Milestone 3 – Artificial Intelligence

This is an *INDIVIDUAL* assignment.

Due

Wednesday, October 18th at 11:55 PM

Late Policy

See milestone one for the late policy (2^(n+1) late policy).

Description

The goal for this assignment is to adapt your character controller into an AI-controlled non-player character (NPC). This NPC will demonstrate a set of behaviors appropriate for a game scenario.

You will create a never-ending gauntlet challenge for your NPC. It will involve a variety of navigation challenges such as reaching random stationary waypoints, intercepting moving GameObjects by predicting their future position, and throwing projectiles at (and successfully hitting) moving targets with a predictive algorithm.

General Requirements

You will be implementing an NPC via custom **Mecanim root motion steering behaviors**, an **AI state machine** (or behavior tree), and **A* path planning** (most likely provided by Unity's NavMesh Agent).

NPC Behavior Requirements
NPCs will demonstrate some combination of: reaching stationary waypoints, predictively chasing/tracking moving targets, and predictively throwing/shooting non-instantaneous hit projectiles at moving targets.

Projectiles
NPCs must demonstrate a projectile throwing ability that involves prediction of the future position of the target at the point of expected collision.

For simplicity, projectiles need not be affected by gravity. Such projectiles will follow a perfectly straight line when launched. These behaviors are meant to simplify the assignment (especially for projectile aiming at predicted future positions).

Implementation Requirements

You will build a custom state-based AI system with real-time root motion based steering and path planning capabilities. You will clone your existing player-controlled character (from M1/M2) and replace the controller script that interfaces with user inputs and instead send mecanim input commands dictated by your AI state machine. Note that this is non-trivial and an AI+Mecanim example is provided here: https://github.gatech.edu/IMTC/CS4455_MecanimTute

Alternatively, you can use other AI tools such as those in the list below or request approval for one of your favorite tools (if not listed). It is recommended that most students just make their own state machine though.

- Open Source NPBehave Unity plug-in (https://www.assetstore.unity3d.com/en/#!/content/75884)

  NPBehave builds on the powerful and flexible code-based approach to define behavior trees from the BehaviorLibrary and mixes in some of the concepts of Unreal's behavior trees. Unlike traditional behavior trees, event driven behavior trees do not need to be traversed from the root node again each frame. They stay in their current state and only continue to traverse when they actually need to.

  Note: This is a new library that has potential, doesn't have all the Unity Editor integration as RAIN AI and documentation is rather limited. However, it is open source can be extended. It's also unclear how easy it is to support root motion based movement out-of-the-box.

- Rival Theory's RAIN AI Unity plug-in (https://www.assetstore.unity3d.com/en/#!/content/23569)

  RAIN AI is powerful, but does have a bit of a learning curve. Officially, RAIN AI is no longer supported by Rival Theory, and some desirable components are deprecated or just completely broken. In fact, the last report we heard is that it no longer works out of the box with the latest Unity. The available documentation is also somewhat limited. A workaround for broken Mecanim support is available here but it was for Unity pre-version 2017: https://github.com/gtjeffw/RAINAIWithMecanim

  Note: While some CS4455/CS6457 students previously had success with RAIN AI, we've also had many that initially start with it but then ultimately abandon it due some of the difficulties working within its framework.

- Lastly, if you identify another behavior tree API and want to use it, just check with the instructor(s) for review and approval.

Visual and Audio Requirements

Your environment should not just be a large open field. It needs to provide a variety of paths and obstacles to demonstrate path planning. You might want to modify your game feel gardens with additional maze-like obstacles. The navigable surfaces can all be at the same elevation (to simplify projectile calculations). There are no specific graphic quality requirements for the scene other than it should be easy to distinguish where the various areas of the playing field begin and end.

A text description of the NPCs current AI state should be shown on the screen. This can be displayed as 3D text attached to the NPC or as an overlay on the HUD.

Additionally, you must tweak the various parameters of your AI and level such that the grader can observe all behaviors and outcomes within a reasonable period of time.

Itemized Requirements:

1) Your NPC must be controlled with an AI state machine with at least three (3) AI states and must be controlled by root motion of mecanim. This means your NPCs are mecanim-controlled animated/rigged meshes, same as previous assignments.

   For instance, navMeshAgent.updatePosition and navMeshAgent.updateRotation should be set to false. Slight position corrections are allowed in OnAnimatorMove(), but should be subtle enough not to notice any sliding.

   If using RAIN AI, NPBehave or another Behavior tree plugin, the states must be controlled via a behavior tree and both translation and turning via mecanim (RAIN: mecanim-based custom motor, NPBehave: script calling mecanim). The tree must have at least three (3) distinct branches representing specific strategies of play.

