CS354p Lab 6: Artificial Intelligence

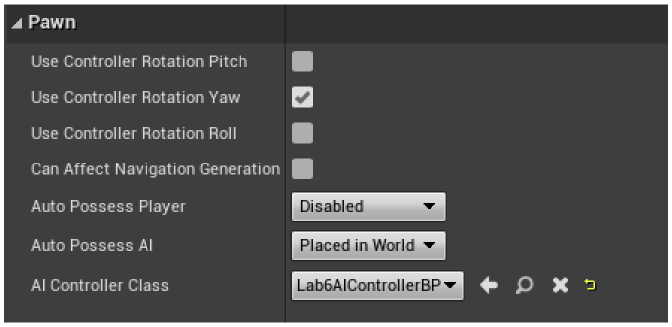
This lab will focus on creating the foundations for AI in C++. You will create an AI agent that moves between patrol points, building functionality primarily in C++ then extending it to Blueprint. This will also show the necessary setup to include AI functionality in your games.

This lab is based on this tutorial: `https://gameprogramming426359492.wordpress.com/2019/09/06/unreal-engine-4-c-artificial-intelligence/`, so you are welcome to look through this for additional information into Sensing Components, as we will only be creating the patrol aspect of the AI. I'd also recommend reading through the official documentation on Behavior Tree Quick Start as it contains a lot of the Blueprint level details and diagrams: `https://docs.unrealengine.com/5.2/en-US/behavior-tree-in-unreal-engine---quick-start-guide/`

Getting Started (Complete Before Class)

Create a new project named `Lab6` from a blank template building on C++. You can choose whether or not to include starter content, but you will need at least one static mesh. Go ahead and include the Mannequin assets in your Content folder, so you can see your AI agent more clearly (we will not be including animations at this time, though). Set up a map with a relatively large traversable area (OpenWorld is fine – just be sure to save it in the Content folder). We want to see our patrol agent move around, so it will need some space.

Create a C++ subclass that inherits from `ACharacter` called `Lab6AI`, and a C++ subclass that inherits from `AAIController` called `Lab6AIController`. Go ahead and create a constructor for `Lab6AIController`, but you don't need to implement anything else within them at this time. Create Blueprints for both classes, and drop in a Mannequin asset for your `Lab6AIBP` skeletal mesh. Set the `Lab6AIBP`'s AI Controller Class to `Lab6AIControllerBP` as well.



Pawn

Use Controller Rotation Pitch	☐
Use Controller Rotation Yaw	✓
Use Controller Rotation Roll	☐
Can Affect Navigation Generation	☐
Auto Possess Player	Disabled
Auto Possess AI	Placed in World
AI Controller Class	Lab6AIControllerBP

Make sure everything compiles and that's it! You're good to go!

Including the AI Modules

By default, UE5 projects do not include the necessary AI modules, so the first thing we need to do is update our `Lab6.Build.cs` script to include "AIModule" and "GameplayTasks" under `PublicDependencyModuleNames`. The `Lab6.Build.cs` script can be found under Source → Lab6. There are many other useful modules you may want to include in later projects, so keep this file in mind going forward. Also, if you start getting strange, cryptic linker error messages after adding some features, and nothing seems to be wrong with your code, double check that you don't need some additional modules included here.

Creating the Agent and Controller

We will now update our code for our AI agent (`Lab6AI`) and its controller (`Lab6AIController`) to support a Behavior Tree and Patrol Points for determining route. In `Lab6AI.h`, we just need to add a couple properties: 1) a BehaviorTree of type `UBehaviorTree`, and 2) a TArray of Patrol Points containing `AActors`. Including a TArray of Patrol Points will allow us to select Patrol Points from the map and include them on the routes of individual instances our AI agents. We don't actually need to do anything in `Lab6AI.cpp`, since the meat of our controls will be in `Lab6AIController`.

For `Lab6AIController`, we need to include a handful of properties. We need to create a `UBehaviorTreeComponent` and a `UBlackboardComponent`, but for our purposes we'll only need a getter for the `UBlackboardComponent`. We will also need an `int32` to store the `CurrentPatrolPoint`, which tells us which Patrol Point is our current target (this will need both a getter and a setter), as well as a `FName` to hold the key for the current Patrol Point target (`NextPointKey`) in the Blackboard, which is the AI agent's working memory of information. We will also want to create another TArray `PatrolPoints`, so we can pass the Patrol Points we selected for the AI agent to the Controller (which will communicate witth the Behavior Tree).

The last thing we'll need to add to our controller is an override of the function `OnPosses(APawn * InPawn)`. We will then need to add DefaultSubobjects via our `Lab6AIController` constructor as well as assignments for `NextPointKey` and `CurrentPatrolPoint`. `CurrentPatrolPoint` can default to 0, while we'll call `NextPointKey` "NextPoint" for our Blackboard value. We will come back to `OnPossess` in just a moment, but let's get some of the other stuff set up first.

