Programming Assignment 1
Detailed Instructions

Overview

In this programming assignment, you're building the Ted the RoboCollector "game".

Step 1: Explore the Unity project

You should use the Unity project I provided in the zip file as the starting point for your solution to this programming assignment. Open the project in Unity and explore the contents of the Unity editor and the scripts in the solution to make sure you understand everything before you start changing the project.

The `Collector` class has lots of code commented out. I commented that code out so the solution would compile but I left it in the code so you could use what you want as you implement your solution.

When you run the game, you should see pickups spawning periodically, with the teddy bear just sitting there.

Step 2: Use pickup spawned event

For this step, you're using the `EventManager` to connect `PickupSpawnedEvent` invokers and listeners.

Add the `PickupSpawner` as an invoker of the event and add the `Collector` as a listener for the event. You'll need to add a delegate in the `Collector` class to listen for the event. For now, just have that delegate print a message to the Console window when it runs.

When you run the game, you should see the message in the Console window every time a new pickup is spawned.

Step 3: Add sorted list of targets

For this step, you're adding targets to the sorted list as they're spawned.

Implement the `CompareTo` method in the `Target` class. You may find the code for the Implementing an Interface lecture in the Dynamic Arrays lesson or the solution to Exercise 2 helpful as you do this. You should make sure `Target` objects will get sorted in descending order based on their distance field. This will put the closest pickup at the end of the list.

Implement the `Add` method in the `SortedList` class. The `SortedList` class uses a `List` instead of an array to hold the items in the sorted list, but the `List` class doesn't let us add an item to a particular location in the list; we can only add to the end of the list. You can use the `tempList`

field I provided to temporarily hold parts of the list until you get the new item added in the right place. You may find the `Add` method in the `OrderedDynamicArray` class in the code for the Dynamic Arrays reading helpful as you figure out where the new item should go in the list. The `List` `Add` and `AddRange` methods are helpful as you build the list containing the new item. Remember, the `Add` method needs to implement an O(n) operation when you don't need to increase the capacity of the list. Analyze your code and read the `List` `Add` and `AddRange` documentation carefully to make sure you're getting the correct complexity.

In the `Collector` class, I've provided a `SortedList` field to hold the targets for the collector and I've called the constructor to initialize the field. In the method that listens for the pickup spawned event, create a new `Target` object for the new pickup and add the new object to the sorted list field. Change the message you print to the Console window to print the distance to the last pickup in the sorted list.

When you run the game, you should see the message in the Console window every time a new pickup is spawned. The distance in the message should always be the distance to the closest pickup (which may not be the newest pickup).

Step 4: Set target pickup to closest pickup when a new pickup is spawned

For this step, you're making sure the target pickup for the collector is always set to the closest pickup in the game.

In the `Collector` class, uncomment the `SetTarget` method and modify it to work properly. Comment out the call to the `GoToTarget` method, because we're not going to actually start moving to the target yet.

You should note that the `targetPickup` field I provided is a `Target` object, not a `GameObject` object. You should leave it that way because when we reach the target pickup and need to destroy it, we need to find it in our list of targets. We can only use the `IndexOf` method in the `SortedList` class on `Target` objects, not `GameObject` objects.

Now you need to implement the body of the method in the `Collector` class that you said would listen for the `PickupSpawnedEvent` . From Step 2 above, that method just prints a message to the Console window when it runs at this point; remove the code that prints the message to the Console window. Add code to the body of the method as discussed below.

You always need to add a new target for the new pickup to the list of targets, but the only time you want to change the target pickup by calling the `SetTarget` method is when the distance to the last target in the list of targets is less than the current distance between the collector and the current target pickup. If you think about this, you'll realize that also means that the new pickup ended up as the closest pickup in the list of targets. Be careful when the first pickup is spawned because the target pickup is null in that case.

Don't actually start the collector moving toward the target in the `SetTarget` method; instead, print a message to the Console window providing the distance to the new target pickup.