   The Steering AI finite state machine or Behavior Tree will have at least
   - 1 state leverages path planning (e.g. Unity NavMesh)
   - 1 state is visiting a stationary waypoint
   - 1 state is intercepting a moving GameObject (e.g. moving waypoint)
   - 1 state is a non-instantaneous projectile throw aimed with at a moving target with position prediction

   The AI should randomly select a goal (e.g. visit waypoint, throw projectile, etc.). Upon completion, another goal is randomly selected. The environment will automatically reset as necessary (e.g. respawn a chase GameObject if it was deleted from being touched).

   Note that no **production rules** implementations are allowed for this assignment.

   (20 points)

2) Your NPC uses Unity's NavMesh and built-in A* support to navigate to its desired destinations. (If using RAIN AI or another AI engine, use equivalent capabilities.)

   (15 points)

3) Successfully visits stationary waypoints via following a dynamically planned path (A*).

   (15 points)

4) Successfully intercepts moving GameObjects via a predictive extrapolation.

   (15 points)

5) The NPC has ability to launch a projectile at a target. The launch vector will be based on position prediction of the target (using position and velocity). The projectile itself must be animated with a velocity (e.g. ball flying through air) and not be an instantaneous hit (like a laser gun). Effects of gravity are not required.

   For partial credit, earn 10 out of 20 points for launching in the general direction of the target GameObject. Earn the additional 10 points for launching precisely at the predicted position and being able to regularly hit the target (linear extrapolation at least using constant velocity).

(20 points)

6) Informative sound effects are utilized to denote launching projectiles, hitting targets, reaching waypoints, etc.

   In addition, you will clearly display the current AI state of each NPC. Use visual debug output such as 3D follow text or HUD.

   (15 points)

Extra Credit Opportunities

Do **ONE** of the options below for **UP TO** 5 points. Do **TWO** of the options below for **UP TO** 10 points. Be sure your readme clearly documents that you have completed extra credit and want the grader to assess. Actual credit awarded is determined by the grader, specifically determining if the spirit of the extra credit task is met and other assignment requirements are met.

Advanced Projectile

Your AI's projectile is modified to be under the influence of gravity, but it is still launched so as to intercept the opponent's predicted positions. The projectile is thrown with a natural parabolic arc. The AI may adjust a variable (but bounded) shooting/throwing force and a variable launch angle. You may solve directly, or use an iterative approach. In either case, make sure that there is not a noticeable hit on frame rate during the calculation.

At least one moving target should be far enough away in the game world to require a throw near 45 degrees elevation to achieve a hit on the moving target.

Advanced Navigation

Implement Unity's OffMeshLink support coupled to your character's jump and falling ability. When traversing an OffMeshLink your character must use a physics-based jump and fall ability and not a scripted animation between the two OffMeshLink endpoints.

Tips

Don't confuse discussion of AI state machines with the Mecanim Animator state machine. State machines can be used for lots of things. There is one for the animation system, and you will be implementing a different one for your AI. These state machines may likely interact in some way, but are independent implementations.

General Projectile Tips:

You should be aware of the benefits of the atan**2**() function for calculating headings. https://en.wikipedia.org/wiki/Atan2

A Normal distribution can be used to add some realistic aiming error to your AI, should you need it for game tuning.

For tips on predictive aim, check out "Strategy #3 - Assuming Zero Acceleration" under http://www.gamasutra.com/blogs/KainShin/20090515/83954/Predictive_Aim_Mathematics_for_AI_Targeting.php

Other sections of the article give some tips on lobbed projectiles.

Moving Targets

It's easy to use the Mecanim Animator to create moving targets. Just create animations of GameObject transforms to move an object around.

General Strategy Tips

You may find it simplifies your logic to have designated spots to throw/shoot from, intermediate waypoints, etc. This is easier than analyzing 2D/3D regions of the map.

If using RAIN AI:

You will definitely want to first become familiar with general *behavior tree* principles: http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Also, check out the following project for help with Mecanim integration: https://github.com/gtjeffw/RAINAIWithMecanim

If using navmesh + custom state machine:

Don't forget to mark objects as "static" if you want the navmesh baking process to pick up on them. Also, objects need MeshRenderers and not just colliders for affecting the navmesh. Though you can use NavMeshObstacles to carve out areas.

Coupling Animation and Unity navmesh:
Specifically refer to "Animation Driven Character using Navigation"
https://docs.unity3d.com/Manual/nav-CouplingAnimationAndNavigation.html

Also, a NavMeshAgent+Mecanim demo is here (scene CS4455_AI_Demo):
https://github.gatech.edu/IMTC/CS4455_MecanimTute

Note that the above demo is designed to be configurable via Inspector settings and can be used with fairly minor changes for a variety of root motion based characters.