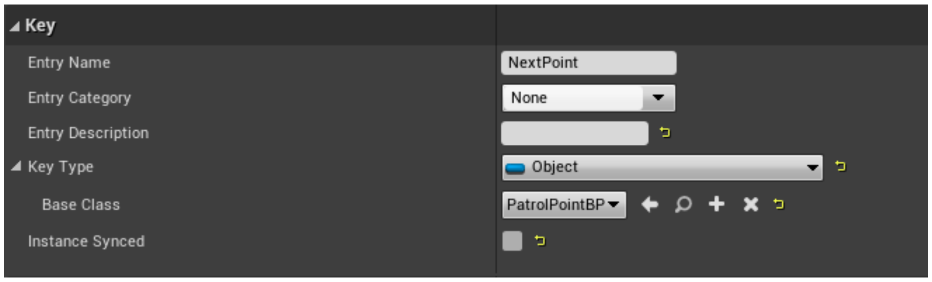
Creating Patrol Points

Before we get too far on the AI logic, we should finish getting the map set up with the actual Patrol Points and hooking them into `Lab6AI`. To do this, we'll create a subclass of `AActor` called `PatrolPoint`. Go ahead and create it in C++, but it's really just a shell class, so you basically are done with the C++ (we'll still make it in C++ in case we want to add more functionality later). Derive a Blueprint from `PatrolPoint`. We're going to add a static mesh here (and disable collision), so we can see it easily in editor, but the mesh will be hidden during gameplay. Thus the mesh you drop in can be pretty much anything. Just make sure under the "Rendering" category, it's set to "Hidden in Game."

We can now drop `PatrolPointBP`s into the level where ever we'd like our agent to patrol. Once these and a `Lab6AIBP` agent instance are placed in the scene, we'll add the Patrol Points to the agent's `PatrolPoints` TArray.

Creating a Blackboard and Behavior Tree

Now that we have all the pieces we need in the level, we can go ahead and create a Blackboard (right-click within the Content and look under "Artificial Intelligence." Create both a Blackboard called `PatrolBB` and a Behavior Tree called `PatrolBT`. Inside `PatrolBB`, create a new key called "NextPoint." Note that this name should match the FName you gave your NextPointKey, and the Key Type should be an Object of base class `PatrolPointBP`.



Key	
Entry Name	NextPoint
Entry Category	None
Entry Description	
Key Type	Object
Base Class	PatrolPointBP
Instance Synced	

Inside `PatrolBT`, make sure the Blackboard Asset is set to `PatrolBB`. We can also set the `Lab6AI` agent's Behavior Tree property (on the archetype — not the instance) to be `PatrolBT` at this time. We will need to do a little more work before we're ready to build out our Behavior Tree, so let's finish of `OnPossess` in `Lab6AIController` and create a Patrol Behavior Tree Task before we get back to this.

Controller OnPossess

`OnPossess` is where we'll actually start up the Behavior Tree. This function requires us to take in the `InPawn` parameter and cast it to `Lab6AI`. From here, we can verify the agent's BehaviorTree is set (what we just did via the Blueprint in the previous step), then initialize the Controller's BlackboardComponent using that data:

```
BlackboardComponent->InitializeBlackboard(*(AICharacter->BehaviorTree->BlackboardAsset));
```

Notice there are three pointers here, each of which could be nullptr, so be sure to check each before accessing the next. We need to assign initial values to both

`PatrolPoints` (pass in what's stored on the `Lab6AI` agent), and set the Blackboard Component's NextPointKey to be PatrolPoints[CurrentPatrolPoint]:

```
BlackboardComponent->SetValueAsObject(NextPointKey, PatrolPoints[CurrentPatrolPoint]);
```

Finally we're ready to start up the tree!

```
BehaviorComponent->StartTree(*AICharacter->BehaviorTree);
```

`OnPossess` will be called when the agent is placed in the world (as per our specification in the agent's BP), so the AI's "brain" should now turn on as soon as play begins.

Creating a BTTaskNode

Just one more piece of code to write and we're (nearly) there! We will create a C++ class that inherits from `UBTTaskNode` called `PatrolTaskNode`. This node will contain the functionality for updating which Patrol Point the AI agent is moving toward. We can then use this task within our Behavior Tree to update where the agent is going. The only function we need to create is `ExecuteTask`, which will contain the code that executes when the agent is updating which Patrol Point it's keeping in working memory. The function signature looks like this:

