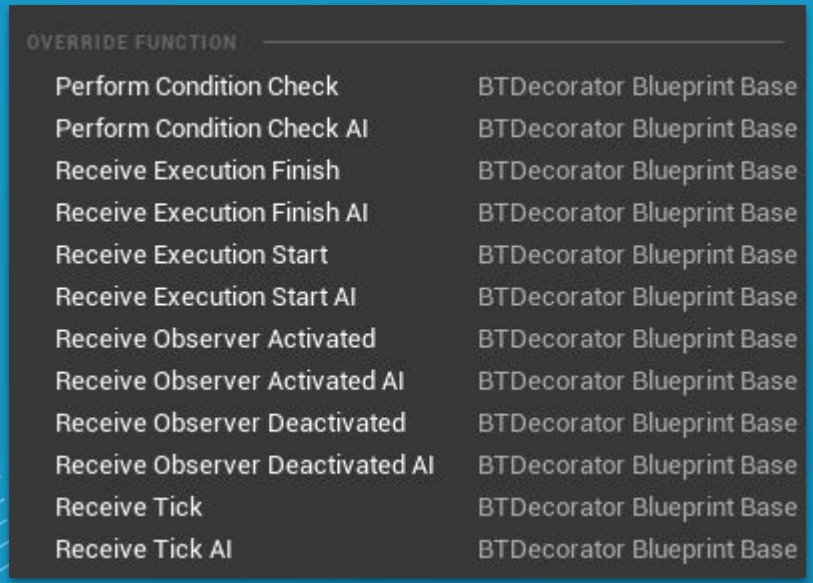


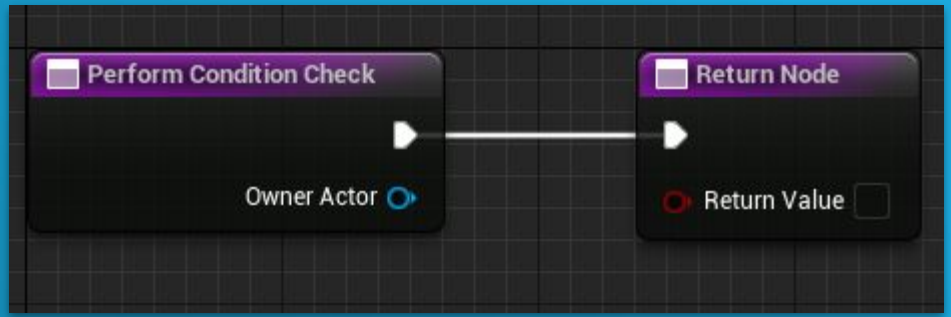
Decoradores

- Implementan una condición a evaluar
- Se añaden a otros nodos para comprobar si estos se pueden ejecutar o no
  - ¿Existe una posición definida a la que moverme?
  - ¿Tengo suficiente munición para disparar?

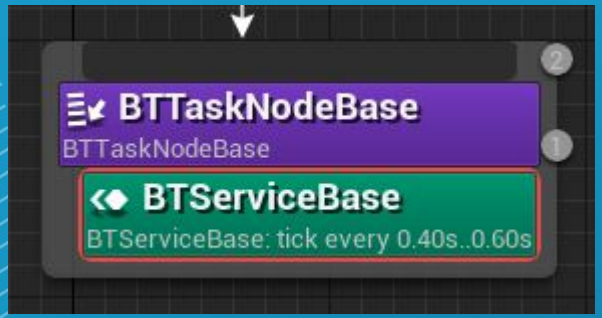
Inteligencia Artificial

Principalmente sobreescribimos las funciones de ConditionCheck





Inteligencia Artificial

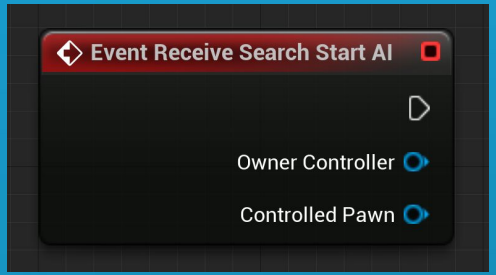


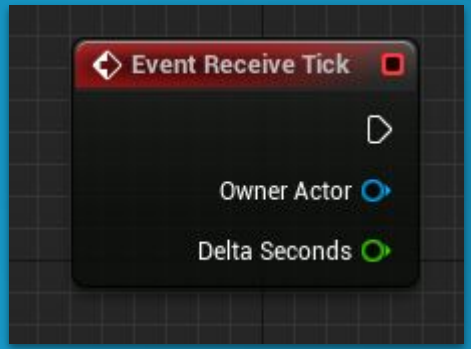
Services

- Funcionan como complemento a un nodo
- Generalmente se emplean para actualizar valores del Blackboard
- Usan un tick a una cierta frecuencia

Una de las propiedades más interesantes que tienen es que se ejecutan **únicamente** cuando un nodo de la rama en la que se encuentran se está ejecutando

Inteligencia Artificial





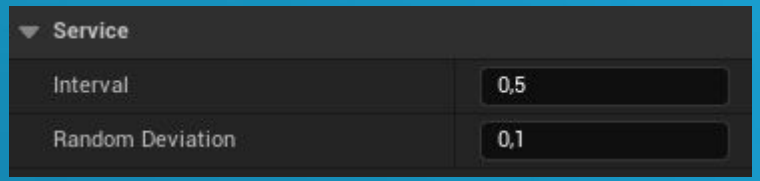
Services

SearchStart: Se dispara al hacer la ejecución del nodo (Task) o si es una secuencia de control de flujo y entramos en ella.

