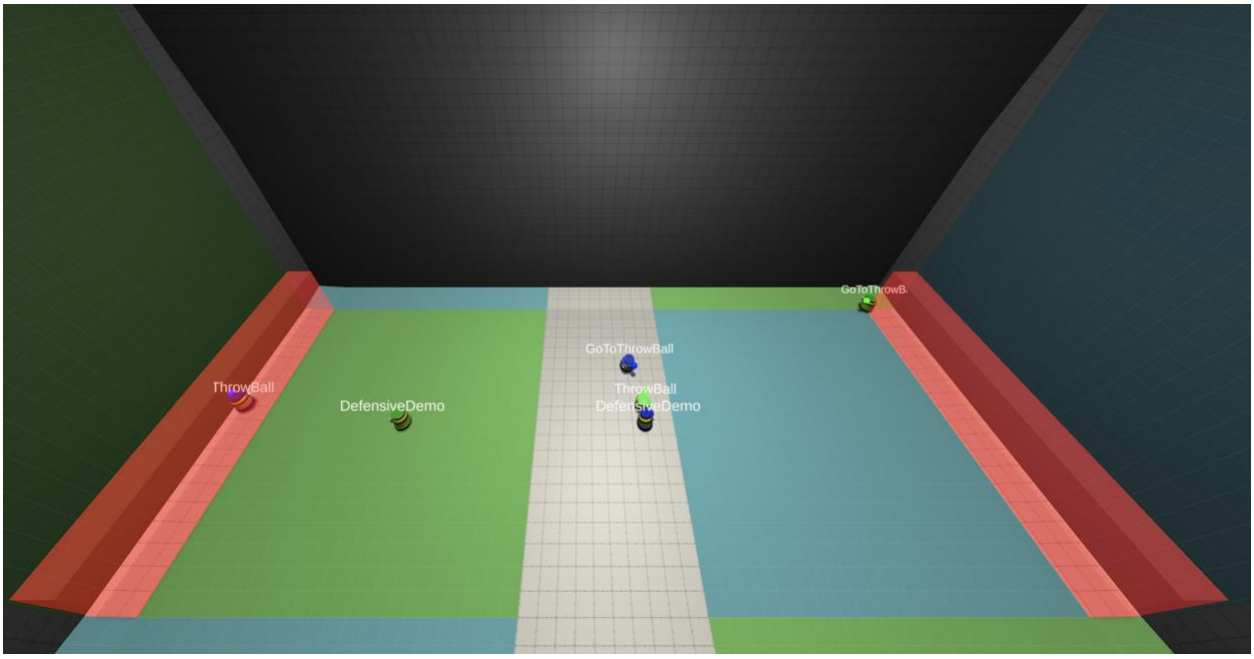
Homework 5 – Prison Dodgeball



In this assignment you will be implementing a Finite State Machine (FSM) to control each of a team of agents playing a game of Prison Dodgeball (description below). You will also be implementing prediction code for calculating the trajectory of a thrown dodgeball to intercept a moving target. Lastly, your FSM must be able to reliably beat the provided demonstration opponent to receive full points. There will also be a class tournament for those that wish to participate. This tournament will be optional but will involve an extra credit opportunity. Details of the tournament are yet to be determined but will likely start with a round robin qualifier by randomly assigned groups. The best of each group advances to a single elimination tournament bracket, seeded by round robin group points (3 points for win, 1 point for tie, no points for loss).

An example FSM is provided that is minimally effective, but does demonstrate most all of the method calls you might be interested in. You can use the code as a starting point, though we recommend starting a state machine from scratch (the individual states) and building up a bit at a time so you fully understand your code. You may also find it useful to remove capabilities from your opponent as you are testing so that you can better observe your code in action.

Your FSM should result in reasonable behavior for a video game. For instance, agents shouldn't get stuck standing still in an unresponsive state, glitch back and forth, etc. They can however be a bit goofy (e.g. two teammates going after the same ball, not throwing at what appears to be the best target, etc.).

Assessment

Grading will be based on the following aspects of your code:

1.) Implement a FSM compatible with the Prison Dodgeball Game
2.) Reliably beat opponent AI, “Glass Joe” with various team sizes (2-5) and number of balls (1-4). Your team should win at least 2/3rds of the time.
3.) Implement ballistic trajectory prediction, utilized in your FSM

Your ballistic trajectory prediction code will be tested in isolation from the dodgeball game, but your implementation is also important for good results in Prison Dodgeball gameplay.

