# Unreal Engine 5 - Lesson 8&9 - AI

Nicolas Serf - Gameplay Programmer - Wolcen Studio
serf.nicolas@gmail.com

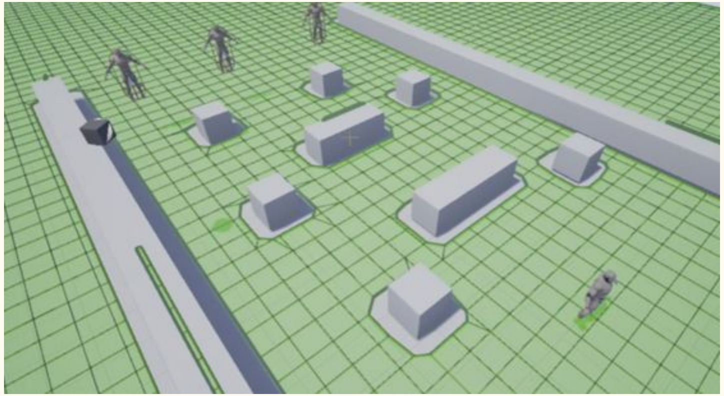
# Summary

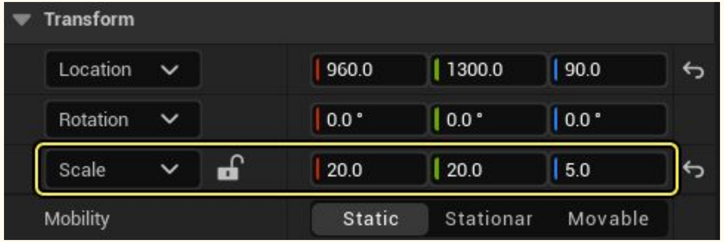
- Navigation Systems
  - Basics
  - Modifier Volume
  - Link Proxies
  - Generating
  - Custom Navigation Area & Queries
  - Avoidance
    - RVO
    - DCM
- Smart Objects
  - Overview
  - Searching
  - Claiming
  - Approaching
  - Releasing
- Behavior Trees
  - Biggest differences
  - Node instancing
  - Nodes
- Environment Query System
- AIPerception

# Navigation System - Basics

- To make it possible to **find** a path between a **start location** and a **destination**, a **Navigation Mesh** is generated from the **world's collision geometry**. This simplified **polygon** mesh represents the **navigable space** in the Level. Default settings subdivide the Navigation Mesh into tiles to allow rebuilding localized parts of the Navigation Mesh.
- The **resulting mesh** is made of **polygons** and a **cost** is associated with each **polygon**. While **searching** for a path, the **pathfinding** algorithm will attempt to find an **optimal path** with the **lowest cost** to the destination.

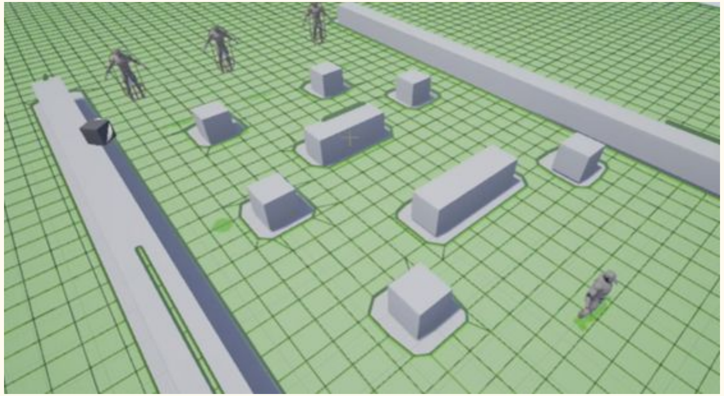
- In order to **generate** a **NavMesh**, it is **mandatory** that you add a **specific actor** into your level which is called **NavMeshBoundsVolume**.
- With the **NavMeshBoundsVolume** selected, go to the **Details panel** and scale the volume to a value that **ensure** to **contains every mesh** that needs to be taken into **consideration** will **baking** the **Navmesh**.
- In order to **visualize** the generated Nav Mesh, you can press **P** key
- You may have some issue regarding the **NavMesh**, with some **artifacts** on **elevation**. It is due to **Draw Offset** value which is located on the actor **RecastNavMesh-Default**. On the **detail** panel of that actor, you can adjusts the **height offset** where the Navigation Mesh is **drawn** for better readability.
- From **RecastNavMesh-Default**, you can modify various **display options** like
  - Draw Poly Edges
  - Draw Tile Bounds
  - Generally speaking, you can modify how it is **generated** in the **Generation section**

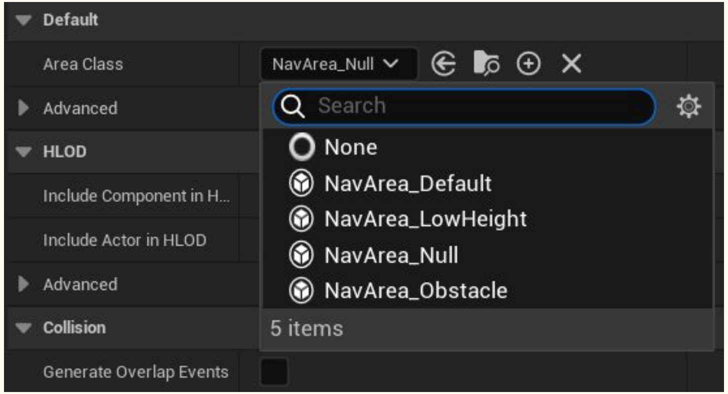




# Navigation System - Navigation Modifier Volume

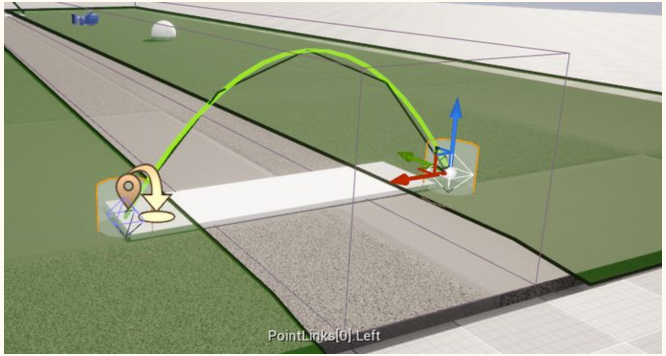
- If you would like to **modifying** the generate NavMesh, the **1st** most obvious is to add some **actors** inside the level with a **collision** on them
  - This simple operation will make the **NavMesh** to include this actor as **interfering** with the NavMesh and therefore creating an **obstacle** in it
- Another solution is to works with **Navigation Modifier Volume**.
- It is an actor that you can find from the **actor finder**
- This can be used to change the **properties** of the **polygons** within the volume space to modify their **traversal cost**
- The polygon **properties** are defined by the appropriate **Area Class** of the Navigation Modifier Volume. This class determines the **effect** on the Navigation Mesh. You can use the **built-in** classes to modify the mesh or create your **custom implementations**.
- As you can see, by default there is some default UObject classes for the Area Class
  - **NavArea_Default** : Assigns the same navigation cost to the area inside the volume and the Navigation Mesh by default.
  - **NavArea_LowHeight** : The Navigation Mesh will not generate navigation data inside this volume.
  - **NavArea_Null** : The Navigation Mesh will not generate navigation data inside this volume.
  - **NavArea_Obstacle** : Assigns a high navigation cost to the area inside the volume.
- Based on the **cost of this Area Class** and the **global cost** of the **path**, your agent will try to **avoid** way more paths that **includes more costly areas**





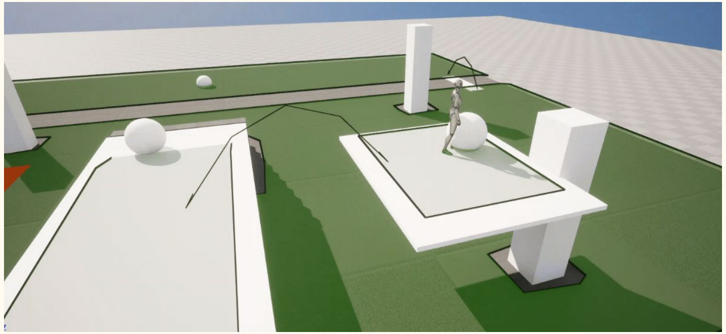
# Navigation System - Navigation Link Proxies

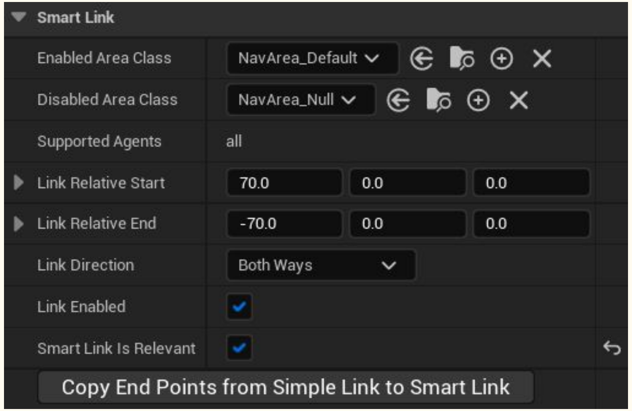
- **Navigation Link Proxies connect** two areas of the Navigation Mesh that don't have a **direct navigation path**. While a path is being **searched**, Navigation Link Proxies are used as **extra connections** that Agents can use to reach their destination.
- Navigation Link Proxies are commonly used to **build bridges** between areas with separate **Navigation Meshes** and to instruct **Agents** that they can **fall** or **jump** from a platform towards their goal when there is no continuous navigation path available.
- A **NavLinkProxy** is an **actor** to place in the **world**
- With the **Nav Link Proxy selected**, click the **PointLinks[0].Left** handle and move it so it's placed on **one side** of the mesh. Click the **PointLinks[0].Right** handle and move it so it's placed on the **opposite side** of the mesh. This is how you define the bridge
- With the **Nav Link Proxy** selected, go to the **Details** panel and expand the section labeled **0** under **Point Links** to find the Direction dropdown. You can choose to have the **Direction** be **Both Ways**, **Left to Right**, or **Right to Left**

# Navigation System - Smart Link

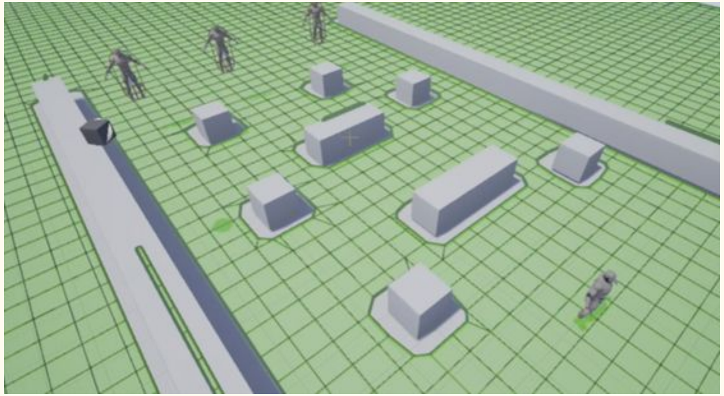
- In this section, you will learn how to use a **Smart Link** on a **Nav Link Proxy** to allow your Agent to **jump** from **one platform** to **another.**
- Create a **BP class** which **inherit** from **NavLinkProxy**
- In the event graph, you can listen to an event called **ReceiveSmartLinkReached**
- From within this event, you can create a **graph** as complex as you want in order to **compute correctly** how you can this link to **behave** on the **actor** that is **crossing** it
- There is a useful when it comes to **computation** of **velocity** while **jumping** from an edge which is **SuggestProjectileVelocityCustomArc**
- Do not forget to click the button **Copy End Points from Simple Link to Smart Link** and **tick** the correct **options**