Tick: Generalmente usamos el método ReceiveTick en BP (es común con el que tenemos en el BTTask)

Inteligencia Artificial

Se pueden definir un cierto rango de desviación (por cada instancia)

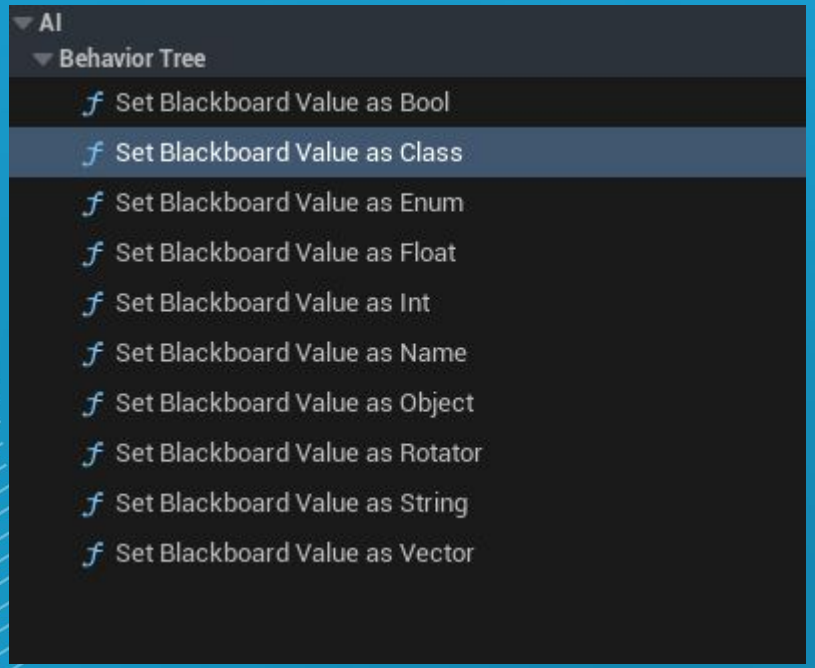


Inteligencia Artificial

Cómo parametrizar los nodos de un Behavior Tree

- Generalmente definimos una variable que permita conectar con los BlackboardKey de los Blackboard
- Dentro de la tarea/decorador/servicio extraemos el valor al que hace referencia
- Podemos tanto acceder a estos valores como modificarlos

Inteligencia Artificial



- **Importante:** Tenemos que asegurarnos que estamos accediendo usando el método apropiado

En caso contrario obtendremos un valor nulo

Inteligencia Artificial

Recomendaciones

- Minimizar dentro de lo posible el uso del Tick en los Tasks
- En los Services definir apropiadamente el delta time del Tick
- Es conveniente que reutilicemos los nodos que creamos para hacer cosas diferentes. Para esto podemos parametrizar por medio de los BlackboardKeySelectors
- **SIEMPRE** hay que asegurarse de que las tareas se terminan (con o sin éxito). Por ejemplo si usamos un Branch o un IsValid, devolver un fin de ejecución en cada camino.
- **Divide and Conquer -** evita crear una que haga todo, crea pequeñas tareas suficientemente personalizables

Inteligencia Artificial

¡Vamos a ver algunos ejemplos!

Inteligencia Artificial

Tasks en C++

- Es el equivalente al ReceiveExecute

```
EBTNodeResult::Type ExecuteTask(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory) override;
```

- Comúnmente ejecuta un comportamiento y devuelve un `EBTNodeResult` que indica el resultado de la ejecución:

```
EBTNodeResult::Succeeded;
EBTNodeResult::Aborted;
EBTNodeResult::InProgress;
EBTNodeResult::Failed;
```

Igual que en BP, la tarea debe devolver un resultado. Si no devuelve `Succeeded` o `Failed` el sistema se quedará atascado en la tarea

Inteligencia Artificial

Tasks en C++

`InProgress` puede permitirnos crear tareas algo más complejas:

- Una tarea que se queda haciendo algo en base al Tick (hay que setear `bNotifyTick = true;`
- Un callback que definimos para finalizar la acción

IMPORTANTE: Si emitimos un InProgress, tenemos que asegurarnos de finalizar manualmente la tarea.

