worksheet 1: Unity, Game Objects, and Scripting

UFCF9M-30-2 Game Engine Programming

The aim of this worksheet is to re-introduce you to fundamentals of working with the Unity game engine. Completing this worksheet will teach you how to work with Unity’s MonoBehaviour architecture, how to script with C# in Unity, and how to work with its game object model.

It is assumed that you have some experience writing Object-Oriented code from Games in C++, working with both Unity and Unity Hub from Principles of 3D Environments last year, and working with GitHub / GitKraken from both modules.

In this worksheet you will implement a state machine for controlling the flow of gameplay, an inventory data structure and an item object. Extension tasks suggested within the worksheet will encourage you to build these in a way that is strongly object-oriented, and add features to these that can be built upon in later worksheets.

This worksheet has been designed to cover learning content from teaching weeks 2 to 4. As such, it is expected that you will complete this over a three-week period. Feel free to read ahead if you have completed core tasks and feel ready to progress before the next session/s.



# Checklist

Below is a checklist of core and extension tasks contained within the worksheet. Feel free to use this to keep track of where you are as you work through it.

## Core Tasks

Project Cloning  
Game Manager & State Machine  
Inventory Script  
Item Object & Script  
Adding Items

## Extension tasks

Custom Scene Assets  
GitHub Branching  
Switch Statements  
Class-based FSM  
Inventory Encapsulation  
Inventory Abstraction & Inheritance  
Item Sorting  
Object-Oriented Items  
Add. Item Properties  
Item Models  
Item Components

# Week #2: Game Engine Architecture

In this week’s content, we covered the underlying architecture of game engine software, including the game loop and game states. This section of the worksheet will help you clone the project from GitHub and implement a state machine for the features you will build over the course of this module.

## Cloning the Template Project

To begin, let’s download the template project for this assignment. This will contain the standard 3rd Person Controller, alongside some changes to the project settings in order to help the teaching of this module and make following worksheets easier. The template is hosted through GitHub classroom here - <https://classroom.github.com/a/rFGkup0y>. Follow the instructions on this page to set up your GitHub repository with your GitHub account.

Now, clone the repository onto your local working space through GitKraken / GitHub Desktop. If you are using the Lab PCs, we recommend that you do so on the D: drive – this is the local storage for the machine and will give you enough space to clone the repo and develop your project. Note that as this is the PC’s local space, you may need to periodically delete other files / folder contained on it and may have your work deleted in turn. It is therefore important that you regularly make commits and push to your repository if working on the Lab PCs.

A screenshot of a computer

AI-generated content may be incorrect.

Finally, open Unity Hub, add your cloned repository from the D: drive to your projects, and click on it in the list to open it. Once the project has loaded, open up the GEP Sample Scene found inside the GEP Core folder. You’re now ready to start developing!

A robot standing in front of a white wall

AI-generated content may be incorrect.

## Adding the Game Manager and State Machine

Now that you have the project cloned and local repository set up, it’s time to start adding some functionality! Start by creating a new folder in Assets for your scripts (Remember, a good folder structure makes it easy to find things when programming problems are stressful enough). Then, create a MonoBehaviour script for your Game Manager.

Let’s create the architecture for our state machine. We are going to include two states; One for gameplay, and the other for when the game is paused. We will toggle between these states by using the Enter / Return key on the keyboard, and implement pausing by setting the internal time scale to zero. This will affect physics and animation speed but will not interfere with the game loop (which runs on an unscaled time value), allowing us to press the enter key again and set the internal time scale back to its default.

Open the Game Manager script and create an Enum for our game states. Next, in the Game Manager class, create a Game State variable to keep track of the current state. In Start(), set the value of this variable to be the gameplay state. Then, in Update(), create if statements to check the value of our Game State Variable, with a nested if statement inside each calling Input.GetKeyDown(KeyCode.Return). Finally, write a line of code in each of these toggle the value of our Game State Variable.

A computer screen shot of a computer program

AI-generated content may be incorrect.

Now, create an object in your Unity scene for your Game Manager, and attach the Game Manager script we have just made as a component. If you play your scene, you should see the game state change in the inspector.

A screenshot of a computer

AI-generated content may be incorrect.

Now we need to add some behaviour for when our game state changes. We could do this in Update with the code we have written so far. However, this could cause complications when multiple state changes and behaviours happen at once, and things can get chaotic when eventually trying to debug. It is always best, therefore, to separate decision making from behaviour when modelling state machines. Let’s use the game loop in Unity to our advantage and put this new code in the Late Update instead, so that behaviour occurs after we have determined which state we end up in.

