Milestone 4 – Artificial Intelligence

General:

- **This is an *INDIVIDUAL* assignment.**
- **Use Unity version: <SEE SYLLABUS FOR REQUIRED VERSION NUMBER>**
- **Late Policy:** See Canvas Course site for late policy.

Description

The goal for this assignment is to modify a copy the Minion from CS4455_M1_Support (or your M1, M2, or M3) to be AI-controlled and run around a waypoint loop. You will use Unity's NavMeshAgent to control the steering behaviors. You will also implement a basic state machine to instruct the NavMeshAgent what to do.

The Minion will visit 5 different stationary waypoints spread across the extent of the scene (this must be visible in your scene). Then, the minion will use position prediction to intercept a sixth waypoint. This moving waypoint should be animated to move back and forth between two points using Mecanim animation.

Waypoints should be made from primitive GameObjects with the collider removed/disabled (e.g. a colored sphere or cube). This means that the meshRenderers are still present and enabled! We need to be able to see them.

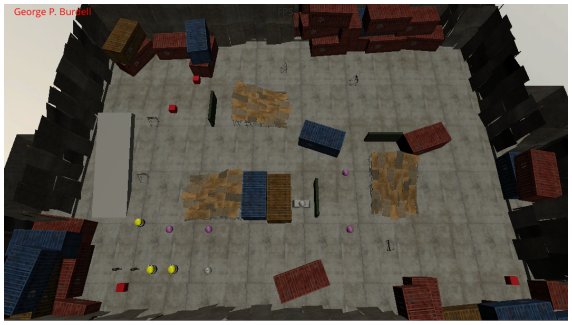
Steps:

(NOTE: You can start with a fresh clone of the starter project. If you don't, at least be sure to remove the pause behavior from M2. If starting from M3, be sure to change scenes included in build so that the M4 demo immediately starts.)

NavMeshAgent navigating through 5 static waypoints

First, you will configure a minion to use the NavMeshAgent to move around instead of keyboard control.

1.) Utilize a **fixed camera** looking down on your scene instead of the chase cam. It should look like this:



2.) Clone the Minion_noRootMotion and name it AI_Minion

3.) Remove all the components related to human character control (CharacterInputController, MinionBasicControlScript, MinionScript)

4.) Add a NavMeshAgent component to the AI_Minion

5.) Create a new script MinionAI

6.) Require and reference component UnityEngine.AI.NavMeshAgent

7.) Grab a reference to the Minion's Animator as well

8.) Add a public array of GameObjects to MinionAI named waypoints.

9.) Add an int currWaypoint property

10.) Create a private method setNextWaypoint(). This method should analyze currWaypoint and waypoints to increment currWaypoint and loop back to zero if currWaypoint is too big for the array. Additionally, the navMeshAgent should be updated of the new waypoint with SetDestination(). You will set to waypoints[currWaypoint].transform.position. Add error handling if the array is zero length.

11.) In Start(), call setNextWaypoint(); Make sure that currWaypoint is first initialized with a value that will cause setNextWaypoint() to go to the 0^{th} item. (-1 might work for you)

12.) In Update() also call setNextWaypoint(), but only if the navMeshAgent has reached the target waypoint. You can figure this out with navMeshAgent.remainingDistance, but you will also want to check that navMeshAgent.pathPending is ***not*** true so that being near a waypoint doesn't cause rapid iteration through the waypoints before the new path can be found. More complicated NavMeshAgent implementations may necessitate NavMeshAgent.hasPath and NavMeshAgent.pathStatus.

13.) In the editor, create 5 waypoints out of GameObjects with no colliders but keep them visible! You should be able to see them in both the editor and game view. Spread them out to make full use of the map. Read ahead about the moving waypoint if you want to plan to leave room for it.

14.) Populate your AI_Minion's waypoints (in Inspector) with these waypoint GameObjects (first manually increase the size of the array as appropriate)

15.) Using Unity's Navigation window, bake a NavMesh. Make sure you see blue navigable area over most of the scene.

16.) Confirm that your AI_Minion is in a blue navigable area and that he's not on top of any other characters.

17.) Confirm that all your waypoints are also navigable and reachable from each other and the Minion

Now you can run the game and you should see the minion translate around from waypoint to waypoint. He will likely be sunk into the ground and will not be making any step animations.