# Navigation System - Generating Nav Mesh

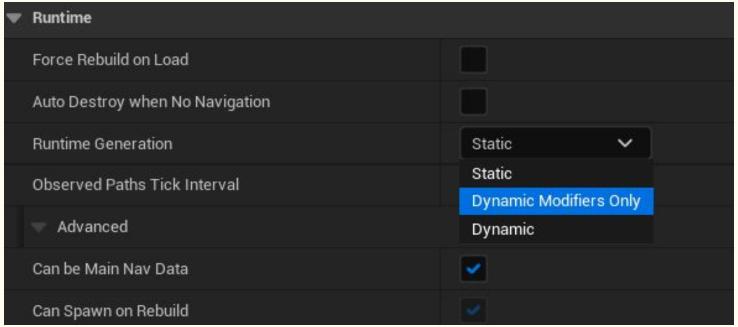
- Unreal Engine comes with **three** Navigation Mesh **generation** modes
    - **Static**
        - The Navigation Mesh is **generated offline** and is **saved** with the **Level**. The Navigation Mesh is **loaded** at **runtime** and **cannot change**.
    - **Dynamic**
        - The **Navigation Mesh** is **generated offline** and is **saved** with the **Level** or **built** at **runtime**. At runtime, the **navigation-relevant data** used by the Navigation Mesh can be **updated** and the generation is performed on the tiles affected by the change in the data.
    - **Dynamic Modifiers Only**
        - The Navigation Mesh is **generated offline** and is **saved** with the **Level**. At **runtime**, only **Navigation Modifiers** such as **Navigation Areas**, **Navigation Links**, and **dynamic objects**, can **modify** the existing Navigation Mesh by changing the **cost** or **block areas.** No new Navigation Mesh surfaces are generated at runtime.
        - This method allows the Navigation Mesh to **cache collision data** and can result in up to **50% cheaper processing** of the affected tiles.
        - Advanced users should use this mode after **careful consideration** of its **benefits** and **limitations**.
- By default, the Navigation Mesh is configured to be **Static**. However, you can set your **Navigation Mesh generation** to one of the dynamic modes so it can change at runtime.



# Navigation System - Generating Nav Mesh - Runtime Generation

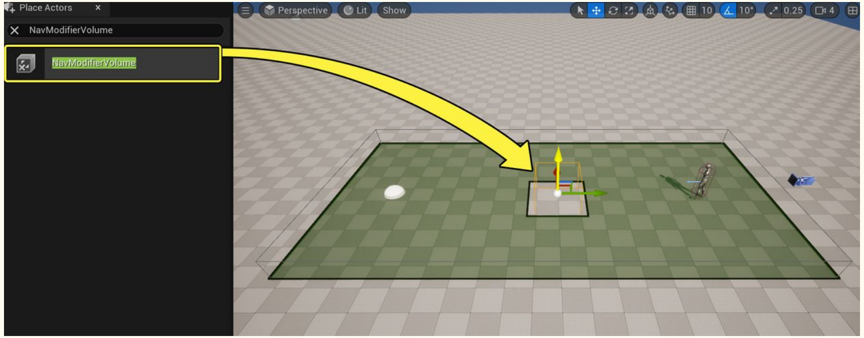
- In order to modify the generation setting, go into **Project Settings**, **Navigation Mesh** under **Runtime section** and in **runtime generation dropdown**, select the option you need
- In order to have a **dynamic modifier**, you need to **add** an component which is called **NavModifier**.
- It is **important** to say once again that **no new Navigation Mesh is being generated** in the Level. The Navigation Modifier is simply **changing** the **existing** Navigation Mesh.

- Following the same idea, you can select **Dynamic** as **Runtime Generation**
- In this situation, it will be **easier** to setup the navmesh to update **dynamically**, but it will be way more costly than **Dynamic Modifiers Only**



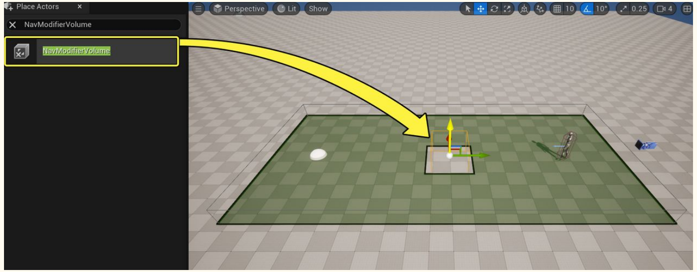
# Navigation System - Custom Navigation Areas and Query Filters

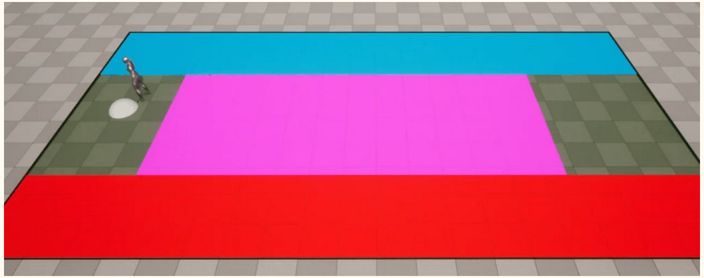
- The Agent determines the **most optimal route** to its destination by comparing the **cost** of each navigation **polygon** within the Navigation Mesh. If all polygons along the route are of equal cost, then the Agent will choose the **shortest path** to its target (usually a straight line).
- You can **influence** the cost of the navigation polygons using **Navigation Modifier Volumes** and **Navigation Query Filters**.
- **Navigation Modifier Volumes** use **Area Classes** to determine the **Default Cost** multiplier of navigation within the volume. Area Classes also define the **Fixed Area Entering Cost,** which is the **initial cost** applied when the Agent enters the area. You can create as **many Area Classes** as needed to influence how your Agents navigate your Level.
- **Navigation Query Filters** contains information about **one** or **more** Area Classes and can **override the cost values** if needed. You can **create** as many Query Filters as needed to further customize how your Agents navigate your Level.

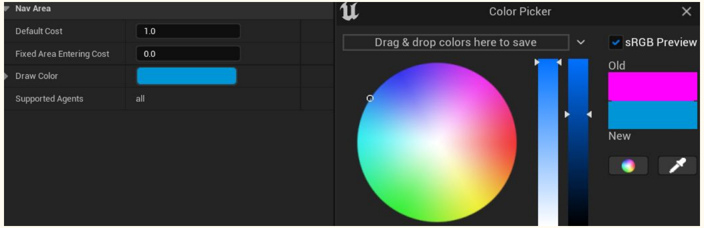


# Navigation System - Custom Navigation Areas and Query Filters

- Obviously, in order to **create** a new area class, you'll need to create blueprint / c++ class which in inheriting from **NavArea**
- You can **modify** debug color which will be easier for **level designer** to **visualize** how they are setting the map
- You can then modify **actual cost** which will not be visible by level designer but is used for **computation**

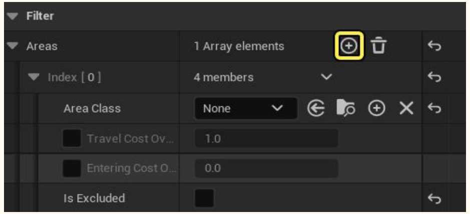


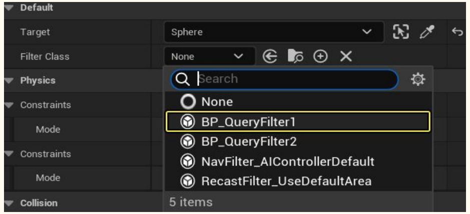





# Navigation System - Custom Navigation Areas and Query Filters

- To create a new **navigation query filter**, you need to create a **blueprint / c++ class** which inherit from **NavigationQueryFilter**
- From within the detail panel, you have access to an **array** which will allow you this **query filter** to specifier how it should **interact** with **specific Area class**
    - **Area Class** : The class which we talk just previously
    - **Travel cost override** : Ignore the **default cost parametrize** in the **area class** if ticked and take this value instead
    - **Entering cost override** : Ignore the **default entering cost parametrize** in the **area class** if ticked and take this value instead
- When you are calling **MoveToActor** which comes from **AIController**, you have an option to specific the **Filter class** that needs to be used during the computation



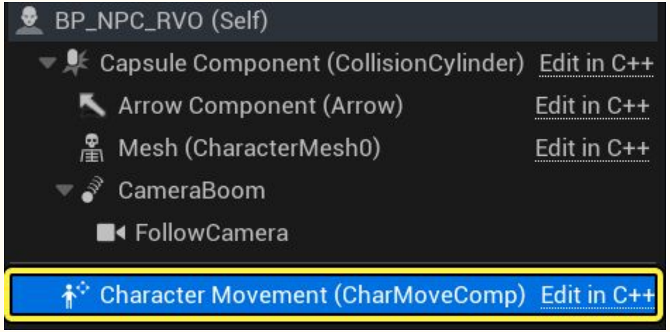

# Navigation System - Using Avoidance

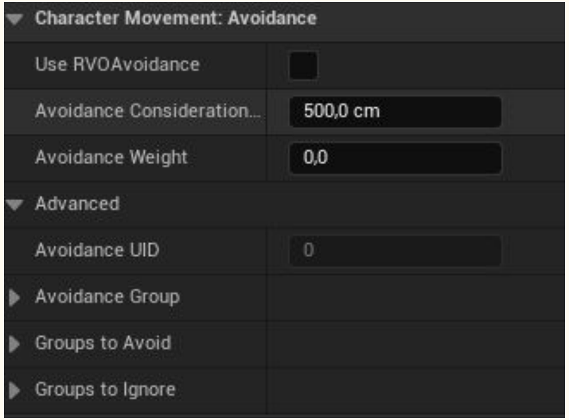
- While **pathfinding** can **determine a path around objects that don't move**, **avoidance** algorithms are better suited to handle **moving obstacles.**
- There are **two methods** of avoidance the Agents can use to navigate around dynamic obstacles and each other
  - **Reciprocal Velocity Obstacles (RVO)**
    - The **RVO** system calculates the **velocity** vectors for **each Agent** to avoid a **collision** with nearby Agents. This system looks at the **nearby Agents** and assumes they are traveling with a **constant velocity** within each time step of the calculation. The **optimal velocity vector chosen** is the closest match to the **Agent's preferred velocity** in the direction of its target.
    - Unreal Engine's **implementation** of RVO includes **optimizations** to **reduce frame rate dependencies**. It also includes **improvements** to avoid **constant path recalculation** and a priority system to help resolve potential collisions. RVO **does not use the Navigation Mesh** for avoidance, so it can be used **separately** from the Navigation System for any Character. It is included in the **Character Movement component** of the Character class.
  - **Detour Crowd Manager (DCM)**
    - The **DCM** system handles avoidance between Agents by using an **adaptive RVO sampling calculation**. It does this by **calculating** a **coarse sample of velocities** with a **bias** toward the **Agent's direction** that allows for a **significant improvement** in the quality of the avoidance over the traditional RVO method. This system also uses **visibility** and **topology path corridor** optimizations to further improve collision avoidance.
    - The Detour Crowd Manager system is **highly configurable** with **options** to specify **sampling patterns**, the **maximum** number of **Agents**, and **Agent Radius**. The system is included in the **DetourCrowd AI Controller** class and can be used with any Pawn class.