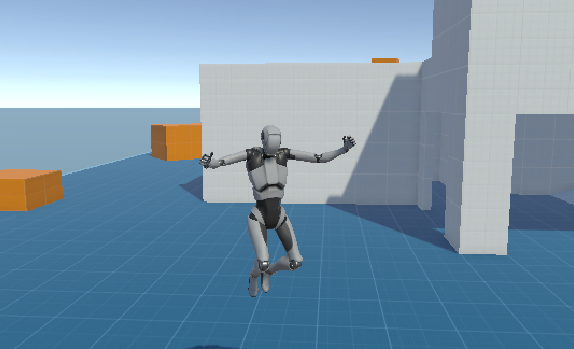
As we are passing through functions, we need to keep track of if the game has changed state this frame or not. Create a boolean in the GameState class to hold if the game has changed state this frame. Then, where we have changed the value of the game state in Update, set the value of this Boolean to true.

Next, add the LateUpdate function to our Game Manager class, and include inside it an if statement on the value of the game state boolean, checking if we have changed state this frame. Inside this, if it is the pause state, set Time.timeScale to 0.0f. If it is the gameplay state, set Time.timeScale back to the default value of 1.0f. Finally, we can set the Boolean determining if we’ve changed state back to false, ready to be toggled next frame if there is a game state change. Your code should look something like this:

A screen shot of a computer program

AI-generated content may be incorrect.

If you play the scene, you should now see that when you press the return key, time stops until it is pressed again. Congratulations, you have built a successful game state machine!



Yippee!

## Extension Tasks

Once you have successfully completed the work outlined above, consider the following ways in which you may want to extend your project to further its quality, your understanding, and your programming skills:

* ‘If’ statements work okay, but get complicated when checking one variable for multiple values. Consider using **switch statements** rather than a series of if statements in the update and late update functions.
* The FSM we have developed for controlling our project is enough for the purposes of the assignment, but we can make it better. You will have seen from the lecture content how we can build a state machine based on **Classes** rather than Enums. Have a go at implementing a **class-based state machine** and **including additional states** and transitions, such as a Pause state.
* The default Unity third-person scene is a little bland. You should already know from Principles of 3D Environments how to import assets and create a game environment. Consider sourcing **additional asset packs** from the Unity Asset Store / elsewhere and **making the scene a bit more interesting** (Be careful of GitHub file size limits and push asset packs one at a time!). This should still be contained within roughly 300 x 100 x 300 units.

# Week #3: Scripting in Game Engines

In this week’s content, we covered scripting in game engines and best practices for doing so. This section of the worksheet will help you to begin scripting your inventory system.

## Creating the Inventory Script

Let’s start by making a new MonoBehaviour script for our inventory and adding it as a component to the PlayerArmature object in the scene. In the following week’s content, we will begin modelling our game objects. For now, we will treat all items as having a single property – their name.

We will need a data structure to keep reference of the items our player has collected. We will need to access this structure at various points, not just at the first or last element (for which a Queue or a Stack would be appropriate). Although an Array would be good for optimized memory and inherently include a maximum inventory size, this worksheet will teach you how to use Lists as its size can be either fixed or dynamic, and accessing items in the way we intend to will be simpler to write.

To create a list, you will need to include the System.Collections.Generic library at the top of your script, then use the following syntax in the Inventory class.



Now that we have a data structure, we can add some functionality to our inventory. Write two public functions, one for adding items and the other for removing them. We will make these public so that they can be called from outside of the script, allowing other objects to tell the inventory to add an item to it, or remove one from it. These should also include a parameter for a string, so that the item to be added / removed can be specified. In each function, call the Add / Remove function on the item list as appropriate, and pass through the parameter variable as an argument like so:

A black background with white text

AI-generated content may be incorrect.

We now want to test these functions work. To do this, we are going to write some temporary code in the Update function for the inventory before we implement our items fully. Using the Input.GetKeyDown() function, implement some code that adds and removes some item names from our inventory using the functions you have just written – below is some example code. You can test this functionality by looking at the Inventory component attached to the PlayerArmature in the inspector.

A screen shot of a computer code

AI-generated content may be incorrect.

We can now test that our code works! – but there’s another problem we’ve introduced. With the code we implemented in the previous section, we can end up adding or removing items from the inventory even when the game is in the paused state. Even if this is only temporary code, we should still control where and when this code is allowed to execute. To get this to work, we need to get our two scripts able to communicate with one another.

In the inventory class, create a public object of the game manager class. This will create a field referencing / pointing to an instance of the game manager type, which we will then need to populate. This can be done by dragging and dropping a game object that our desired class is attached to as a component, or by calling the FindAnyObjectByType<T> function replacing the generic identifier T with the class you are looking for. The latter is a useful but expensive function and is best to do as infrequently as possible (not in an Update function, for example).

A screenshot of a computer

AI-generated content may be incorrect.

With this done, have a play in the scene with the inputs you have tied your functions to. Try changing game states, see if your code runs in the functions you’ve specified only.

A screenshot of a computer program

AI-generated content may be incorrect.