Ballistic Trajectory Prediction

You can use any method for predicting including Millington’s static target method coupled with iterative refinement, Law of Cosines (LoC) with 10% Holdback, LoC with iterative refinement, or directly solve with a more advanced method (you will likely need to research quartic solving methods and implement).

In terms of difficulty of implementation, Millington’s static targeting and LoC are about equal in difficulty to implement. Adding iterative refinement to static targeting (to adjust for target movement) is only slightly more effort to code than LoC with 10% holdback and it gives better range. LoC iterative is arguably difficult to implement. So, if you are looking for a recommendation you might start with Millington plus iterative refinement.

Regarding assessment of your ballistic trajectory implementation, your code will be tested against static and constant velocity projectiles at roughly the same elevation. You will not be expected to accurately predict intercepts for targets that are initially unreachable but become reachable after some time, are moving at extreme speeds, or are at large elevation differences. Your implementation is expected to reliably determine whether a target is reachable or not (correct Boolean return value).

Prison Dodgeball

If you aren’t familiar with Prison Dodgeball, here is a description.

The game consists of two teams opposing one another. The team size is variable, but the number of members should be equal between the two teams. Each team has one side of a symmetrical playing area (e.g. a gym). Sometimes there is also a neutral area in the middle of the two sides. Both teams are allowed in the optional neutral area, but only members of a team are allowed in the team area that they are assigned. There are also two gutters along the long axis and to the sides of the playing area. This is so prisoners can exit the field of play to go to and from prisons at either end of the playing field. The prisons are at the far ends of the playing area.

The objective of the game is to throw dodgeballs at opponents to tag them (without a bounce on the playing field or walls). If an opponent is tagged, they must go to the prison behind their

opponents playing area. A player in prison must wait for a teammate to throw a ball to them for a rescue. The ball must be caught without bouncing first. Prisoners are free to move back and forth in the prison boundaries but cannot leave until rescued. Also, the prisoner gets to keep the ball. Prisoners are not supposed to interact with a neutral ball, but it's not uncommon to see kids throwing/kicking a neutral ball back to their non-prisoner teammates.

A rescued prisoner can immediately leave the prison but only by way of one of the gutters. The freed prisoner cannot be tagged while in the gutter until they pass the line that transitions from the opponents' area to the neutral area. If there is no neutral area, then use the transition line that divides the two team areas. Rescued prisoners that don't leave the prison/gutter in a timely fashion can be sent back to prison by the referee/gym instructor.

There are some number of dodgeballs as well. These are placed uniformly around the playing area and may appear in the neutral area or team areas.

Players have assigned starting locations. Often this is from the end line of each team's playing area. Players can also be from assigned individual starting positions. When play starts, players can rush to collect a ball if they choose or focus on evasion.

Gameplay progresses until one team has all players in prison. Once that happens, the other team wins. There is an optional time limit. If time runs out there is a tie.

Special circumstances

A dodgeball that bounces off an opponent and hits a second opponent without first touching the playing field tags both players out. This chain can go further as well. A dodge ball that hits a held dodgeball of an opponent (and no body part) is considered a deflection. In this case, the opponent does not go to jail.

In the normal version of prison dodgeball, a player that catches a throw from the opposing teams causes the thrower to be sent to jail.

Potentially, two teams can reach the point where there is one free player on either team. Furthermore, each could throw their ball at each other and simultaneously send each other to jail. If it is clear that one was tagged before the other, then there is clearly a game winning tag. However, if it cannot be determined the result is a tie.

A player that goes into the opposing team's side, the prison containing the opposing team prisoners, and possibly also the gutters is considered to be out and sent to prison. Balls stuck in an area that cannot be reached are dealt with in various ways. For instance, prisoners may be required to place errant balls on the line between the prison and the opponents' playing area.

Differences in Minion Prison Dodgeball

It's not possible to catch opponents' throws (unless there is a bug in the code). Minions play with a neutral area. Gutters and prisons have a sloped floor to return the balls. Minions play with a time limit (varies). Game rules block minions from entering an area that they are not allowed, so no penalties are necessary.

Rescued minions that refuse to leave the prison or gutter will be returned to prisoner status after 10 seconds without return to the neutral area.

Homework Resources

You will build upon the Unity project provided to you via git.