| Method | Description | Limitation |
|---|---|---|
| RVO | • Agents avoid obstacles by using velocity vectors within a specified radius.<br>• Included in the Character Movement component of the Character class. | • Less configurable compared to the Detour Crowd Manager.<br>• Limited to the Character class.<br>• Does not use the Navigation Mesh for avoidance so Agents can potentially go "out of bounds". |
| DCM | • Agents avoid obstacles by using path optimizations along with velocity vectors within a specified radius.<br>• Included in the DetourCrowd AI Controller class. Has a fixed maximum number of Agents defined in the project settings. | • Has a fixed maximum number of Agents defined in the project settings. |

# Navigation System - Using Avoidance - RVO

- Given RVO is located on the **character movement component**, 1st step is obviously to add a **Character Movement Component** into the **actor** that needs to have **avoidance** through **RVO**
- In the details we can configure if we want to use the **avoidance** and some **parameters**
    - **UseRVOAvoidance** : Should RVO algorithm be used
    - **AvoidanceConsiderationRadius** : RVO algorithm **only consider** obstacles that fall into this radius. It is an important parameter to tweak
    - **AvoidanceWeight** : How **heavy** RVO algorithm needs to **intervene**. **0.5** is default value which works in most cases
    - **Avoidance UID** : Automatically generated and important for interacting with **AvoidanceManager** in C++
    - **Avoidance Group** : Which group this character **belongs** to
    - **GroupsToAvoid** : Which group this character should **avoid**
    - **GroupsToIgnore** : Which group this character should **ignore**
- In the standard version it is as easy as that
- Obviously, given everything is open in Unreal, if you want to override how **RVO** works, you could decide to override the **CharacterMovementComponent** but it is a **big chunk**

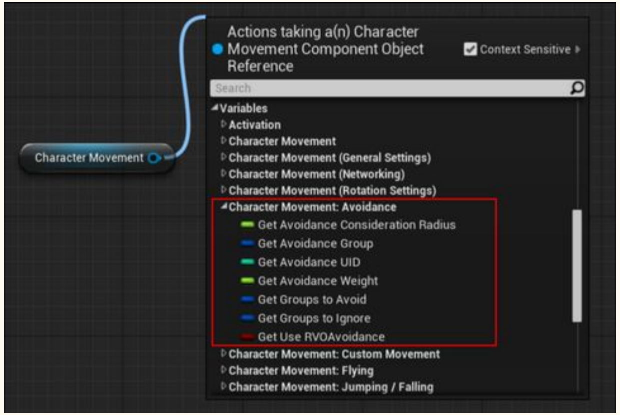


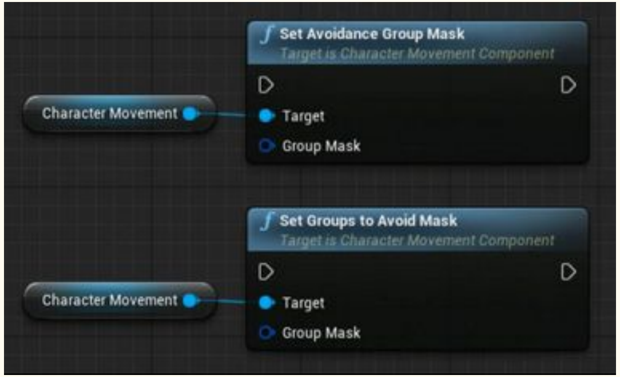



# Navigation System - Using Avoidance - Advanced RVO

- If you have a reference to the **CharacterMovement**, from blueprint you'll see that you can **read** all the **variables** and **parameters** we talked the slide before.
- From Blueprint, we can't sadly modify this values.
- However, we can still modify **slightly** the behavior of **RVO**. We have access to
  - **Enabling RVO algorithm**
  - **Set the avoidance group**
  - **Set the group to avoid**

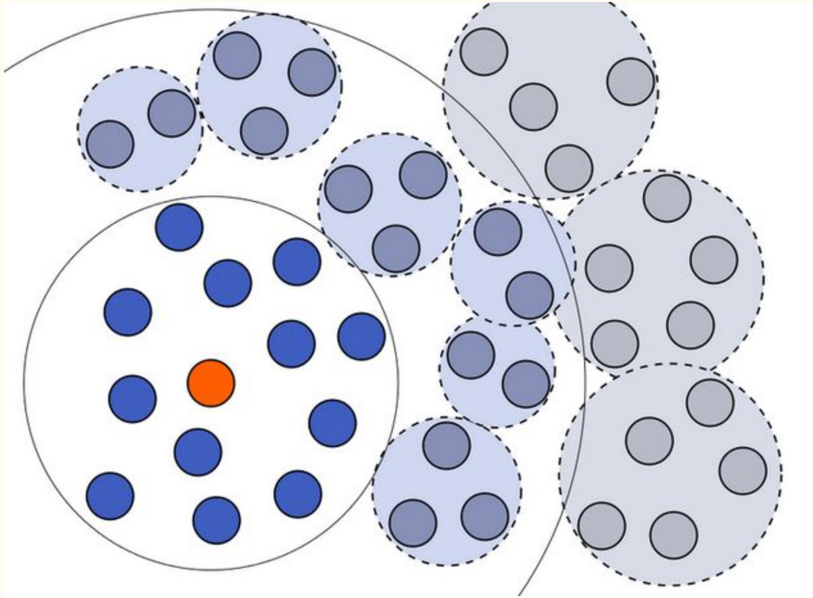
- From C++, you have a new functionalities from which you'll be able to modify the RVO.
  - 1st, you have access to **UAvoidanceManager**, which stores **data** of **all agent** that use **RVO**
  - Thanks to **AvoidanceUID**, you can **query** the **Avoidance Manager** to get **FNavAvoidanceData** struct which hold specific avoidance **data** of the character.
  - You could for example have access to the current velocity avoidance through **GetAvoidanceVelocity()**



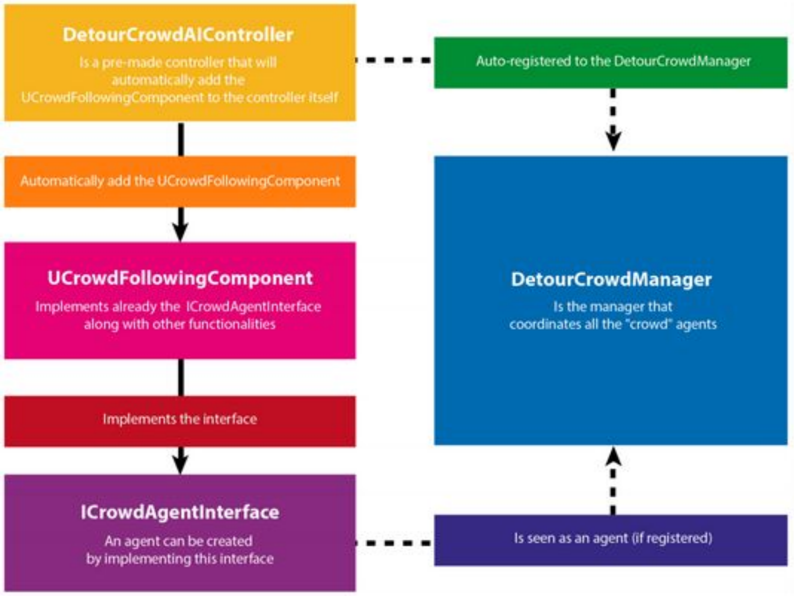

# Navigation System - Using Avoidance - Advanced RVO

- It important to understand that RVO do not take into account the NavMesh, which could mean that the agent get pushed out of the navmesh.
    - Even if it is quite **minor**, it is something that you need to take into **account**
    - You could choose to have **check** while moving to ensure that you **stick** into the **NavMesh**
    - You could choose to have **wall boundaries** into your levels to ensure that the movement out of the navmesh is not possible
    - etc...
- If you want RVO to **works** on **non-character** actor, you'll either needs to **recreate RVO algorithm**, or **adapt** your actor to uses the **CharacterMovementComponent**
- RVO do **not works very well** when there is a **large** number of agent in a **confined space**
- RVO is really **fast**
- Finally, if you want more detail about the avoidance RVO... Check the **source code** at Runtime/Engine/Classes/AI/Navigation/AvoidanceManager.h



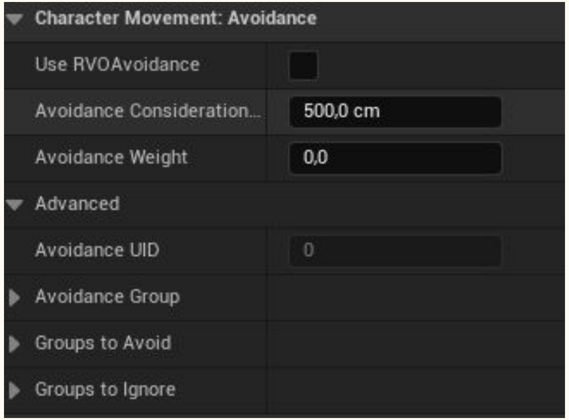
# Navigation System - Using Avoidance - DCM

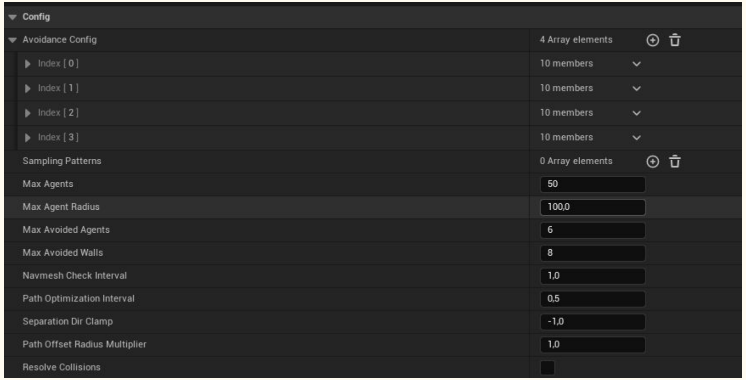
- Within a world there is always an object called **DetourCrownManager**.
    - It is a **singleton**
    - It is responsible for **coordinating crowds** in the game
    - An actor needs to be **registered** into the **DetourCrownManager** to be considered by the **system**
    - The **DetourCrownManager** accepts anything that implements the **ICrowdAgentInterface**, which provides **data** to the Manager
- Even if you could create an actor which implement the interface, Unreal provides a **special component** called **UCrowdFollowingComponent** which implement the interface
    - It makes directly the agent registered for the **DetourCrownManager**
    - It activate by default the **Detour Behavior**
- To make things easier, Unreal provides a **special controller** called **DetourCrownAIController** which you can select in the **dropdown** of the pawn
- Obviously, you need to **inherit** from this special controller in either blueprint or c++