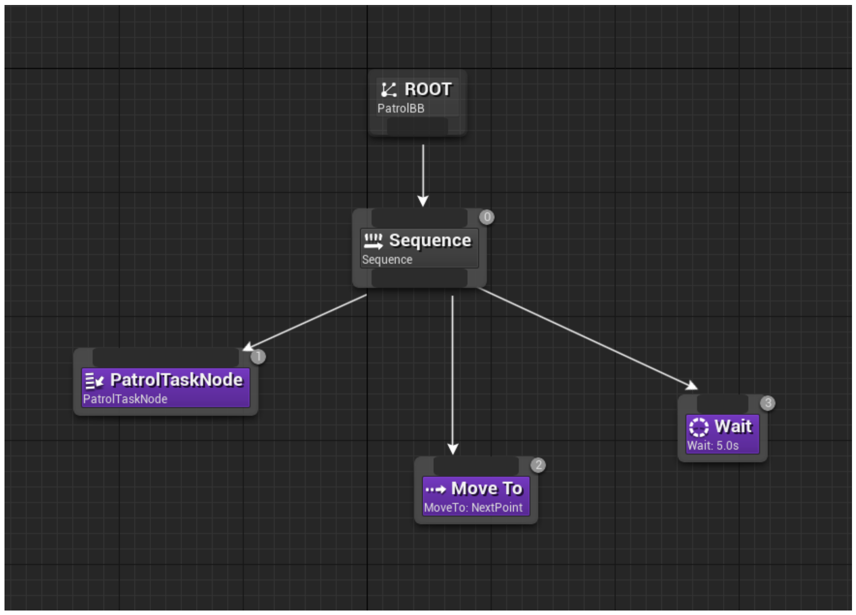
```
EBTNodeResult::Type ExecuteTask(UBehaviorTreeComponent & OwnerComponent,
uint8 * NodeMemory);
```

The `OwnerComponent` is a `UBehaviorTreeComponent` that conveniently contains the AIController. `OwnerComponent.GetAIOwner()` will retrieve an `AIController` which we can then cast to `Lab6AIController`. We will then access the BlackboardComponent, check which Patrol Point the AI agent is currently targeting (remember: `CurrentPatrolPoint` is our index into the `PatrolPoints` TArray), and update it to target the next one in the array (or loop back to the first one). We can now update both the `CurrentPatrolPoint` and the `NextPointKey` in the BlackboardComponent.

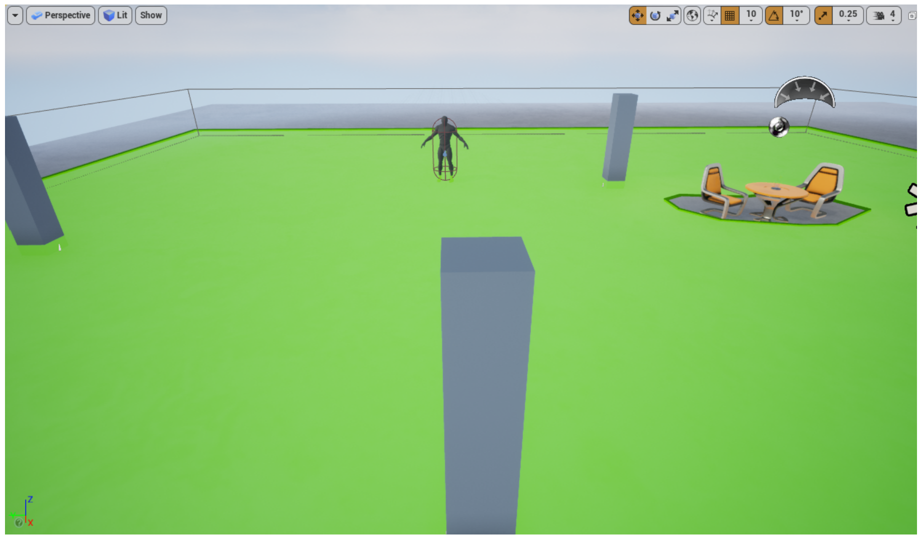
Again, this function will only be called when the agent is ready to move onto the next Patrol Point, so that's all the logic we need! Return `EBTNodeResult::Succeeded` if we can successfully execute this task, or `EBTNodeResult::Failed` if we cannot.

Defining the Behavior Tree

Back to Blueprints! Now it's just a matter of connecting our logic in the Behavior Tree itself. The sequence is: Figure out which Patrol Point to target, Move to that Patrol Point, then Wait 5 seconds. It will look something like this:



Yay! Practically done! Except for the NavMesh! Sadly your agent can't actually move without one, so you'll need to search through Placeable Assets and drop a Nav Mesh Bounds Volume onto the area your agent will need to patrol. Once you drop on in, you can see the bounds by pressing P in the editor:



As you can see, testing the code basically means completing the whole project before you see any results. In practice, it's better to do a little debugging between each step (printing to screen, dropping in break points to examine values, looking at the execution within the Behavior Tree execution etc). That said, if you get here and it's just not quite working, go back through each step and confirm there is a value where you need values, and the thing is being called as you'd expect, before looking in the next place. There are a lot of small details where things can go wrong, but once you have it, you're ready to create exciting behaviors and interactions in your own AI agents!

Submission

After you're satisfied, collect some video footage showing the system in action. Also screenshot exciting parts of your code, and submit these files plus the project code via your GitLab account and include a link to your video via Youtube. Link your repository as your Canvas submission.