This project contains scene PrisonBall which you can open and play for a demonstration.

The file you will be working with is:
Assets/Scripts/GameAIStudentWork/MinionStateMachine.cs

This file includes a demo implementation of a state machine. You should replace the implementation of the states with your own. However, you will likely want to keep most of the MinionStateMachine implementation. This class inherits from FiniteStateMachine and does most of the work of running the individual states. You can instantiate and AddStates() to MinionStateMachine in the method Start().

Each State you implement has and Enter(), Exit(), and Update() method that you can implement. When the FSM runs a state, it first calls Enter(). Then it calls Update() until a transition occurs. Lastly Exit() is called after a transition. States can initiate a transition by returning a DeferredStateTransition to another state from Update(). If the state doesn't want to transition yet it should return null. A DeferredStateTransition is just an object that notifies the FSM what state you want to transition to, passes any parameters the new state needs, and optionally allows the transition to immediately update before the next simulation frame (default is to **not** immediately update).

There is a special global state to handle wildcard transitions. Also, there is a simple "TeamShare" implementation that allows for coordination between your team. It is somewhat like a Blackboard Architecture. It currently provides a way to track teammates. But it could also be extended by you to track responsibilities, avoid redundant behavior, etc. Be aware that with the TeamShare, you should always check it for null before accessing. Also, check any values you store in TeamShare for null as well before using them. In all cases of a null (or otherwise invalid data), have safe fallback behavior.

Each State has access to a variety of information available including the parent FSM (MinionFSM), the Minion that it is attached to, the game manager (Mgr), the assigned Team, and a TeamData object shared amongst the team. The Minion and Mgr will probably be used a lot in your states.

Here are useful features of these objects:

Minion

Tip: when referring to minion information. All the minions have the same abilities and dimensions. So, a minion can just reference itself for things like ThrowSpeed and Radius/Height. However, transform.position, Velocity, etc., are unique to specific minions.

transform.position/rotation – minion pose (centered on capsule collider centroid)
Radius – radius of minion (capsule collider)
Height – height of minion (capsule collider)
ThrowSpeed – speed in m/s of fastest throw minion can make
Velocity – current velocity on the navmesh
SpawnIndex – what order of spawning into simulation? Good for an arbitrary id relative to team
MaxAllowedOffAngleThrow – how far from directly facing target is allowed in ABS degrees?
TurnToFaceSpeedDegPerSec – how fast does FaceTowards() turn?
EvadeCoolDownTimeSec - how long (in sec) till another evasive action can be taken
HasBall – Does the minion have a ball?
DodgeballIndex – If a ball is held, the integer index into the DodgeballInfo of the PrisonDodgeballManager (see GetDodgeballInfo()). The value is -1 if a ball is not held
HeldBallPosition – Where is the ball for throwing start position?
IsPrisoner – Has minion been tagged and marked a prisoner?
TouchingPrison – is touching the prison area?
IsFreedPrisoner – Is minion freed but still walking back to playing area?
CanCollectBall – Can the minion pick up a ball?
CanBeRescued – Can the minion be rescued by throwing a ball to him?

DisplayText(string s) – Write text above minion's head. Helpful for debugging.
FaceTowards(Vector3 target) – Turn to face target while standing still
SignedAngleWith(Vector3 target) – Angle minion forward vec makes with vector to target
AbsAngleWith(Vector3 target) – ABS angle minion forward vec makes with vector to target
GoTo(Vector3 target) – Navigate to target
ReachedTarget() – Did agent reach target from GoTo() call?
Stop() – Stop following navmesh path
Evade(EvasionDirection ed, float strength) – take an evasive action with varying strengths (normalized scale)
ThrowBall(Vector3 unitVDir, float normSpeed) – Throw ball in direction with percentage of ThrowSpeed

NOTE: You may **not** use any methods marked INTERNAL_ (even if public). If there is something that is public but not mentioned above, just ask about it on Piazza.