# Navigation System - Using Avoidance - DCM

- Note that **UCrowdFollowingComponent** implement the ICrowdAgentInterface which internally uses the avoidance settings located in the **CharacterMovementComponent**
  - Therefore, the settings for **RVO** are still **impacting** the **DCM**
  - **Weight will not be used**
  - Notice that you should **not activate RVO & DCM** at the same time except if you know what you are doing
- From this point, you can go into **Project Settings** and navigate in **Crow Manager** section to **parametrize** furthermore the DCM
  - In this section, you can adjust **several settings** for the **Detour Crowd Manager system**, such as the **Max Agents** used by the system and the **Max Agent Radius** used for the avoidance calculation.
  - There is **4 avoidance config** which correspond to quality : **Low**, **Medium**, **Good**, **High**
  - The quality value is set in **UCrowdFollowingComponent** with the **Avoidance Quality**

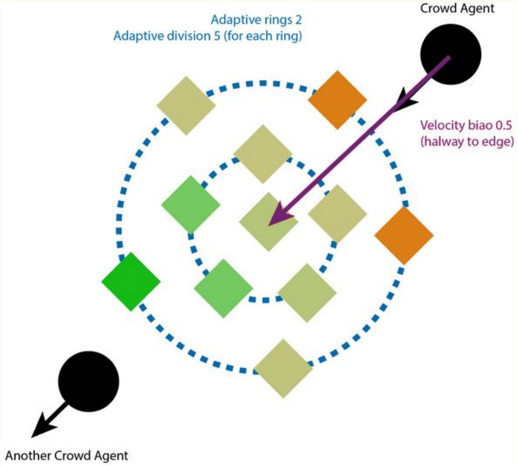


# Navigation System - Using Avoidance - DCM

- In order to fully understand the settings in **avoidance config**, it would require to **understand** how the **algorithm** works
- Let’s make a **simplified** version just to get the idea, you can **always document yourself** for more details

- Algorithm does a **sampling** by creating a **set of rings** (**Adaptative Rings**) around the center point (where the agent is) with a bias (**Velocity Bias**) in the direction of the velocity
- Each ring is sampled (divided) by **Adaptative Divisions**
- Then the algorithm recursively **refines** the search by using a **smaller set** of **rings** which are **centered** on the best sample of the **previous iteration**
- The algorithm repeat this process **Adaptative Depth** times
- During each iteration, the best sample is considering the following
    - Does the direction of the agent match the current velocity ?
      The weight is **DesiredVelocityWeight**
    - Does the agent goes sideway ?
      The weight is **SideBiasWeight**
    - Does the agent collides with any known obstacles ?
      The weight is **ImpactTimeWeight**
      It scans a **range** by considering the **current velocity** within **ImpactTimeRange** parameter




| | |
|---|---|
| Velocity Bias | 0,5 |
| Desired Velocity Weight | 2,0 |
| Current Velocity Weight | 0,75 |
| Side Bias Weight | 0,75 |
| Impact Time Weight | 2,5 |
| Impact Time Range | 2,5 |
| Custom Pattern Idx | 255 |
| Adaptive Divisions | 5 |
| Adaptive Rings | 2 |
| Adaptive Depth | 1 |

# Smart Objects - Overview

- Smart Objects are **objects placed** in a level that **AI Agents** and **Players** can **interact** with. These objects contain all the information needed for those interactions.
- Smart Objects are part of a **global database** and use a **spatial partitioning structure**. This means that they can be **queried at runtime** by using filters such as location, proximity to the Agent, and tags.
- At a high level, Smart Objects represent a **set of activities** in the level that can be used through a **reservation system**. It's important to note that Smart Objects **do not contain execution logic**. Instead, they provide all necessary information to the **interactor** to be able to perform the interaction, depending on its implementation. Each interactor (Agent or Player) does its **own implementation** logic for a Smart Object.

- **SmartObject Subsystem**
    - The **SmartObject subsystem** is responsible for **keeping track** of all **Smart Objects available in the level**. This is the link between the **Smart Object components** and the **collection**. This subsystem is **automatically created** in the level when the **Smart Objects plugin is active**, and will create the Smart Object collection if it's missing.
- **SmartObject Collection**
    - The **SmartObject Collection** contains a list of all **SmartObject** components associated with the level. The collection works with **persistent levels** and must be **built manually** by the user.
    - This means that Smart Objects **registered with a collection** are kept **alive**, regardless of whether the Smart Object components are loaded or unloaded from the persistent level.

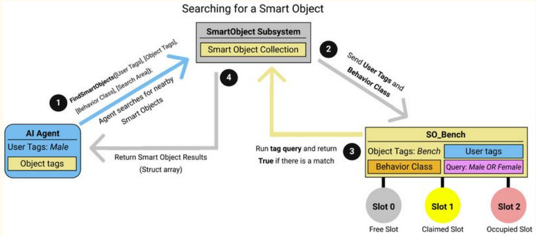
# Smart Objects - Overview

- **SmartObject Component**
    - The **SmartObject component** can be added to any **Actor** to mark it as a **Smart Object** in the level. The component points to a **Smart Object Definition asset**, which stores the **configuration** of a given **Smart Object template**.
    - The Actor containing the SmartObject component may be **loaded** and **unloaded** at runtime using **Streaming**. If the SmartObject component is included in the **persistent** world's Smart Object collection, a **runtime instance w**ill remain active in memory and will be considered as part of the simulation. If the Actor containing the SmartObject component is **spawned at runtime**, then it will not remain active in memory once it's unloaded.
- **Smart Object Definition**
    - A **Smart Object Definition** is a **data asset** that contains the **immutable data** shared between multiple Smart Object runtime instances. A Smart Object Definition stores **filtering information** such as **user-required tags**, **Activity tags**, **Object Activation tags**, and the default set of **Behavior Definitions** that could be used to interact with the Smart Object.
    - A Smart Object Definition **exposes one** or **more Slots** that can be used by **Agents** or **Players** for the specific Smart Object. Each Slot includes the **location** and **rotation** relative to the **parent anchor** (baked from editor placement), as well as several overridable properties. Common examples of overridable properties include user-required tags and specific Behavior Definition per Slot.
- **Smart Object Behaviors Definition**
    - **Smart Object Behaviors Definitions** contain the data needed by the **Agent** or **Player** for a **given interaction**. The following types of Behaviors Definitions are currently available:
        - **Mass Entity Behavior** - contains data used to configure Smart Objects that can be used by Mass Entities.
        - **Gameplay Behavior** - contains data used to configure Smart Objects that can be used by the Gameplay Behavior plugin.

# Smart Objects - Flow - Searching

- **Agent Data**
  - In order to **search** for Smart Objects, the Agent needs to have **one** or **more user tags**, and a list of **object tags**. This information will be used when **searching** for matching Smart Objects in the level.
- **Smart Object Data**
  - Smart Objects contain **one** or **more object tags** that are used to **describe** the object. They also contain a list of **user tags**, and a **tag query**. The tag query is an **expression** used to determine if the user requesting the use of the Smart Object is **allowed** to interact with it.
  - Smart Objects have a **Behavior Class** which contains the **Behavior Definition** that will be used by the **user** once they are **interacting** with the Smart Object.

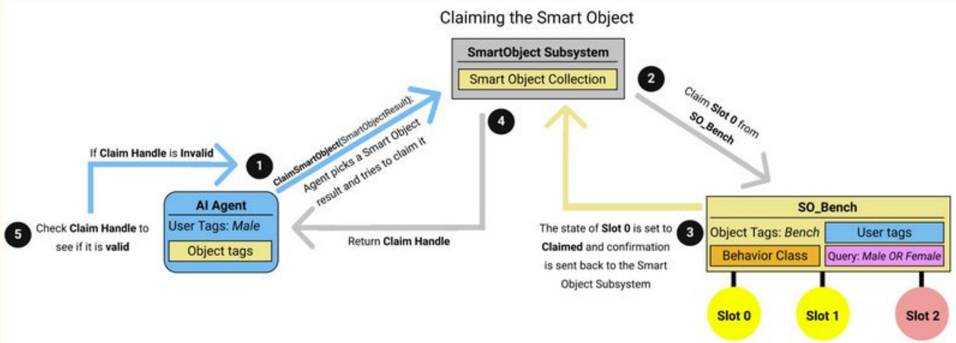
- **Searching for a Smart Object**
  1. The Agent **searches** for **nearby** Smart Objects on a **specified interval**. The Agent performs the search by calling the **FindSmartObjects** method in the **Smart Object subsystem**. This method contains the **user tags**, **object tags**, **Behavior Class**, and **search area**.
  2. The **Smart Object subsystem** finds all Smart Objects within the **search area** that match the object tags.
  3. Each Smart Object runs its **tag query using the user tags** to see if there is a match and sends the **results** to the Smart Object subsystem.
  4. The Smart Object subsystem returns the **Smart Object Results** to the **Agent**. The Smart Object Results are a **Struct array** containing all matching Smart Object IDs and their **free Slots**.




# Smart Objects - Flow - Claiming

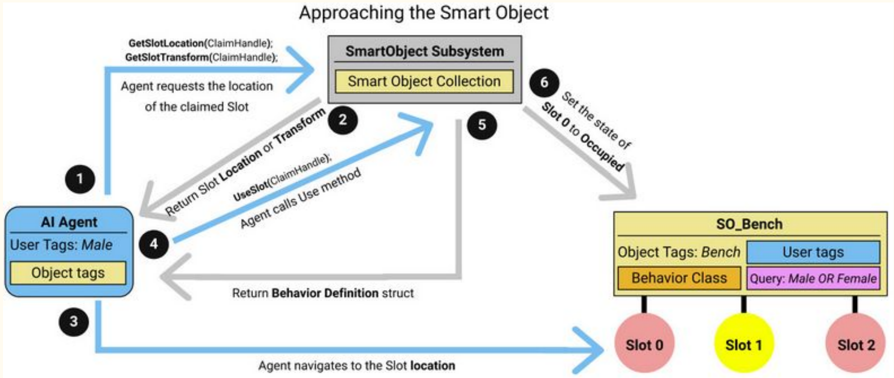
1. The Agent selects a desired **Smart Object Result** and calls the **ClaimSmartObject** method in the **Smart Object subsystem**. This method will **attempt** to **claim** an available Slot from the Smart Object.
2. The Smart Object subsystem **attempts** to **claim** an **available Slot** from the Smart Object.
3. An available Slot is **claimed** in the Smart Object and its **state** is set to **Claimed**. Confirmation is sent to the **Smart Object subsystem**.
4. Smart Object subsystem returns a **Claim Handle** to the **Agent**.
5. The Agent checks if the **Claim Handle is valid**. If it is valid, the claim attempt was **successful** and it can proceed to the **next step**. However, if the **Claim Handle is invalid**, the Agent may **attempt** to claim **another Smart Object from** the Smart Object Results.

- **The claimed Slot may not be claimed by another Agent until it is released by the current Agent occupying the Slot.**




# Smart Objects - Flow - Approaching

1. The Agent calls the **GetSlotLocation** or **GetSlotTransform** method in the **Smart Object subsystem** and passes the **Claim Handle**. This method returns the **location** or **transform** of the claimed Slot.
2. The **Smart Object subsystem** returns the location or transform of the claimed Slot to the Agent.
3. The Agent can now start **navigating** to the Slot location in the Level. The Agent can use any **desired navigation method** to reach its destination.
4. The Agent arrives at the **Slot location** and calls the **UseSlot** method in the **Smart Object subsystem** and passes the **Claim Handle**.
5. The Smart Object subsystem returns the **Behavior Definition** struct to the **Agent**. The Behavior Definition contains all the **required data** for the Agent to **perform** its desired behavior at the **Slot location**.
6. The **UseSlot** method triggers a **state change** for the **claimed Slot**. The Slot's state is changed from **Claimed** to **Occupied**.