Next up, you will get the bouncy step animation working.

1.) Modify your AI_Minion's NavMeshAgent component "Base Offset" to 0.75. This will fix the issue with the model sinking into the ground.

2.) Modify your MinionAI script in Update() to tell the animator the forward animation parameter according to the navMeshAgent component's speed. It should look something like: anim.SetFloat("vely", agent.velocity.magnitude / agent.speed). Note that agent.speed is max speed.

Now test the game. You should see the minion hopping around and not sunk into the ground!

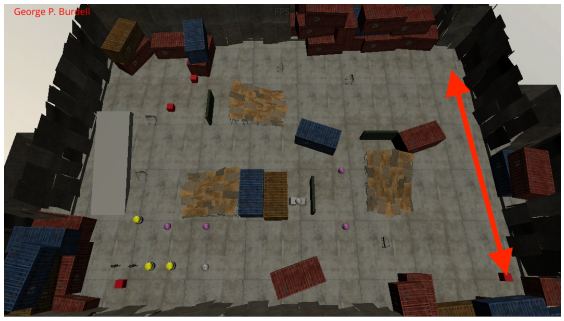
Intercept Moving Waypoint and State Machine

The last couple tasks are to:

1.) Convert your MinionAI script into a state machine. I recommend the "Procedural State Machine" for simplicity. See AI Tips section below.

2.) Add a moving waypoint that the Minion uses **prediction** to track down. This means that your minion needs to move towards a predicted intercept location for the moving waypoint, and not towards the waypoint itself. **No credit** will be assessed for solutions that set the location of the moving waypoint as the agent's destination.

To make the moving waypoint, you can use Mecanim with an approach like the elevator from M2. However, adjust the animation curves so that the waypoint moves linearly without acceleration. Make sure the waypoint is always just above the navmesh (blue area) and not going over an obstacle or through a wall. See the screenshot below for the location that you should use:



(See the red arrow and the right side opposite the end with the ramp.)

Your state machine should have at least two states. One state can be for your agent navigating through your 5 stationary waypoints (**state0**). The second state can be for your one moving waypoint (**state1**). The sequence of states should loop forever (e.g. state0->state1->state0->state1->…). This means that your minion should do the following (note that steps 1-5 are all part of state0):

1. Move to Static Waypoint 0
2. Move to Static Waypoint 1
3. Move to Static Waypoint 2
4. Move to Static Waypoint 3
5. Move to Static Waypoint 4
6. Move to intercept the Moving Waypoint
7. Go back to step 1.

You may find the use of a state machine for this simple purpose a bit silly/overkill but focus on understanding the mechanics of state transitions so that it helps on your team projects.

Note that your AI may never completely reach the waypoint with default NavMeshAgent settings so you may need to relax the condition for the distance required to have success. Note that you want to measure distance to the actual moving waypoint and not measure distance to the predicted destination when determining a successful capture. Probably something like distToMovingWaypoint – someCaptureDistance for the moving waypoint successfully reached condition.

For prediction, implement a simple distance function to determine lookahead time (discussed in class and the basic algorithm is in the lecture notes). Then use the moving waypoint's velocity and position to extrapolate based on this look ahead time. You can use the following component script to get a velocity from the animated waypoint. Then you can use GetComponent() to get the VelocityReporter from a reference to the moving waypoint.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class VelocityReporter : MonoBehaviour {


    private Vector3 prevPos;

    public Vector3 rawVelocity
    {
        get;
        private set;
    }

    public Vector3 velocity
    {
        get;
        private set;
    }

    public float smoothingTimeFactor = 0.5f;

    private Vector3 smoothingParamVel;

    // Use this for initialization
    void Start () {

        prevPos = this.transform.position;

    }


    // Update is called once per frame
    void Update () {

        if(!Mathf.Approximately(Time.deltaTime,0f)){
            rawVelocity = (this.transform.position - prevPos) / Time.deltaTime;
            velocity = Vector3.SmoothDamp(velocity, rawVelocity, ref smoothingParamVel, smoothingTimeFac
tor);
        }
        else {
            rawVelocity = Vector3.zero;
            velocity = Vector3.zero;
        }

         prevPos = this.transform.position;

