Design Rationale For OOD Assignment 3

Documentation of Each Task And How It Is Implemented Into The System

Going to Town

Implement a Town level.
This new level will need to be hard-coded as a list of strings and be passed into a new GameMap instance along with the same groundFactory used to initialise the map in the Application class.

Creating a Vehicle
A new class called Vehicle will be created that will inherit from Item. To move the player, Vehicle needs a MoveActorAction() but it must be added through the Application class because the Vehicle does not know about the gameMap. We create a method called addAction that takes in an action and adds it to the Vehicle's allowableActions array.

Place a vehicle somewhere on the existing map
A Vehicle object will be instantiated and placed on the map through the Application class. Adding the MoveActorAction to the Vehicle allowableActions will also be done in the Application class.

New Weapons

Using range weapons with submenus.
Both the shotgun and sniper rifle have complicated mechanics that will require many options for the player to pick to use them. These options would include things such as; for the shotgun: aiming in all 8 directions on the map; for the sniper rifle: selecting which zombies to shoot and how long the user should aim for. The menu provided to the player at every turn can only hold up to 26 options, so it would be bad usability to spam the menu with multiple options for an action. Hence, submenus will be used for both these range weapons. When the player has either a shotgun or a sniper rifle in his/ her inventory, there will be an option to use it. Selecting it will prompt a submenu provided to the player so that they can further specify what they would like to do with the ranged weapon.

The implementation of the submenu will be similar for both shotgun and sniper rifle. We will create a respective Action for both items; UseShotgunAction, and UseSniperAction. Both of these actions will generate more actions and create a new Menu instance to display the action to the player. With the selection of an action within this new submenu, it will return the

respective action to the original action menu which will then return the action in the Player playTurn method.

Giving players Actions

To give players actions to use the range weapons, we create a new enumeration called RangedCapability that contains SHOTGUN and SNIPER. Having an item with SHOTGUN will give the player a UseShotgunAction. Having an item with SNIPER will give the user UseSniperAction.

Shotgun

A new class called Shotgun will be created which inherits from weaponItem class. On instantiation, it will be given the SHOTGUN capability. To retrieve this action for the player, we use the player's getAdditionalActions method implemented in the last assignment which iterates through the player's inventory and retrieves possible actions from the player's items based on their capabilities.

Shooting with the shotgun

The UseShotgunAction inherits from the Action class. The Execute method of this action creates a 8 ShootShotgunAction, one for each of the possible 8 directions, and presents them as a submenu for the player to choose which direction they would wish to shoot in. UseShotgunAction stores each direction as a set of coordinates that can be used on the player's location to calculate the range of the shotgun shot in a specific direction. ShootShotgunAction is an action that takes in an array of coordinates, and uses this array to calculate the Locations of the range, and deal damage to the actors contained within each Location. The submenu is a Menu() object that is an attribute of USeShotgunAction. 8 different actions are needed as the Menu() class only takes in actions in its showMenu() method and returns a single action that the player chooses.

How shooting will work will be as follows: when the execute method of UseShotgunAction is called, it creates 8 new UseShotgunActions with each direction and places them into a new instance of Actions. UseShotgunAction then calls the showMenu() to display the submenu. The ShootShotgunAction that showMenu() returns will be saved and then its Execute method is called. ShootShotgunAction will then calculate the range, deal the damage to the actors, then return a String which UseShotgunAction returns as well.

Shotgun ammo.

To keep track of the shotgun ammo, a new class called ShotgunAmmo that inherits from PortableItem is created. ShotgunAmmo will have an attribute 'quantity' that will hold the quantity of ammunition, a method to decrement quantity, and a method to increment quantity. A player should only have one instance of ShotgunAmmo in the inventory, picking up more ShotgunAmmo will result in adding its quantity to the quantity stored in the player's inventory.

To do this, a new CollectAction was created that inherits from PickUpAction. Its Execute method will be overwritten to check for any items in the player's inventory with SHOTGUNAMMO capability. If there is then Execute method will narrow cast item into SHOTGUNAMMO; this is ok because only items with SHOTGUNAMMO capability are instances of ShotgunAmmo class. This way, we can access in ShotgunAmmo, and add the quantity of the picked up ShotgunAmmo object into the existing instance in the inventory. If no items with SHOTGUNAMMO capability exists, then the ShotgunAmmo would be added to the inventory like how PickUpAction would. The getPickUpAction for the ShotgunAmmo class is also overwritten to return a new CollectAction instead of a PickUpAction; This was also the reason why CollectAction needs to be inherited from PickUpAction as the return type of GetPickUpAction Method is a PickUpAction.

Shooting with Ammo

The player can only shoot with a shotgun if he/she has the ammo for it. In the Execute method for UseShotgunAction before the submenu is displayed, the method will check the Actor inventory for any instances of items with SHOTGUNAMMO capability. If there is none, the method will return a String indicating that the player has no ammo, else it will continue presenting the user with the submenu.

Using up Ammo

Inside the ShootShotgunAction's Execute method. The first few lines of code will search the actor's inventory for the instance of item with SHOTGUNAMMO capability, this item will then be narrow casted to ShotgunAmmo type; Note that there will always be an instance of this item as if there was not, the UseShotgunAction would not have generated a ShootShotgunAction for the player. The Execute method will also save the index of the item in the inventory so that the item can be removed from the inventory when the quantity of the ammo is 0. Once the item is found, the Execute method calls the ShotgunAmmo's Use() method which decrements the quantity. The Execute method also runs an if statement with ShotgunAmmo's Empty() method, which returns true if the quantity is 0. When the empty method returns true, the Execute will remove the object at the index stored in the player's inventory; this index represents the ammo as mentioned earlier.