# Smart Objects - Flow - Releasing

1. The Agent **performs** the **desired behavior** described in the Behavior Definition.
2. Once the **behavior** is **completed** or **aborted**, the Agent calls the **ReleaseSmartObject** method with the **Claim Handle** in the **Smart Object subsystem.**
3. The Smart Object Subsystem changes the **Slot state** from **Occupied** to **Free**.
4. The Agent is now **free** to perform **another behavior** or search for **another** Smart Object.

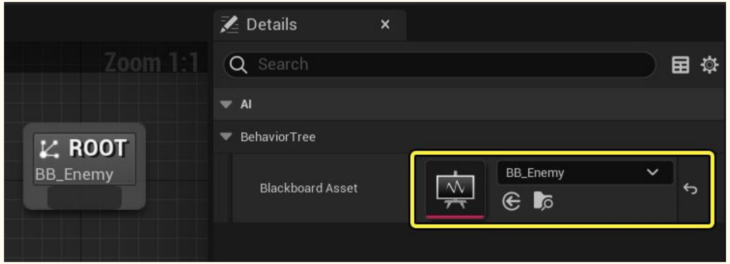
- **Agents** are **responsible** for releasing their **claimed Slots**. This can happen once their behavior is **completed** or **interrupted**.

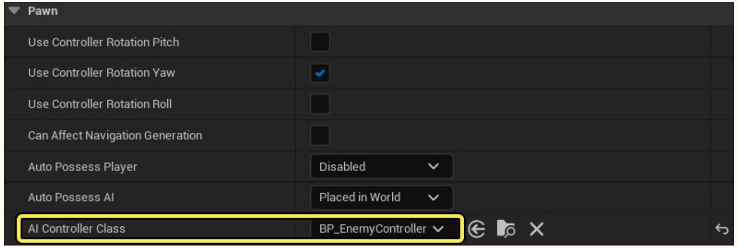
# Smart Objects - Flow - Aborting

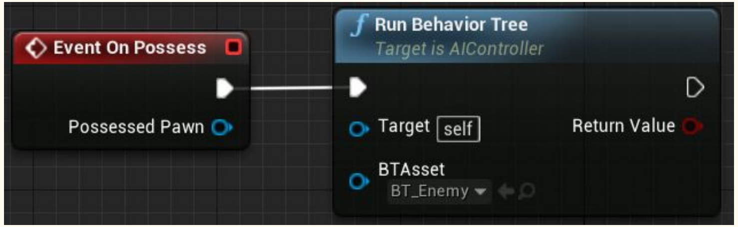
- The process described above can be **interrupted** or **aborted** by the **Agent** or the **Smart Object** at any time.
- If the Smart Object's **state changes** it will automatically **release** all **Claimed** or **Occupied Slots** and will notify the corresponding Agents through the **OnSlotInvalidatedDelegate** callback. A common example is the Smart Object being **destroyed** during gameplay.
- The Agent can also abort the process at any time for any reason. In this scenario, the Agent is responsible for **releasing** the **Slot** so other **Agents** can claim it. Common examples include the Agent **dying** or performing **another task** with a **higher** priority.

# Behavior Trees - Basics

- **Behavior Trees** are created in a **visually like Blueprint** in node based that have some functionality attached to them to a Behavior Tree Graph. While a **Behavior Tree executes logic**, a separate **asset** called a **Blackboard** is used to **store information** (called Blackboard Keys) the Behavior Tree needs to know about to make **informed decisions.**
- A typical workflow would be to create a **Blackboard**, add some **Blackboard Keys**, then create a **Behavior Tree** that uses the **Blackboard asset** (pictured below, a Blackboard is assigned to a Behavior Tree).
- In the **Behavior Tree detail panel**, you can **select** which **blackboard asset** needs to be assigned to it
- **Behavior Trees** in Unreal Engine **execute** their logic from **left-to-right**, and **top-to-bottom**. The **numerical** order of operation can be viewed in the **upper-right corner** of nodes placed in the graph.
- On the image we can see, a **blue node** is referenced as a **Decorator**, also known as **Conditional** in other **Behavior Tree system**
- Once you have created your Behavior Tree and logic, you will need to **run** the **Behavior Tree** during **gameplay**. Usually, you will have a **Pawn** and that Pawn will have an **associated AI Controller** which will be used to take control of and direct the **Pawn** in **performing actions**. Below, we have assigned a custom AI Controller class to our Pawn.

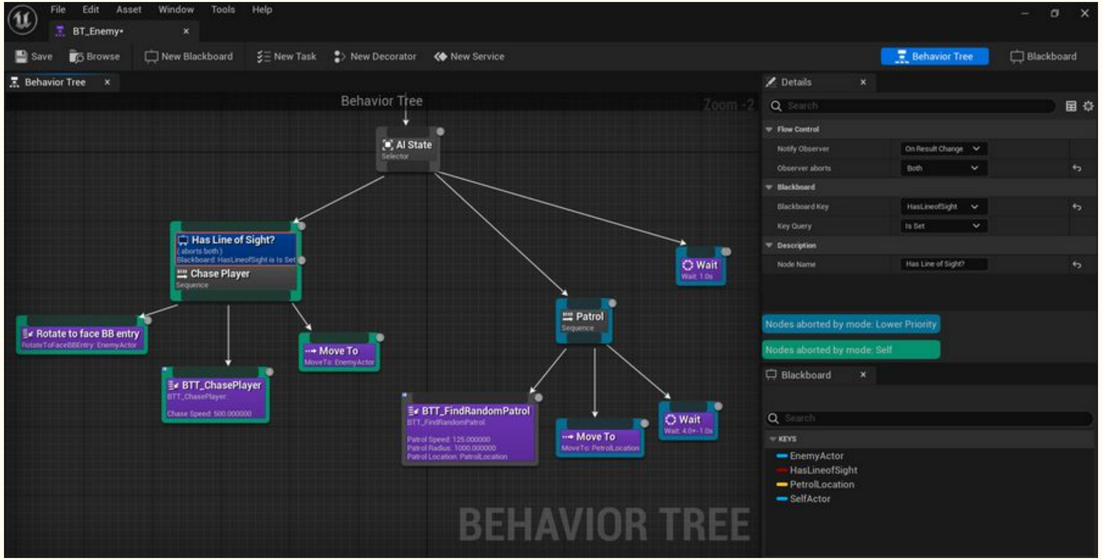


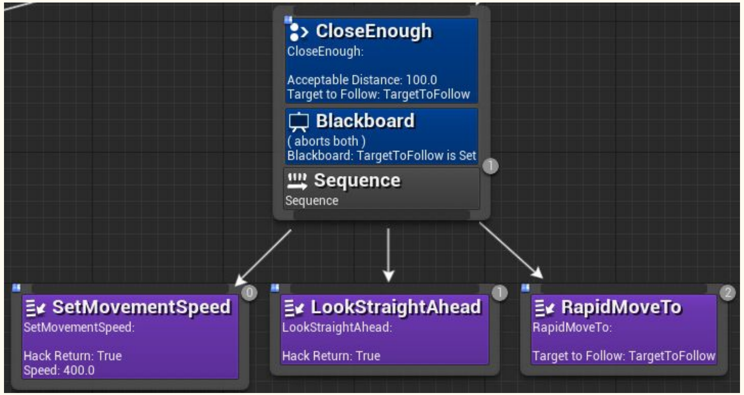
# Behavior Trees - Biggest differences - Event Driven

- One of the ways Unreal Engine Behavior Trees **differ** from other **Behavior Tree systems** is that Unreal Engine Behavior Trees are **event-driven** to avoid doing **unnecessary work every frame**. Instead of constantly checking whether any relevant change has occurred, the Behavior Tree **passively** listens for "**events**" that can be used to **trigger changes** in the tree.
- Having an **event-driven architecture** grants improvements to both **performance** and **debugging**.
- **Performance-wise**, there is not much explanation to give in order to understand why it is a great difference
    - You do **not waste loop cycle to iterate** over a tick function checking booleans etc...
- **Debugging-wise**, in you are iterating over **useless loops**, **useless values** which dictated how the behavior tree should direct the calls, it may become way more **complicated** to easily identify what you need to look at



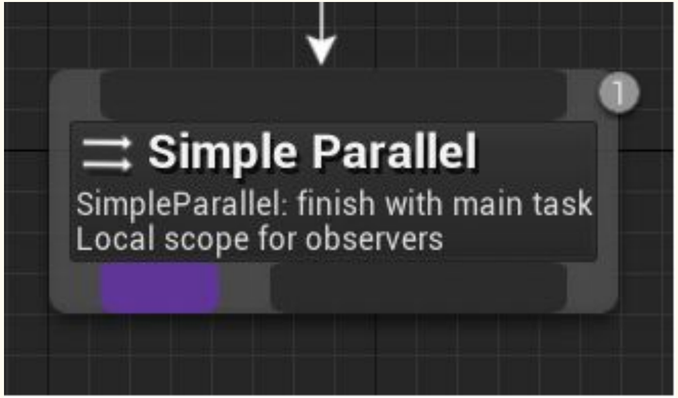
# Behavior Trees - Biggest differences - Conditionals

- In the **standard** model for **Behavior Trees**, **conditionals** are **Task leaf nodes**, which simply do not do anything other than **succeed** or **fail**.
- Although nothing prevents you from making traditional **conditional tasks**, it is highly recommended that you use **Decorators** for **conditionals** instead.
- Making **conditionals Decorators** rather than Tasks has a couple of **significant advantages**:
  - Conditional Decorators make the Behavior Tree UI more **intuitive** and **easier to read**.
  - Since all **leaves** are action **Tasks**, it is easier to see what **actual actions** are being ordered by the tree.
- Since **conditionals** are at the root of the **sub-tree** they are **controlling**, you can immediately see what part of the tree is "closed off" if the conditionals are not met.
- In the section of a Behavior Tree screen, the **Decorators Close Enough** and **Blackboard** can prevent the **execution** of the **Sequence** node's children. Another advantage of conditional Decorators is that it is **easy** to make those **Decorators act** as **observers** (waiting for events) at critical nodes in the tree. This feature is critical to gaining full advantage from the event-driven nature of the trees.