    }
}
```

Be careful extrapolating as you might end up with a predicted position off the NavMesh. You should limit your computed lookahead time value with Mathf.Clamp() and also use NavMesh.Raycast()(https://docs.unity3d.com/ScriptReference/AI.NavMesh.Raycast.html) to see if the waypoint would hit the edge of the NavMesh if the waypoint moved from its current position to its extrapolated position. If there is a hit, then either don't extrapolate or shorten the amount of extrapolation, perhaps just a bit before the raycast's collision point. For some applications, you might also use NavMesh.SamplePosition() (https://docs.unity3d.com/ScriptReference/AI.NavMesh.SamplePosition.html). But generally, NavMesh.Raycast() is probably the most useful of the two.

Dynamically place a tall, skinny box (destination tracker) at the location that your minion is predicting for the duration that it is seeking the moving waypoint. This box should not have a collider and should be visually distinct from your waypoint in both size and color and be able to be seen if it overlaps your moving waypoint. It's ok if this destination tracker shows up when the minion is moving towards static waypoints as well. The destination tracker should move as conditions change and clearly be ahead of the moving waypoint until the minion converges.

Make sure that your minion recalculates the predicted position every frame. Additionally, make sure that you are determining when the minion reaches the actual waypoint (within some small error bounds). You don't want the minion to assume reaching the predicted point is the same as reaching the actual waypoint.

Lastly, check out all the public properties of the navMeshAgent. Tweak speeds, accelerations, turn rates for best visual results, and to ensure that the entire loop can be observable in under 45 seconds. Make sure that your minion approaches the moving waypoint from very far away so that prediction can best be observed. Also, coordinate the NavMesh bake settings with your navMeshAgent dimensions.

Itemized Requirements:

1.) AI controlled Minion visits 5 stationary waypoints (that are visible) **(30 pts)**
2.) AI controlled Minion heads off moving waypoint (that is visible) and has a visualization of minion's predicted position (visible destination tracker object). The prediction must update every frame and the minion must reach the actual waypoint (not predicted position) to be successful. Visualization must be visible in built/executable version of project! **(30 pts)**
3.) AI is controlled by procedural state machine of at least 2 states where the minion visits 5 stationary waypoints, then heads off moving waypoint, then goes back to visiting 5 stationary waypoints, then heads off moving waypoint,….going on forever in that pattern. Ensure that the entire loop is completed in under 45 seconds. **(20 pts)**
4.) Minion is animated with steps and not sunk into ground **(20 pts)**
5.) Use the Auditor to test your build
6.) Make certain that your Build is properly set up for the grader to see the correct camera perspective and see the minion performing the appropriate AI behaviors. The TAs predominantly use your Build to determine correctness of runtime deliverables.

Submission:

You will submit a Zip/7Zip of your project via Canvas. If the file is too big for Canvas, then submit a link to a private cloud hosting (such as GT's Box license). **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. Refer to Assignment Packaging and Submission on the Canvas Syllabus for further details.**

The submissions should follow these guidelines:

a) Your name should appear on the HUD of your game when it is running.
b) Follow the Assignment Packaging and Submission steps including:
    i. ZIP file: <lastName_firstInitial>_m<milestone number>.zip
    ii. Complete Unity Project
    iii. Builds
    iv. Readme file should be in the top-level directory: <*lastName_firstInitial*>*_m*<*milestone number*>*_readme.txt* and should follow base requirements from *Assignment Packaging and Submission*
    v. Size reduction

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). Do not alter or remove your submission from cloud hosting until your grade has been returned.
AI Tips

State Machine Implementation:

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.

Procedural State Machine:

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;

  }

}
```

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.

You might also consider an object-oriented approach: https://blog.playmedusa.com/a-finite-state-machine-in-c-for-unity3d/

An advanced fully root motion NavMeshAgent example implementation is provided at:

This project demonstrates a much more complicated integration of the NavMeshAgent with root motion. It handles both translation and turning.

https://github.gatech.edu/IMTC/CS4455_MecanimTute

In addition to basic state machines, you may be interested in Behavior Trees:

- 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 the now defunct 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 NO LONGER AVAILABLE

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.

NavMesh Baking:

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.