If implementing your own NavMeshAgent script from scratch:

Don't forget to set navMeshAgent.updatePosition and navMeshAgent.updateRotation to false since we want mecanim to control movement. Also, go ahead and implement OnAnimatorMove() with full Mecanim control of transform:

```
this.transform.position = anim.rootPosition; //npc only changes pos if mecanim says so
this.transform.rotation = anim.rootRotation; //npc only changes rotation if mecanim says so
```

Additionally add NavMeshAgent position resets to OnAnimatorMove():

agent.nextPosition = this.transform.position; //pull agent back if she went too far

Try changing the NavMeshController's default vehicle properties to match your root motion performance envelope as best you can. Also, configure the NavMeshController to match your capsule dimensions and other locomotion abilities. You can look at your individual animations in the Inspector for a summary of your average velocity and angular velocity. Use these values from your fastest running/turning animations for configuring your NavmeshController. You now need to analyze the NavMeshAgent's planned movements in (Fixed)Update() and map them to Mecanim inputs (hint: normalize velocity and angular velocity relative to the performance envelope to get in right form for mecanim inputs). Then start tweaking/filtering things slowly to reduce jittery movement. For instance, a scaling factor on turn angles is useful to deal with turning that is too aggressive. Finally, add filtering on your mecanim inputs using Lerp() to smooth out the last bit of jitter (but only do so once you have exhausted all other tweaks). Some filtering strategies are demonstrated in the github project above.

State Machine Implementation:
Test and develop one AI feature at a time, perhaps hard-coding your AI to stay in one state as you work on it.

If you need to slow down your AI for gameplay debugging, just put a cap on the maximum mecanim speed input passed from your steering calculations. For instance, if 1.0 is full speed then only allow a value of 0.8 for 80% of full speed. You can also adjust your NavMeshController top speed.

State machines can be implemented in a very simple manner. Consider an object-oriented approach: https://blog.playmedusa.com/a-finite-state-machine-in-c-for-unity3d/

Alternatively, you might consider a procedural approach similar to below:

```
public enum AIState
{
  Patrol,
  GoToAmmoDepot,
  AttackPlayerWithProjectile,
  InterceptPlayer,
  AttackPlayerWithMelee,
  ChasePlayer
  //TODO more? states…
};

public AIState aiState;

// Use this for initialization
void Start ()
{
  aiState = AIState.Patrol;
}

void Update ()
{
```

```
  //state transitions that can happen from any state might happen here
  //such as:
  //if(inView(enemy) && (ammoCount == 0) &&
  //  closeEnoughForMeleeAttack(enemy))
  //  aiState = AIState.AttackPlayerWithMelee;


  //Assess the current state, possibly deciding to change to a different state
  switch (aiState) {

    case AIState.Patrol:

    //if(ammoCount == 0)
    //    aiState = AIState.GoToAmmoDepot;
    //else
    //  SteerTo(nextWaypoint);

    break;

    case AIState.GoToAmmoDepot:

    //SteerToClosestAmmoDepot()

    break;

    //... TODO handle other states

    default:

    break;

  }

}
```

Submission:

You should submit a 7ZIP/ZIP file of your Unity project directory via t-square. **Please clean the project directory to remove unused assets, intermediate build files, etc., to minimize the file size and make it easier for the TA to understand.**

The submissions should follow these guidelines:

a) Your name should appear on the HUD of your game when it is running.
b) ZIP file name: <lastName_firstInitial>_mX.zip (X is milestone #)
c) A /build/ directory should contain a build of your game. Please make sure you preserve the data directory that accompanies the EXE (if submitting a Windows build)
d) Readme file should be in the top level directory: < lastName_firstInitial >_mX_readme.txt and should contain the following
    i. Full name, email, and TSquare account name
    ii. Detail which platform your executable build targets (Windows, OSX, GlaDOS, etc.) Also let us know if you implemented game controller support, and which one you used.
    iii. Specify which requirements you have completed, which are incomplete, and which are buggy (be specific)

iv. Detail any and all resources that were acquired outside of class and what they are being used for (e.g. “Asset Bundles downloaded from the Asset Store for double sided cutout shaders,” or “this file was found on the internet at link http://example.com/test and does the orbit camera tracking”). This also includes other students that helped you or that you helped.
v. Detail any special install instructions the grader will need to be aware of for building and running your code, including specifying whether your developed and tested on Windows or OSX
vi. Detail exact steps grader should take to demonstrate that your game meets assignment requirements.
vii. Which scene file is the main file that should be opened first in Unity

e) Complete Unity project (any file you acquired externally should be attributed with the appropriate source information)

Submission total: (**up to 20 points deducted** by grader if submission doesn’t meet submission format requirements)

Be sure to save a copy of the Unity project in the state that you submitted, in case we have any problems with grading (such as forgetting to submit a file we need).