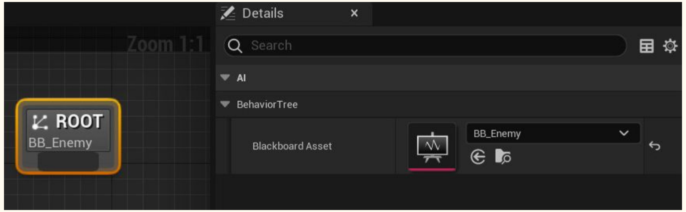
# Behavior Trees - Biggest differences - Concurrent Behaviors

- Standard Behavior trees often use a **parallel composite node** to handle concurrent behaviors and the **parallel node** begins execution on all of its **children simultaneously**. Special rules determine how to act if one or more of those child trees finish (depending on the desired behavior).
  - Parallel nodes are **not necessarily multi-threading** (executing tasks at the same time). They are just a way to **conceptually** perform **several tasks at once**. Often they still run on the same thread and begin in some sequence.
- Instead of **complex parallel nodes**, Unreal Engine Behavior Trees use **Simple Parallel nodes**, a special node type called **Services**, and the property **Observer Aborts** on **Decorators** to accomplish the same sorts of behaviors.
- **Simple Parallel nodes** have only **two children**
  - One which must be a **single Task node** (with **optional Decorators**), and the other of which can be a **complete sub-tree**. Think of the Simple Parallel node as "**While doing A, do B as well**." For example, "While attacking the enemy, move toward the enemy." A is a **primary task**, and B is a **secondary** or filler task while waiting for A to complete.
- While there are some options as to how to handle the **secondary task** (Task B), the node is **relatively simple** in concept compared to traditional parallel nodes. Nonetheless, it supports much of the **most common usage** of **parallel nodes**. Simple Parallel nodes allow easy usage of our **event-driven optimizations** while full parallel nodes would be **much more complex to optimize**.



# Behavior Trees - Nodes Instancing

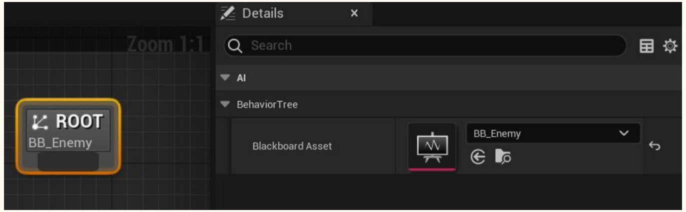
- **Behavior Tree Nodes** (referred to here as "nodes") exist as **shared objects**, meaning that **all agents** using the same **Behavior Tree** will share a **single set** of node instances. This improves **CPU performance while reducing memory usage**, but also prevents nodes from storing **agent-specific data**. However, for cases where agents need to **store** and update information related to a node, Unreal Engine provides three solutions:
  - **Instancing Nodes**
    - **bCreateNodeInstance variable**, when set to true, will grant **each agent** using the Behavior Tree a **unique instance** of the node, at the cost of some **performance** and **memory usage**. Classes like **UBTTask_BlueprintBase**, **UBTTask_PlayAnimation**, and **UBTTask_RunBehaviorDynamic**, use this feature.
  - **Storing on the blackboard**
    - A common solution is to **store variables** on the **Blackboard**. To do this, expose the name of the variable from your node, then fetch and store the **Blackboard Key** using that name during the node's initialization.
  - **Storing on the Behavior Tree node**
    - You can create a **custom struct** or **class** to store variables inside the node's memory
    - Many virtual functions in UBTNode take a **uint8* parameter**
    - Override version of **GetInstanceMemorySize**.
    - However, this memory is **not part of the UObject ecosystem**




```
struct FBTMoveToTaskMemory
{
    /** Move request ID */
    FAIRequestID MoveRequestID;

    FDelegateHandle BBObserverDelegateHandle;
    FVector PreviousGoalLocation;

    TWeakObjectPtr<UAITask_MoveTo> Task;

    uint8 bWaitingForPath : 1;
    uint8 bObserverCanFinishTask : 1;
};
```

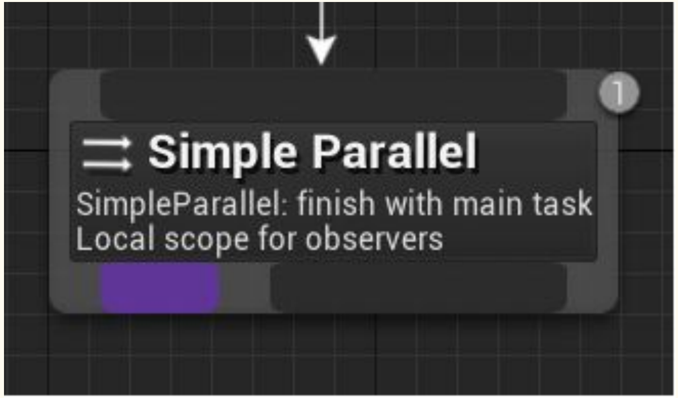
# Behavior Trees - Nodes

- **Behavior Tree Nodes** (base class **UBTNode**) perform the **main work** of Behavior Trees, including tasks, logic flow control, and data updates.
- The node that serves as the starting point for a Behavior Tree is a **Root node**. This is a unique node within the tree, and it has a **few special rules**.
    - It can have only **one connection**, and it does **not support attaching Decorator Nodes** or **Service Nodes**.
    - Although the Root node has **no properties** of its own, selecting it will show the **properties** of the **Behavior Tree** in the Details panel, where you can set the Behavior Tree's **Blackboard Asset**.
- Il existe 4 types de nodes
    - **Composite nodes** : These are the nodes that **define** the **root** of a **branch** and the **base rules** for **how** that branch is executed.
    - **Task nodes** : These are the **leaves** of the **Behavior Tree**, these nodes are the **actionable things** to do and don't have an **output connection**.
    - **Decorator nodes** : Also known as **conditionals**. These **attach** to **another node** and make **decisions** on whether or not a **branch** in the tree, or even a single node, can be **executed**.
    - **Service nodes** : These **attach** to **Composite** nodes and will execute at their **defined frequency** as long as their branch is being executed. These are often used to make **checks** and to **update** the **Blackboard**. These take the place of **traditional Parallel nodes** in other Behavior Tree systems.



# Behavior Trees -Composites

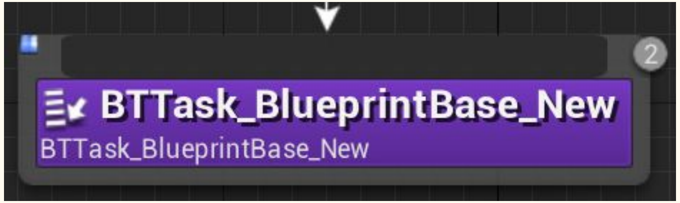
- **Composite nodes** define the **root** of a **branch** and the **base rules** for how that branch is **executed**. They can have **Decorators** applied to them to **modify entry** into their branch, or even **cancel out mid-execution**. Also, they can have **Services** attached to them that will only be **active** if the **children** of the **Composite** are being **executed**.
- Only **Composite nodes** can be **attached** to the Root node of a Behavior Tree.
- By default, there is 3 type of composite available
    - **Selector**
        - Selector nodes **execute** their children from **left to right.** They **stop** executing when **one** of their children **succeeds**. If a **Selector's child succeeds**, the **Selector succeeds**. If **all the Selector's children fail**, the Selector **fails**.
    - **Sequence**
        - **Sequence nodes** execute their children from **left to right**. They **stop** executing when one of their children **fails**. If a child fails, then the Sequence fails. If all the Sequence's children succeed, then the Sequence succeeds.
    - **Simple Parallel**
        - The Simple Parallel node allows a **single main Task** node to be **executed alongside** of a **full tree**. When the main Task finishes, the setting in Finish Mode dictates if the node should finish immediately, aborting the secondary tree, or if it should delay for the secondary tree to finish.



# Behavior Trees -Task

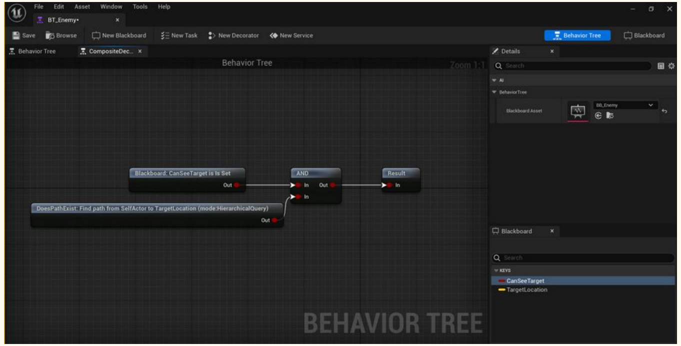
- Tasks are nodes that "do" things, like **move an AI**, or **adjust Blackboard values**. They can have **Decorators** or **Services** attached to them
- There is a **lot of task** by **default** in the engine, and the most works will comes from **you** but let's try to **highlight** some important one
    - **FinishWithResult**
        - The **Finish With Result Task** node can be **used** to **instantly** finish with a given result. This node can be used to **force** a **branch** to **exit** or **continue** based on the defined result.
    - **RunBehavior**
        - The **Run Behavior Task** enables you to **run another** Behavior Tree by pushing **sub-trees** onto the **execution stack**. One limitation to consider however is that the **subtree asset cannot be changed during runtime**. This limitation is caused by support for the subtree's Root-level Decorators, which are injected into the Parent tree. Also, the **structure** of the **running** tree **cannot be modified at runtime**.
    - **Wait**
        - The **Wait Task** can be used in the Behavior Tree to cause the **tree** to **wait** on this node until the **specified Wait Time** is complete.
- You can **create new Tasks** with your custom **Blueprint logic** and (or) **parameters** by clicking the **New Task button**.
- For a complete reference of existing task, check Unreal Documentation





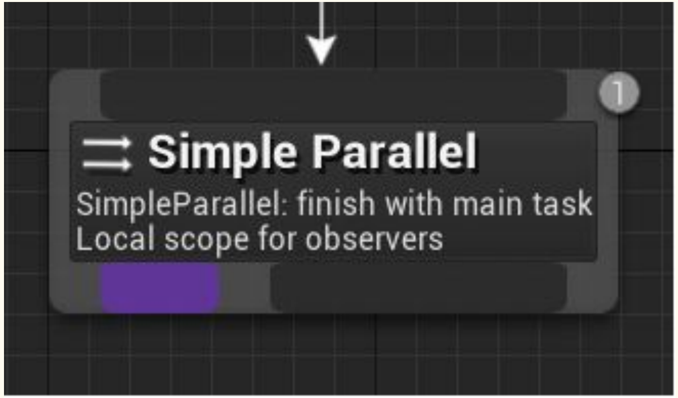
# Behavior Trees -Decorator

- **Decorator**, also known as **conditionals** in other **Behavior Tree systems**, are attached to either a **Composite** or a **Task** node and define **whether** or **not** a branch in the tree, or even a single node, **can** be **executed**.
- Again, there is a **lot of decorator** offers by **default** by the Engine, you can check the Unreal Documentation
- Let’s highlight some important one
    - **Blackboard** : The **Blackboard node** will check to see if a **value** is set on the given **Blackboard Key**.
    - **Composite** : The **Composite Decorator** node enables you to set up more **advanced logic** than the built-in nodes but **not as complex as a full Blueprint**. Once you have added a **Composite Decorator** to a node, **double-click** the Composite Decorator to bring up the **Composite's Graph**. By right-clicking in the graph area you can add **Decorator nodes as standalone nodes**, then wire them together through **AND nodes**, **OR nodes**, and **NOT nodes**, to create more **advanced logic**
    - **Conditional Loop** : As long as the **Key Query** condition is met, this **Decorator** will have the node it's attached to the loop.
    - **Custom Decorators** : You can create **new Decorators** with your **own** custom Blueprint logic and (or) parameters by clicking the **New Decorator button.**




# Behavior Trees -Services

- **Services** are **special nodes** associated with any **Composite node** (**Selector**, **Sequence**, or **Simple Parallel**) or **Tasks**, which can **register** for **callbacks** every specified amount of **seconds** and perform **updates** of various sorts that need to occur **periodically**.
- For example, a service can be used to determine which enemy is the **best choice** for the AI Pawn to pursue while the Pawn continues to act normally in its Behavior Tree toward its current enemy.
- Services **are active as long as execution remains in the sub-tree of the Composite node** where the service has been added
- These are often used to make **checks** and to **update** the **Blackboard**. These take the place of **traditional Parallel nodes** in other Behavior Tree systems
- Just like other nodes, you can create **Services** by clicking the **New Service button** from the **toolbar**.