When you run the game, you should see the message in the Console window every time a new pickup is spawned. The distance in the message should always be the distance to the closest pickup (which may not be the newest pickup). You should only see the message from the `SetTarget` method when a new closest pickup has been added to the game.

Step 5: Move collector toward target pickup

For this step, you're moving the collector toward the target pickup.

In the `Collector` class, uncomment the `GoToTargetPickup` method. Change the `SetTarget` method to call the `GoToTargetPickup` method instead of printing the message to the Console window.

When you run the game, the collector should move toward the first pickup that's spawned into the game, then stop when it collides with that pickup.

Step 6: Collect pickup and move to next one

For this step, you're making it so the collector destroys the target pickup and moves to the next one.

In the `SortedList` class, implement the `RemoveAt` method. Remember, the `RemoveAt` method needs to implement an O(1) operation under the assumption that you're removing the last item in the list. Read the `List RemoveAt` documentation carefully to make sure you're getting the correct complexity.

In the `Collector` class, change the `OnTriggerStay2D` method to remove the target pickup from the list of targets and destroy the target pickup to remove it from the game. Be careful here, because by the time you get to the target pickup it may not be at the end of the targets list. You'll need to find the target pickup in the list of targets, then remove that element from the list.

If the list of targets isn't empty at this point, you need to update the distance for each target in the list of targets because all the distances are incorrect (assuming the collector moved to pick up the target pickup). Luckily, I provided an `UpdateDistance` method in the `Target` class, so all you have to do is call that method for each target in the list. Of course, now that you've changed all the distances you need to sort the list of targets; calling the `Sort` method on the list of targets will do that for you. Finally, call the `SetTarget` method to set the new target pickup (it's at the end of the list of targets) and start moving toward it.

If the list of targets is empty in the `OnTriggerStay2D` method after you remove the target pickup, set the target pickup field to null so you respond correctly next time a new pickup is spawned.

When you run the game, the collector should collect the pickups properly, always targeting the closest pickup for destruction ... I mean collection.

Step 7: Speed up collector when a new pickup is spawned

Let's make things a little more interesting. In this step, you're making the speed the collector moves at dependent on how many pickups there are to collect. This is interesting because it seems like the collector is hurrying the more work (collecting) it needs to do.

Before changing any code, select the collector game object in the Hierarchy window and change the Collision Detection mode in the Rigidbody 2D component from Discrete to Continuous. The Unity scripting reference for the `CollisionDetectionMode2D` enumeration says that with Discrete collision detection the engine will only detect collisions at the new rigidbody position while with Continuous collision detection the engine will detect all collisions as the rigidbody moves. We want to avoid a scenario where the collector is almost at the target pickup on one frame but is past the target pickup on the next frame because the collector is moving so fast. If we use Discrete collision detection in that scenario, we'll never detect the collision between the collector and the target pickup.

In the Collector class, change the `GoToTargetPickup` method to apply a force equal to the `BaseImpulseForceMagnitude` plus the `ImpulseForceIncrement` times the number of pickups currently in the game.

When you run the game, you should see the collector move faster the more pickups there are in the game.

Step 8: Cap the total number of pickups the PickupSpawner spawns

It's hard to see the collector actually slow down at this point because the number of pickups in the game stays fairly constant once it reaches `MaxNumPickups` in the `PickupSpawner` class. In this step, you're capping the total number of pickups the `PickupSpawner` spawns.

In the `PickupSpawner` class, declare a constant to hold the total number of pickups to spawn and set the constant to 50. Add a field to keep track of how many pickups have been spawned so far. Add the appropriate code to the `HandleSpawnTimerFinishedEvent` method to make the spawner stop spawning pickups once it's reached the total number of pickups to spawn.

When you run the game, you should see the collector race to collect the pickups, then slowly slow down as it picks up the remaining pickups once the pickup spawner stops spawning pickups.

That's it for this programming assignment.