Sniper Rifle

Shooting with the Sniper Rifle

For this task, We created a new class called SniperRifle class that inherits from WeaponItem class, a UseSniperAction class that inherits from Action class, a AimSniperAction that inherits from Action class , a ShootSniperAction that inherits from Action class and a SniperAmmo class that inherits from PortableItem.

The UseSniperAction would have an execute method that creates a new Actions object. Then it would check through the whole map and if there are any Zombies or boss enemies on the map, we create a new AimSniperAction that takes in the target and its aim concentration for each of the enemies and add them to the Actions object. After that, we call showMenu() in order to get a submenu based on the Actions object and if the action is not a null, then it calls the execute method of that action. We also had an overriding getNextAction that returns a new AimSniperAction and a menuDescription() method. We had to check through the whole map to find targets is because if not, we have to take in a new parameter for UseSniperAction for the ActorLocations, which is not really required.

The AimSniperAction takes in a target actor and an integer for its concentration, the class would have an execute method that creates a new actions object, then it would add a new ShootSniperAction and a new AimSniperAction so that the player is able to choose from the submenu which action to run. After that, we would call showMenu() on the new actions and if an action is an instance of ShootSniperAction, we would run the actions execute method, if the action is an instance of AimSniperAction , we would a boolean aiming set to true, which it was initialized as false. Else, it would return null. We also a returnTarget method which returns the target actor and a getNextAction() method which checks that if the boolean aiming is true, we create a new AimSniperAction with concentration + 1. This checks that if the user wants to aim again then it would redo that action with added concentration.

The ShootSniperAction also has a constructor that takes in an actor and integer concentration. It also has an execute method that checks that depending on the value of the concentration, it would increase the chances of hitting the target and increasing the damage toward the target, where aiming twice would result in an instant kill of the target. Having a concentration of 0 will have 75% chance to hit with standard damage, concentration of 1 will have 90% chance to hit with double damage and concentration of 2 will have 100% chance hit and will instakill.

Sniper ammo.
How SniperAmmo will work is exactly the same as ShotgunAmmo as mentioned above, except the capability of SniperAmmo will be SNIPERAMMO instead of SHOTGUNAMMO.

Losing Concentration
Adding on to the RangedCapabilites is an enum called FOCUS. When a player carries out a UseSniperAction, its Execute method will add the FOCUS capability to the player before displaying the menu to choose a target. In AimSniperAction, the Execute method will check if the player has the FOCUS capability before displaying the menu. The Execute method will return a String indicating that the player has lost concentration otherwise.

To lose the FOCUS capability, the player’s hurt method was overwritten so that whenever the player is attacked, it will remove the FOCUS capability in the player.

Mambo Marie

For mambo marie, we would create a new class called MamboMarie which would inherit from the Actor or ZombieActor class , a ChantingAction class which inherits from Action , a ChantingBehaviour that implements Behaviour and a NewWorld that inherits from World class. We can also add a new capability to ZombieCapability called UNDEAD BOSS for MamboMarie to define her role and can be used for other tasks.

The ChantingAction class would have an execute method which would having an integer variable which would count the number of zombies spawned and a while loop that runs until there are 5 spawned zombies on the map. If the map does not contain an actor, then we would add an actor randomly onto the map and increase the count variable.

The ChantingBehaviour class use getAction() method which would have an turnCount int variable which would keep track of the turn where MamboMarie is on the map, and whenever it is a multiple of 10 (or divisible by 10), then it would return a new Action called ChantingAction. Else, it would increasing the turn by 1 and return null.

The NewWorld class would have an overriding run() method and an overriding processActorTurn(). The run() method would, in addition to running the whole game, also checks through actorLocations to find out whether MamboMarie is on the map or not. If we loop through the actorLocations and there is no actor which has ZombieCapability of UNDEADBOSS, then we create a new MamboMarie object and there is a 5% chance of randomly adding the boss in the 4 edges of the map.
The run() method would also check that if the turn reaches 30 and boolean bossKilled is false, we remove her from the map. Else, we set bossKilled to true.

The processActionTurn() is mostly identical to the method in World class, however we add that if there is an Actor close the player which is of ZombieCapability of UNDEADBOSS, then we add a new AttackAction for that enemy type to actions.

Ending the game

A “quit game” option in the menu

For this option, we create a new class called QuitGameAction that inherits from Action class where it would have an execute method which removes the actor from the map and returns a string stating that the player has quit the game.

We then add this action in the Player class method, playTurn(), so that we are able to quit the game at any time.

A "player loses" ending for when the player is killed, or all the other humans in the compound are killed

For this ending, we create a new method called stillRunningLoseEnding() in NewWorld class that checks through actorLocations and if both player and human are still on the map then it returns true. Else, it means that one of them is dead which means that the player loses the game so we return false in order to break the while loop that runs the game.

A "player wins" ending for when the zombies and Mambo Marie have been wiped out and the compound is safe

For this ending, we do the same thing we did for the "a player loses ending" , where we create a new method in NewWorld class called stillRunningWinEnding() which does almost the same as stillRunningLoseEnding() , except that it checks whether MamboMarie or a Zombie is still on the map, then it returns true. Otherwise, it means that the player wins since he has defeated the boss and there is no more zombies on the map. For this, we assume that at least a Zombie or a boss in already on the map if not it automatically shows that the player wins the game.

The 2 methods stillRunningWinEnding() and stillRunningLoseEnding() are used in conjunction as conditions in the while loop to run the game so that if one of the conditions return false, it means that the player either wins ,loses or quits the game.