PrisonDodgeBallManager

There are a variety of markers you can access for determining positions (gutter, middle of team area, etc.). Markers are just positions that you can use for path planning. You can create your own positions stored in Vector3 variables, possibly offset from the known markers.
TeamSize
BallsPerTeam
IsGameOver
IsTie
IsWinner(Team)
bool FindClosestNonPrisonerOpponentIndex(Vector3 myPos, Team myTeam, out int foundIndex) – Find a minion worth throwing at
bool GetOpponentInfo(Team myTeam, int index, out OpponentInfo oi) – Get details about opponent
public bool GetAllOpponentInfo(Team myTeam, ref OpponentInfo[] oppInfo)
public bool GetDodgeballInfo(Team myTeam, int ballIndex, out DodgeballInfo di,
    bool determineRegion)
public bool GetAllDodgeballInfo(Team myTeam, ref DodgeballInfo[] dodgeballInfo, bool determineRegion)

It is possible that you can access much more information (e.g., cheat), but please limit yourself to the public properties/methods not labeled INTERNAL_. We will screen for disallowed use of INTERNAL_ methods and also GetComponent<T>(), low level runtime access such as Reflection, and Unity game state managers. **If you want to use a particular data source but are unsure, just ask on Piazza.**

Throwing

You must implement *PredictThrow()* in your MinionStateMachine without modifying the method signature or changing its namespace. Refer to the lectures on Ballistic Projectile Trajectory solving to determine the method you prefer.

Tip: You can implement an incremental solver that itself calls a static position solver with the same method signature as below. This can be useful for testing because you can test that your static solver works before adding the iterative refinement for a moving target (see Testing section regarding the shooting range). If you find that you need a non-static method for throwing you can simply make a *MyMinionThrow()* (or similar) and have it call *PredictThrow()* as defined below. The most likely reason you will want to do this is if you want to consider live game state for shot selection.

// Returns TRUE if the throw is possible, FALSE otherwise **<- Make sure to implement this logic and DO NOT just always return true!**

```
public static bool PredictThrow(
    // The initial launch position of the projectile
```

Vector3 projectilePos,
// The maximum ballistic speed of the projectile
float maxProjectileSpeed,
// The gravity vector affecting the projectile (likely passed from Physics.gravity)
Vector3 projectileGravity,
// The initial position of the target
Vector3 targetInitPos,
// The constant velocity of the target (zero acceleration assumed)
Vector3 targetConstVel,
// The forward facing direction of the target. Possibly of use if the target
// velocity is zero
Vector3 targetForwardDir,
// For algorithms that approximate the solution, this sets a limit for how far
// the target and projectile can be from each other at the interceptT time
// and still count as a successful prediction
float maxAllowedErrorDist,
// Output param: The solved projectileDir for ballistic trajectory that intercepts target
out Vector3 projectileDir,
// Output param: The speed the projectile is launched at in projectileDir such that
// there is a collision with target. projectileSpeed must be <= maxProjectileSpeed
out float projectileSpeed,
// Output param: The time at which the projectile and target collide
out float interceptT,
// Output param: An alternate time at which the projectile and target collide
// Note that this is optional to use and does NOT coincide with the solved projectileDir
// and projectileSpeed. It is possibly useful to pass on to an incremental solver.
// It only exists to simplify compatibility with the ShootingRange
out float altT)

Testing

There are a few resources to aid with testing.

<u>Shooting Range Scene</u>

Open Assets/Scenes/ShootingRange.scene
You can test out your ThrowBall() implementation in a controlled environment.
The Assets/Scripts/ShootingRange/ShootingRange.cs code can be modified if you like (but you won't be submitting it)
The Shooting Range already has some keyboard presets for different types of motion, which you can extend/modify. Mode "4" is most similar to the Prison Dodgeball scenario. The Spacebar can be used to toggle between multiple shooting algorithms. **Make sure you are aware of whether your code is being tested. Check the name that appears.** More aim methods

can be added in Awake() provided that each method has the method signature defined above for PredictThrow(). You can reset stats with “r”.

For more extensive customization, refer to the files in Assets/Scripts/ShootingRange/ as well as the game objects in the scene.

Unit/Integration Testing

You can find unit/integration tests under Assets/Scripts/GameAIStudentWork/EditorModeTests and Assets/Scripts/GameAIStudentWork/PlayModeTests. For editor mode, there is an example of calling PredictThrow. For play mode, there is a test that can pit two agents head-to-head over an arbitrary number of matches.

Note that you must select either the PlayMode or the EditorMode tab in the Unity Test Runner to access each test group.

Submission

Submit file *MinionStateMachine.cs*

- Be sure to change the name string
- **Remove ALL debug print statements for more efficient code**
    - **TEST** your code after you remove print statements!
- Only submit the one source file

Tournament entry will be a separate submission. Final tournament details are TBA.