Practical Gaming 2022

# Name of Student…..

Jack Patterson

# T Number ….

T00217640

# Name of Project

ResourceCollector/Builder

# Gameplay

You will begin with 5 characters at the beginning. Each will have no job assigned at the start of the game. They will stand idly by until you click on them which will pop up a window which will display the name of the character, it has a dropdown menu for you to decide which job to choose for this person and also a brief description of what the character is currently doing (e.g. idling).

Once a job is selected for the character they will go to complete their job by pathfinding to the nearest relevant object of that type. As of the writing of this document only the miner and the lumberjack work and the other two jobs require features I didn’t have time to implement.

The characters will begin collecting resources which they will begin to deposit into their storage. The aim of this was to allow the person to build a building with this but the building system wasn’t satisfactory for this so I chose to exclude it. None of the code has been removed however and certain parts are accessible through the use of certain keyboard keys. Right now it is in a somewhat basic stage overall but there are parts of other unfinished elements such as a partly completed script for wild animals I intended a hunter class to use instead of the currently present “farmer” job.

# Coding

Under each of the following headings, please describe the concept, why is it or isn’t it useful/needed, where do you implement in you project, you may provide screenshots or cut and past code segments etc..

* Frame Rate Independence
  + I didn’t have to put much effort into this as the project made heavy use of the unity AI features which are framerate independent.
  + One area that is frame rate independent would be my use of coroutines which use real work time using the functions such as yield return new WaitForSeconds(time);
* Interfaces

I didn’t make heavy use of interfaces as in most occasions I had more than one option to make scripts communicate, generally through the manager. However I did one of interface in particular which was IInteractable. The interface originally started out to aid my state transition system for character states but eventually changed into one which returned resources which were heavily used throughout the project with most objects including the player having at least one resource associated with them. It was also used to establish an essential link between scripts such such as harvestableObject and CharacterControl to allow them to exchange resources etc. . interface IInteractable

{

ResourceManager.Resource interact(CharacterControl NPC);

}

* + The other interface I used was IQuestable which was used by all quests which would set the quest as completed. This feature technically works but is not fully implemented due to lack of time.
* Inheritance
  + Once again, inheritance wasn’t heavily used and was only done with the interface. A lot of my scripts in hindsight could have been inherited as there is a lot of overlap between certain scripts and I intended to with some, but I prioritised getting a working game before working on remaking scripts to be cleaner like this.
  + Scripts such as my CharacterControl and the mentioned ForestCreature do inherit from a common ControlScript which simply contain common methods such as the case control and the moveTo scripts.
* Case pattern
  + I make heavy use of switch statements throughout my project, generally to return something else I need that used enums to cycle through.
  + My biggest one would be my characterControl one which makes use of two big case patterns with it. My character states governs how my overall character moves while I also have a secondary case pattern which serves to clean up anything left from my primary one such as leftover animations etc. I make use of it one more time in that script to return what the character is doing to the UI.
  + Manager makes use of one to return a sprite value from a file.
  + ResourceManager makes use of them for returning the Item object associated with a resource, to return the time to mine of the one item, to return the time for the harvestable object to respawn once its destroyed and finally to return the job which is associated with that resource.
  + CharacterUI uses it to set the job of the selected job from the dropdown menu.
* Observer Pattern
  + I make use of Observers in the managers. While they themselves are singletons they contain data similar to an observer.
  + QuestsManager stores data on all quests and their current status.
  + ResourceManager saves data on all resources, items and much more data associated with them.
* Polymorphism
  + I make use of it in my ControlScript and the classes that inherit it which allow me to overwrite classes such as the moveto script.
  + One notable exception however is making use of colliders when gathering which objects near my characters target. They make use of the super class “Collider” to interact with all colliders in the area.
  + One other example is in my CharacterUI script where I overload the super “Array” and cast into an array of type job[] which allowed me to input data as I needed.
  + I did have another overwrite when I was attempting an IO approach to quests but I scrapped that in favour of an interface.
* Communication between scripts/game objects
  + As stated in interfaces I made use of IInteractable to allow the CharacterControl to communicate by giving each a reference to the other. I also use IQuestable to communicate with quests from the QuestsUI and QuestManager.
  + Primarily due to what I focused on a lot of communication is with the resourcemanager which the majority of scripts use in some form. Anything that uses a resource communicates with it and when taking in new items the resourcemanager is told that is done to update the total. Since the resourcemanager knows everything about the resources it manages basically everything talks to it I some way.
  + The other big example in my mind is the CharacterUI which talks to the Manager to be enabled/disabled as well as the CharacterControl telling the manager to enable it when a character is clicked.
  + Other scripts that talk a lot are the UI scripts with their relevant manager. So ResourceUi talks heavily to the ResourceManager, the CharacterUI to the general Manager and the QuestsUI to the QuestsManager. That is also true for the subcategories of scripts with them talking to their manager such as the Builder scripts to the BuildManager.
* Instantiation and Prefabs
  + The ResourceManager instantiates a lot of trees and rocks on game start.
  + The build scripts make heavy use of instantiation for things such as the objects and the walls.
  + Prefabs are used so heavily I’ll just generalise it to say they’re everywhere. The map is made up of solely prefabs excluding the terrain. The UI is made up of prefabs etc. I have a whole folder with a ton of prefabs.
  + Each manager is a singleton which makes use of a single instance of a script to work.
* Magic Numbers
  + Magic numbers are used in some places such as the idle/rest position
  + Village bounds are also hardcoded to avoid trees and rocks being spawned within the village
  + Some of the maths for the wall script is hard coded, especially to do with angles.
  + Inventory max numbers are also hard coded
* Model Animation
  + Model animation is only done in the primary characters with them focusing on a few core animations that can be reused such as a hitting/swinging animation and a walking one.
* Self made models and or animations
  + TBA
* Interactions between objects/scripts
  + My UI is made up of scripts which control how the UI functions and what it displays.
  + The characters interact with objects such as trees, rocks and storages using their scripts.
* Propper code placement
  + I made use of proper code placement throughout by placing vital parts within the various managers which kept track of vital things such as resources etc.
  + I made sure to place appropriate UI elements within the various UI scripts which don’t handle other game elements themselves and refer to the managers to do that.
  + All scripts outside of the general “Manager” have a set function and cover a certain appropriate area. For instance, the ResourceManager is only used for resources and the HarvestableObject script only handles the mineable objects such as rocks and trees while referring to the ResourceManager where appropriate.
* Code repetition
  + I attempted to avoid code repetition where possible by doing things such as creating an object system for my Item script.
  + I also originally had overlapping code for my characterControl and forestCreature script which I avoided by creating a common class for them to inherit from.
* Feature 1: IO
  + I made use of IO in my scripts. The biggest example of this is in my CharacterControl and Manager script which will read a random line from a file and will give a character that name on game start. It also does this based on a selected gender.
  + I also made use of IO to retrieve a sprite from my Resources folder (reserved folder which is supposed to be used to retrieve such things at runtime). These sprites are then used on my UI.
* Feature 2: UI
  + I make use of UI heavily in the use of the game. The bar on the top keeps track of all resources in storage at that time.
  + The characterUI which can be accessed by clicking on any character at runtime will open a bar at the side which can be used to assign a character a job and will tell you what a character is currently doing at any time. This can be exited using escape.
  + The pause menu will pause the game when used and will give you three options: the resume will unfreeze time and will allow you to play as normal, the exit to menu is intended to move you to the main menu and the exit to desktop will close exit the game.
  + Finally while it’s not completed in time the QuestsUI was intended to load up a quest and allow you to complete by doing the listed task on the side bar.