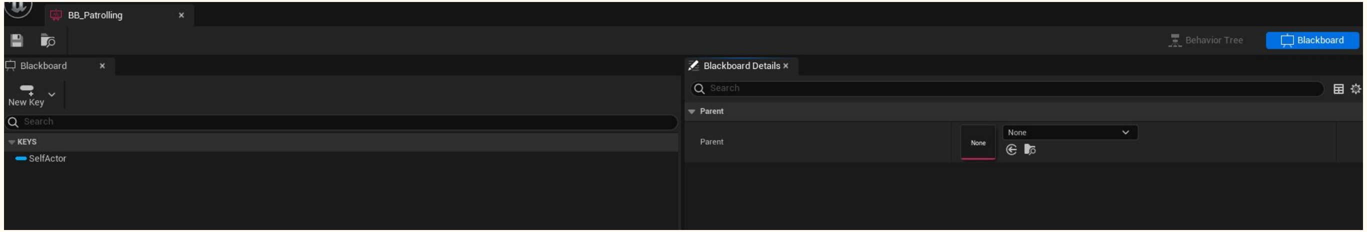


# Behavior Trees -Observer Aborts

- One **common usage** case for **standard parallel nodes** is to **constantly check conditions** so that a task can abort if the conditions it requires becomes false.
- For example, if you have a cat that performs a **sequence**, such as "**Hiss**" and "**Pounce**", you may want to **give up immediately** if the mouse **escapes** into its mouse hole.
  - With **parallel nodes**, you would have a **child** that checks if the **mouse** can be **pounced on**, and then **another child** that the sequence would **perform**.
  - Since Unreal Engine **Behavior Trees are event-driven**, we instead handle this by having our **conditional Decorators observe their values and abort** when necessary. In this example, you would have a "**Mouse Can Be Pounced On?" Decorator** on the Sequence, with "**Observer Aborts**" set to "**Self**".

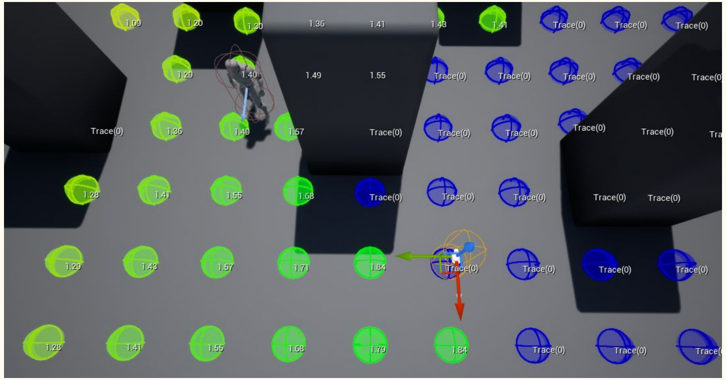
# Behavior Trees - Blackboard

- It is important to understand that a **BT** and more generally speaking **nodes** are using a **CDO** mode, which mean that you cannot save per-instance variable in them. There is some exception to handle that case
    - **Instancing Nodes**
        - The node's **bCreateNodeInstance** variable, when set to true, will grant each agent using the Behavior Tree a **unique instance** of the **node**, making it safe to store **agent-specific data** at the cost of some **performance** and **memory usage**. Some Unreal's node classes, including **UBTTask_BlueprintBase**, **UBTTask_PlayAnimation**, and **UBTTask_RunBehaviorDynamic**, use this feature.
    - **Storing on the Blackboard**
        - A common solution is to **store variables** on the **Blackboard**. To do this, expose the **name** of the **variable** from your node, then **fetch** and **store** the **Blackboard Key** using that name during the node's **initialization**. You can then use the Blackboard Key to **get** and **set** the variable's value on your agent's Blackboard instance.
    - **Storing on the Behavior Tree Node**
        - You can create a **custom struct** or class to **store variables** inside the **node's memory**. The **UBTTask_MoveTo** class, for example, uses **FBTMoveToTaskMemory**.
        - You'll need to override the **GetInstanceMemorySize()** function in order to provide the **new memory size of your struct**
        - It is the **less recommended** approach as it is not of the **UObject ecosystem**



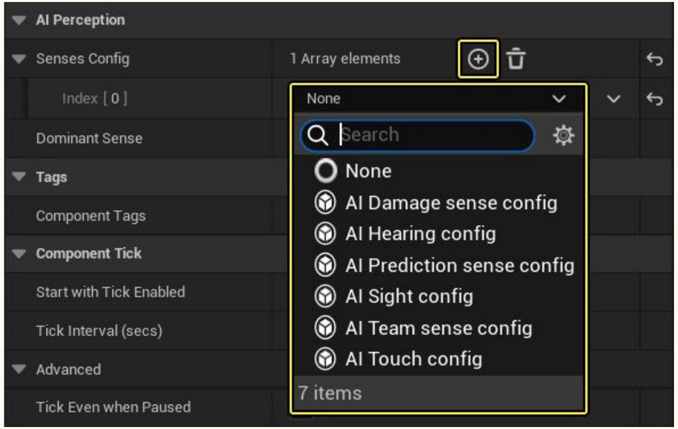
# Environment Query System (EQS)

- The **Environment Query System (EQS)** is a feature within the **Artificial Intelligence** system in Unreal Engine 5 (Unreal Engine) that is used to **collect data** from the **environment**. Within EQS, you can ask **questions** about the **data collected** through a variety of different **Tests** which produces an **Item** that best fits the **type** of **question** asked.
- An **EQS Query** can be called from a **Behavior Tree** and used to **make decisions** on how to **proceed** based on the **results** of your Tests. **EQS Queries** are primarily made up of
  - **Generators,** which are used to produce the **locations** or **Actors** that will be **tested** and **weighted**
  - **Contexts,** which are used as a frame of reference for any Tests or Generators.
- **EQS Queries** can be used to **instruct AI characters** to find the best **possible location** that will provide a **line of sight** to a player to **attack**, the **nearest health** or ammo pickup, or where the **closest cover point** (among other possibilities).
- Once you have a general understanding of how **Behavior Trees** work in Unreal Engine and want to have your **AI query the environment.**
- **It is still an experimental implementation and therefore, EQS should not be used for production**



# AI Perception

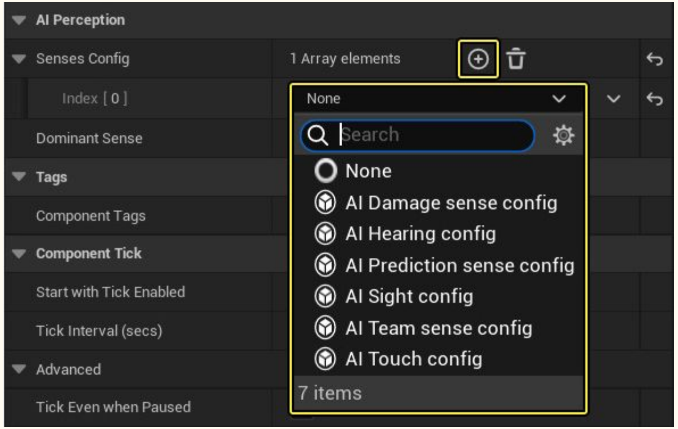
- In addition to **Behavior Trees** which can be used to **make decisions** on which logic to execute, and the **Environmental Query System (EQS)** used to **retrieve information** about the **environment**; another tool you can use within the **AI framework** which provides **sensory data** for an AI is the **AI Perception System**. This provides a way for Pawns to **receive data** from the environment, such as **where noises** are coming from, if the AI was **damaged** by something, or if the AI **sees** something. This is accomplished with the **AI Perception Component** which acts as a **stimuli listener** and gathers registered **Stimuli Sources**.
- When a **stimuli source** is **registered**, the event **On Perception Updated** (or On Target Perception Updated for target selection) is called which you can use to **fire off** a **new Blueprint Script** and (or) **update variables** that are used to **validate branches** in a **Behavior Tree**.



# AI Perception Component

- The **AI Perception Component** is a type of **Component** that can be added to a **Pawn's AIController Blueprint** from the Components window and is used to define what **senses** to listen for, the **parameters** for those senses, and **how** to respond when a sense has been detected. You can also use **several different functions** to get information about what was **sensed**, what **Actors** were sensed, or **even disable** or **enable** a particular type of sense.

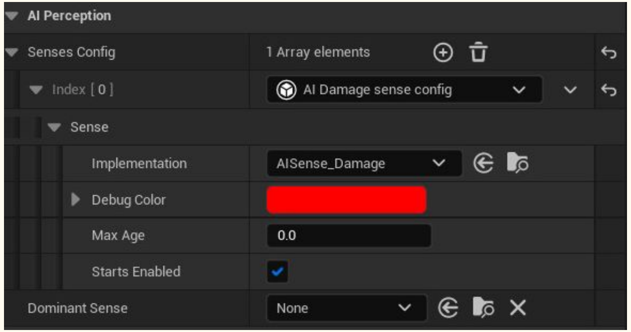
- In addition to the **common properties available** in the Details panel for the AI Perception Component, you can add the **type of Senses** to **perceive** under the **AI Perception** and **Senses Config section**. Depending on the **type of Sense**, **different properties** are available to adjust how the Sense is perceived.



# AI Perception Component - AI Damage Sense Config

- If you want your AI to **react** to **damage** events such as **Event Any Damage**, **Event Point Damage**, or **Event Radial Damage**, you can use the **AI Damage Sense Config**. The Implementation property (which **defaults** to the engine class **AISense_Damage**) can be used to **determine how damage events are handled**, however, you can **create** your damage classes through C++ **code**.

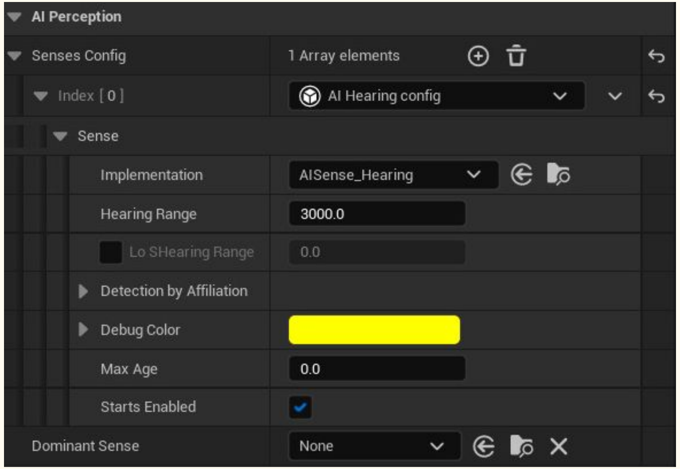
- **Implementation**
  - The AI Sense Class to use for these **entry** (defaults to **AISense_Damage**).
- **Debug Color :**
  - When using the **AI Debugging tools**, what color to draw the debug lines.
- **Max Age :**
  - Determines the **duration** in which the **stimuli** generated by this sense becomes **forgotten** (**0** means **never** forgotten).
- **Starts Enabled :**
  - Determines whether the given **sense starts** in an **enabled state** or must be **manually** enabled/disabled.