## Extension Tasks

Once you have successfully completed the work outlined above, consider the following ways in which you may want to extend your project to further its quality, your understanding, and your programming skills:

* The inventory structure we have created uses public variables throughout. While this works, it could lead to unwanted modification later down the line, either by yourself or another developer. Consider how we can use **accessors** to prevent unwanted manipulation from elsewhere in code, keeping with the OOP principle of **encapsulation**.
* After you have done this, if you still want to modify these values in the inspector, look into the **[SerializeField]** attribute in Unity and consider using these with your private variables.
* The inventory is attached to the player and works for that purpose - In future development, we may want to add different kinds of inventories for other game elements, such as item containers or NPCs. Consider how we can use **abstract classes, base classes and derived classes** to reuse our inventory class later, in keeping with the OOP principles of **abstraction** and **inheritance.**
* The data structure we have used works for the purposes of this worksheet. In gameplay, we may want to automatically organise items for finding a given item within this structure, either on entry or on command. You will have learned about **sorting algorithms** from Games Tech 101 last year – consider choosing one and try implementing it.

# Week #4: Game Object Models

In this week’s content, we covered game object models. This section of the worksheet will help you to implement items into your game world, as well as tie them to the inventory system.

## Creating the Item object and script

Having completed our inventory and got functions for adding and removing items working, we can move on and begin creating items that can be collected by the player. Before we start, we should try to organise our scene so that as our scene gets busier, we avoid struggling to navigate through our hierarchy. Create an empty game object as a child of Environment that we can use to parent all our potential items, and name this appropriately.

To create the item object, right click the hierarchy and select one of the 3D object primitives available. Drag this new item into the object we previously created to parent it. Alternatively, right click the previous object instead of the hierarchy and the created object will automatically parent to it. You will see in the inspector this object by default has Mesh Filter, Mesh Render and Box Collider components. You should remember from last year that the Mesh Renderer allows our object to be represented by a 3D model, and the Collider allows our object to behave with physical properties when other objects intersect it. Create a new MonoBehaviour script for our items and give it a field for the item name.

A cube on a blue surface

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Note: When you create a script in Unity, it will automatically inherit it from the MonoBehaviour base class, along with the Start and Update functions. These functions will be called by Unity’s engine during the update cycle as we discussed in the lecture content. It also treats the script as a component for Unity’s GameObject model. If your script is pure code, it does not need to inherit from MonoBehaviour and may cause problems if it does. Therefore, if your script is not something that should be an object component, you should remove the MonoBehaviour inheritance.

## Adding Items to the Inventory

Unity’s Character Controller, which has been included in the template project, does not behave with the rest of its physics system and other object cannot detect collision with it – but it can detect when it collides with another object. Therefore, we are going to implement item interaction through the Inventory class. Do this by adding the OnControllerColliderHit function to the script.

This function will be called when the character controller collides with any collider object. If we tried to add every object it collided with to our inventory, we would be trying to add the floor, the buildings etc. Therefore, we need to check the object called is an item, and not anything else. There are a few ways to do this, but I suggest evaluating if the object collided with has an item component by calling the GetComponent function for the item component type. If this returns true, we can get the item name string from the component and add it to our data structure.

A screen shot of a computer code

AI-generated content may be incorrect.

We now want the item object to disappear from the game world – it doesn’t make sense for it to both be in our inventory and in the world, and this would break the player’s sense of immersion. To do this, call the Destroy function below the code we have just written, passing through the gameObject property of the collider our character controller has hit. This will dis-instantiate the object from the game world and memory.



Finally, we want to disable the temporary code we wrote in the previous section for testing items being added to the inventory – we now have a way to do this in the game world. It is best practice to comment out old code rather than deleting it, even if you’re pretty certain it won’t be used any more – that way, other developers that might have to look at your code understand how it used to work, and you have something to return to if your new code introduces new problems while trying to develop new features.

A robot standing on a blue surface

AI-generated content may be incorrect.

Congratulations on completing the first GEP worksheet! At this point, you should have implemented a finite state machine for controlling gameplay, an inventory attached to the player, an item that exists in the game world, and a way to take the item from the game world into the inventory’s data structure.

## Extension Tasks

Once you have successfully completed the work outlined above, consider the following ways in which you may want to extend your project to further its quality, your understanding, and your programming skills:

* Like our inventory class, the item class we have implemented is poorly aligned with OOP principles of encapsulation, abstraction and inheritance. Consider refactoring this using **encapsulation, abstraction and inheritance** as suggested previously.
* Our items currently only have the name property. Consider **additional properties** that you may want items to have in your inventory system and implement them. Extension tasks in later worksheets will suggest ways in which this extra data can be used.
* We are currently using 3D primitives to represent our objects. Consider **sourcing 3D assets** to make our items look like the things we want them to represent. These should fit thematically with your 3D environment where possible to find those that are appropriate.
* In making the Item object we only added a collider to it, and at the moment, interacting with it isn’t very interesting. Consider **additional components** that could be added to the items in your project.