Object Pooling System Team Assignment

IMM215 Programming For Immersive Experiences

# Objective

Create a pluggable object pooling system. The system can either be a generic one (given some prefab) or specific to a use case (like, say, bullets).

You also create some example project to test and demonstrate the system.

Students may work in groups no larger than two. There is an option for individuals, but this is just to handle odd numbered classes with someone volunteering to work alone.

The code needs to be on in a git repository that is uploaded consistently as progress is made, and the git repo will be a small part of the process.

# Submission

Submit a link to the repository. The readme.md should include student names and a brief description of how collaboration went, in addition to detailed instructions on how to set up and use the system.

Likely, the project will just *really* be one or two scripts, but the whole repository needs to be updated and operated on through github.

Not required, but a future goal is to be able to create an official release in the form of a .unitypackage that I can find in the releases tab of the repository. Creating unityPackages is done from the assets menu>export package menu with a folder selected.

The project should be in its own namespace and organized into appropriate folders.

# What’s an Object Pooling System?

Instantiating and Destroying gameObjects is an “expensive” process. It’s more efficient to enable and disable gameObjects. Disabling objects keeps them in memory.

An object pooler is a system where you use a list or a dictionary to keep track of gameObjects. You create your own replacements to **instantiate** and **destroy**, which enabled and disables objects, and moves/rotates/resets the disabled object when it gets enabled.

# Purpose

This assignment is twofold: First, it is a practical and hands-on use of git for collaboration. Hypothetically, the students do not need to meet up in person! It would be silly not to, but still.

The main scripting goal is working with lists. Lists are amazing! We love them and use them all the time in Unity development.

# Grading Information

**For team projects:**

70% Pooling system. The code. How well it works and how efficient it is.

20% of the grade will be on how well the team collaborated, if at all. If only one student did most of the work, they will both lose this part of the grade.

10% The Git repository. Are commit messages descriptive? Was code added to branches and then merged? Does the official release have a description? Were errors resolved without restarting or redoing any work? Etc.

**For Individuals:**

70% Pooling System. The code. More lenience on functionality and bugs given to individual than groups.

15% Internal code documentation. Can I read your code and understand it? Are all functions commented with what they do and how to use them? Are function and class names clear and descriptive? Etc.

15% Github repository, as above, but with an added emphasis on the readme and documentation, making it easier for an outsider to set up the system.

Basically, I am putting an honus on individuals to assist with working with *future* collaborators.

# Assignment Bonus Points

A way you can achieve up to 5% back (5% bonus but cant get over 100) on this assignment by contributing to a different team’s object pooler project.

You must do it through GitHub, *fork* their repository into your own (use github UI), clone that to your computer, make a branch, commit the changes, and create a “pull request” to propose your changes which they will have to accept.

In terms of academic integrity, you may **not** complete any of the object pooler for the other group. Your contributions must be meaningful – an actual change – but not meaningful to the graded assignment. Basically, they should be in the example scene that the pooler uses. They must not interfere or touch their assignment in any way. All changes will be publicly visible in the pull request.

# Getting Started

A prereq here is to know about object instantition. That’s the GameObject.Instantiate function we love so much.

You have three strategies for making a pooling system, and they involve how much you are going to assume about the objects you are creating and destroying.

The first is if they are all the same type of object, or is this a global system that can create anything, copying it from a prefab. These usually use a collection of lists for each prefab type, and use a dynamically created dictionary to go from prefab-type to appropriate list. It’s more complicated.

The second assumes all the objects have the same components, so it doesn’t worry about that, just the position of the object. The other scripts would likely be set up to use OnEnable() and OnDisable() for initialization to replace Start() and Destroy(). Basically, it’s a system for a single prefab. A game that required multiple types of objects being pooled would use multiple of these systems at the same time.

The last is specific to a game like bullets, which give you what you need in the instantiation properties, like its spawn location and velocity, and the pooling system sets all the various properties of various components correctly. Very convenient! Sometimes I will write an object pooling system right into the code I am working on, such as when I am being lazy and want future headaches for myself.

**I recommend the second approach.** Have a system that takes a single prefab and has public functions to create and destroy those prefab objects.

--

The function to enable or disable a GameObject is “.SetActive(true);” or” someGameObject.SetActive(false);”

--

Enabling a disabled gameObject means we need to get a specific reference to the gameObject, and we have a list. If we have a list of only disabled gameObjects, then we just grab one from it, take it off that list, and there we go. If we have a list of all of the gameObjects – enabled and disabled – then we need to loop through the list and check if the gameObject is active or not (if its disabled) (“someGameObject.activeInHierarchy”), and once we find a disabled one, we set it to active.

https://docs.unity3d.com/ScriptReference/GameObject-activeInHierarchy.html

If we don’t find any, then we either throw an error, or Instantiate one and add it to our list. Doing that means the list will dynamically grow as large as it needs to, if needed. This does mean that we may end up with a really large list of disabled objects just because the number on-screen spiked once. This… is probably fine.

Looping through a list is simple, we can use a for loop.

for(int I = 0;i<someList.Count;i++)  
{   
if (!someList[i].activeInHierarchy){  
 //it isn’t active. Note the ! above.  
}else{

//its active  
}

}//end loop