En un BehaviorTree, cuando estamos ejecutando un BTTask esta se convierte en la tarea latente. Podemos finalizar esa tarea en caso de que se haya quedado pendiente.

```
FinishLatentTask(OwnerComp, EBTNodeResult::Succeeded);
```

Inteligencia Artificial

Vincular BlackboardKey al BTTask

- Hacemos uso del FBlackboardKeySelector en el BTTask

```
UPROPERTY(EditAnywhere)
FBlackboardKeySelector  KeySelector;
```

- Para operar con los KeySelector hay que usar métodos propios del BlackboardComponent

```
OwnerComp.GetBlackboardComponent()->GetValueAsString(KeySelector.SelectedKeyName);

OwnerComp.GetBlackboardComponent()->SetValueAsString(KeySelector.SelectedKeyName,
                                                     TEXT("SomeValue"));
```

Generalmente todas las funciones del BTTask (y otros nodos) tienen un UBehaviorTreeComponent& OwnerComp del que podemos sacar el Owner y a su vez el Blackboard

Inteligencia Artificial

Vincular BlackboardKey al BTTask

- Es posible forzar que los Blackboard key solo acepten ciertos tipos de valores desde C++

```
BlackboardKey.AddObjectFilter(this, GET_MEMBER_NAME_CHECKED(UBTTask_MoveTo, BlackboardKey), AActor::StaticClass());
BlackboardKey.AddVectorFilter(this, GET_MEMBER_NAME_CHECKED(UBTTask_MoveTo, BlackboardKey));
```

Importante: Esto se debe hacer desde el constructor

Inteligencia Artificial

Decorators

En el caso de los decoradores, generalmente solo tendremos que sobreescribir:

```
bool CalculateRawConditionValue(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory) const override;
```

Inteligencia Artificial

Services

En el caso de los servicios, generalmente usamos las funciones de Search y Tick:

```
virtual void OnSearchStart(FBehaviorTreeSearchData& SearchData) override;
virtual void TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float DeltaSeconds) override;
```

Nota final

Inteligencia Artificial

- Más allá de lo que hemos visto, todo el sistema de IA en Unreal Engine está diseñado para que podamos reemplazar las estructuras que ya existen
  - El sistema de **Navegación** (navmesh, componente de pathfinding, etc...)
  - El **AIController** implementa métodos que hablan con todo el sistema. Podemos sobreescribirlos en base a nuestras necesidades
  - El **PerceptionSystem** puede ser reemplazado por uno propio más simplificado o con más utilidades
  - El dúo **BehaviorTree** y **Blackboard** puede ser descartado a favor de otros sistemas más convenientes

Inteligencia Artificial

- **Otros sistemas interesantes:**
  - **Environment Query System in Unreal Engine**: Consultas complejas sobre el entorno
  - **State Tree in Unreal Engine** **:** Una combinación entre BT y máquinas de estado
  - **MassEntity** : Un Framework ECS para gestionar un número obsceno de agentes
  - **Smart Objects in Unreal Engine**: Objetos con bloqueos para interacciones entre múltiples agentes

Ejercicios

Ejercicios

Ejercicio 1: Definir un AIController para todos los enemigos del juego. Cuando los enemigos sean poseídos, deben almacenar una referencia al PlayerController que haya en ese momento en el juego.

Ejercicio 2: Por medio del sistema de percepción, definir el sentido de vista. Hacer que los enemigos sean capaces de detectar al jugador y se giren hacia este.

Ejercicio 3: Implementar un sentido auditivo en los enemigos. Cada vez que el personaje jugador salte y aterrice debe emitir un sonido que alerte de su presencia.

Ejercicio 4: **Mente colmena -** Todos los enemigos actúan como uno. Cuando un enemigo detecta al jugador, todos los demás deben recibir la información del jugador, el punto en el que ha sido detectado, etc... Hacer uso de un gestor.

Ejercicio 4.1: Repetir el ejercicio pero esta vez usando un Blackboard y haciendo uso del bInstanceSynced

Ejercicio 5: Crear un BT para patrullar entre dos puntos definidos.

Ejercicios

Ejercicio 6: Crear la siguiente secuencia por medio de un BT:

1. Si he detectado al jugador lo persigo
2. Si no lo he detectado o lo he perdido de vista, me quedo patrullando entre un número de posiciones determinado

Ejercicio 7: Implementar una tabla de correlaciones entre un número de facciones determinado. Dentro de la tabla tenemos que definir el nivel de afinidad que existe usando ETeamAttitude.

Ejercicio 7.1: Haciendo uso de la tabla que hemos definido, implementar la lógica de percepción entre estas facciones siguiendo lo que hemos visto en clase.

Ejercicios

Ejercicio 8: Por medio de un BT, crear un compañero que nos siga a lo largo del nivel. Este NPC debe comportarse de la siguiente forma:

- ❏ Si el jugador se mueve, debe seguirlo. Crear un umbral de distancia para que se mueva cuando ya esté a una cierta distancia.
- ❏ Si el jugador se para, se debe colocar a una cierta distancia de este.
- ❏ Cuando el jugador salte, el NPC debe saltar.
- ❏ Si ve a un enemigo, debe priorizar huír de este.