# AI Perception Component - AI Hearing

- The AI Hearing sense can be used to detect sounds generated by a Report Noise Event, for example, a projectile hits something and generates a sound that can be registered with the AI Hearing sense.

- **Implementation**
  - The AI Sense Class to use for this **entry** (defaults to **AISense_Hearing**).
- **Hearing range**
  - The **distance** in which this sense can be **perceived** by the AI Perception system.
- **Lo SHearing Range**
  - This is used to **display** a **different radius** in the **debugger** for Hearing Range.
- **Detection By Affiliation**
  - Determines if **Enemies**, **Neutrals**, or **Friendlies** can trigger this sense.
- **Debug Color**
  - When using the **AI Debugging tools**, what color to draw the debug lines.
- **Max Age**
  - Determines the **duration** in which the **stimuli** generated by this sense becomes **forgotten** (**0** means **never** forgotten).
- **Starts Enabled**
  - Determines whether the given **sense starts** in an **enabled state** or must be **manually** enabled/disabled.

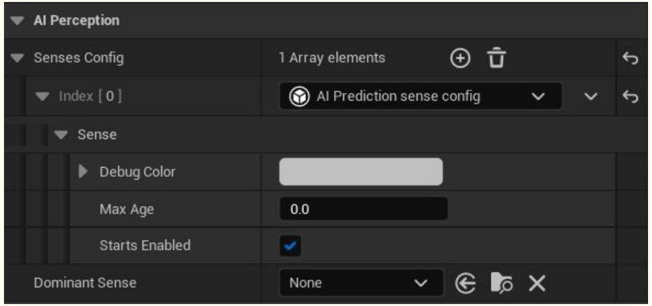


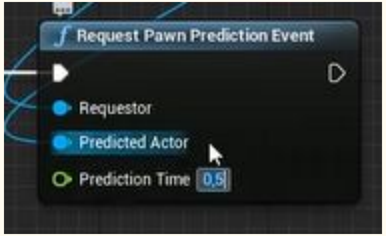
# AI Perception Component - AI Prediction

- This asks the Perception System to supply the **Requestor** with PredictedActor's predicted location in PredictionTime seconds.

- **Debug Color**
  - When using the **AI Debugging tools**, what color to draw the debug lines.
- **Max Age**
  - Determines the **duration** in which the **stimuli** generated by this sense becomes **forgotten** (**0** means **never** forgotten).
- **Starts Enabled**
  - Determines whether the given **sense starts** in an **enabled state** or must be **manually** enabled/disabled.

- A common use case for **AI Prediction** is when you want your AI to be **firing** by **predicting** how **far** your player will go if he **continue** to go into the same direction
  - From that point, you'll specify who is the **requestor**, who is the **predictor actor**, and from the **prediction time** you gave, it will feed accordingly the stimuli

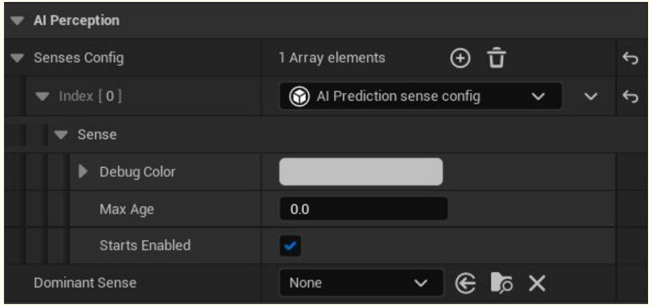


# AI Perception Component - AI Sight

- The **AI Sight config** enables you to define **parameters** that allow an **AI character** to "**see**" things in your Level. When an Actor enters the **Sight Radius**, the AI Perception System signals an **update** and passes through the Actor that was seen

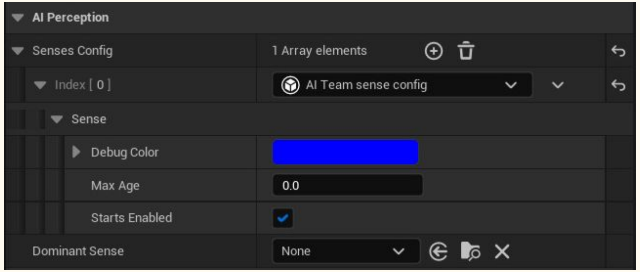
- **Implementation**
    - The AI Sense Class to use for this **entry** (defaults to **AISense_Sight**).
- **Sight Radius**
    - The **max distance** over which this sense can **start perceiving**.
- **Lose Sight Radius**
    - **The** max **distance** in which a seen target is no **longer perceived** by the sight sense.
- **Peripheral Vision Half Angle Degrees**
    - How far to the **side** the AI can see in **degrees**. The value represents the **angle** measured in **relation** to the **forward vector**, not the whole range.
- **Detection by Affiliation**
    - Determines if **Enemies**, **Neutrals**, or **Friendlies** can trigger this sense.
- **Auto Success Range from Last Seen Location**
    - When **greater than zero**, the AI will **always** be able to see the a **target** that has **already** been seen as long as they are **within** the range specified here.
- **Debug Color / Max Age / Starts Enabled**
    - Same as before



# AI Perception Component - AI Team

- This notifies the **Perception component** owner that **someone** on the **same team** is **close** by (radius is sent by the gameplay code which sends the event).

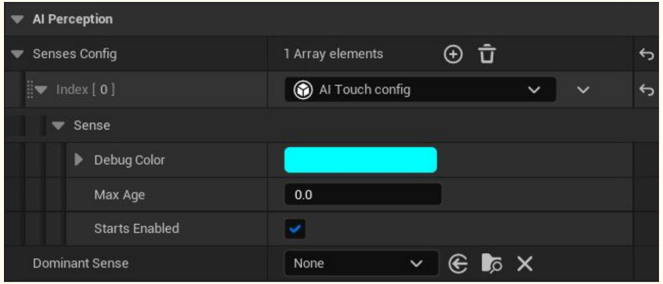
- **Debug Color / Max Age / Starts Enabled**
    - Same as before



# AI Perception Component - AI Touch

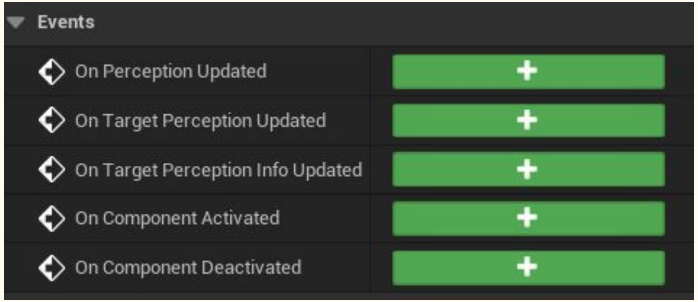
- The **AI Touch** config setting gives you the **ability** to **detect** when the AI **bumps** into something or something bumps into it. For example, in a **stealth based game**, you may want a Player to sneak by an enemy AI without touching them. Using this **Sense** you can determine when the Player **touches** the **AI** and can respond with **different logic**.

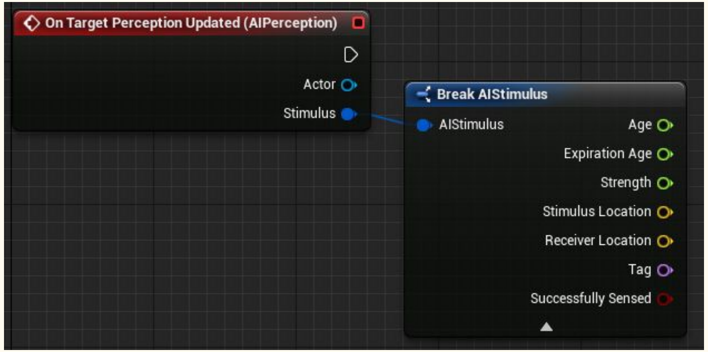
- **Debug Color / Max Age / Starts Enabled**
    - Same as before



# AI Perception Component - Events

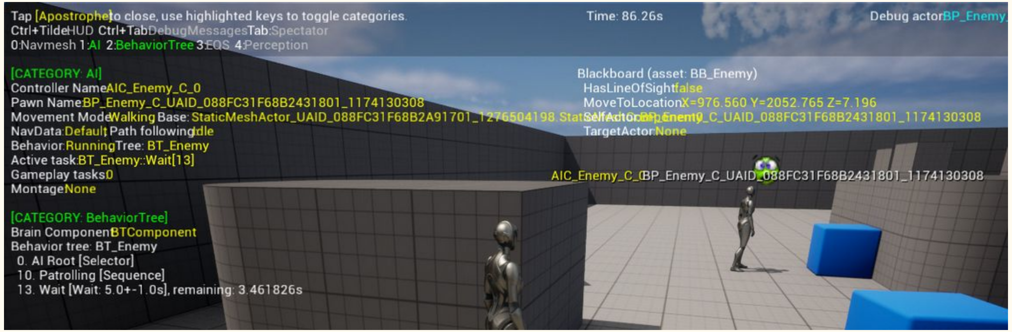
- When it comes to responds to perception update, you have 5 events
- **OnPerceptionUpdated**
  - This Event will fire when the **Perception System** receives an **update** and will return an **array of Actors** that signaled the update.
- **OnTargetPerceptionUpdated**
  - Notifies **all bound objects** that perception info has been **updated** for a given **target**. The notification is **broadcasted** for any **received stimulus** or on **change** of state according to the stimulus configuration.
- **OnTargetPerceptionInfoUpdated**
  - This Event will **fire** when the **Perception System receives an update** and will return the **Actor** that **signaled** the update. It also returns an **AI Stimulus** struct that can be **broken down** to retrieve additional information.
- **OnComponentActivated**
  - An **Event** that is fired when the **AI Perception Component** is activated.
- **OnComponentDeactivated**
  - An **Event** that is fired when the **AI Perception Component** is deactivated.



# AI Debugging

- Once you've created an AI entity you can **diagnose** problems or simply view what an **AI is doing** at any given moment using the **AI Debugging Tools**. Once enabled, you can **cycle** between viewing **Behavior Trees**, the **Environment Query System (EQS)**, and the **AI Perception systems** all within the same centralized location.
- To enable **AI Debugging**, while your game is running, press the **' (apostrophe)** key.
- When AI Debugging panel is visible, you can access **various informations** from within the text and using some **shortcuts**
    - **Numpad 0** : Toggles the display of the currently **available Nav Mesh data**.
    - **Numpad 1** : Toggles the display of the general **AI debug information**.
    - **Numpad 2** : Toggles the display of the **Behavior Tree debug information**.
    - **Numpad 3** : Toggles the display of the **EQS debug information**.
    - **Numpad 4** : Toggles the display of the **AI Perception debug information**.



# Time to.... highlight a concept

**Your turn !**

# Practice

- **General**
  - We'll only be practicing in general in order to get our hand in AI programming
  - Create a setup with navigation working with AI agent
    - Create different areas and place them into the work to affect the NavMesh you generated
    - Ensure to play with the values in order to understand how it makes the AI avoid some areas
    - Try to use different situation like forcing the AI to go into an area that it don't want
  - Create a Behavior Tree with everything related to it in order to make a patrolling behavior
    - Idle
    - Alert
    - Track
    - Attack