Stories from the Gorbapork Nebula

Nick Bucher, Wood McGowan, and Kaden Goodell
10/20/16

Document History

Name	Reason	Date
Nick, Kaden, Wood	Initial Draft	09-16-16
Kaden	Added Decision Trees	10-18-16
Nick	Added Sequence Diagrams and Descriptions	10-18-16
Kaden	Added NPC Description for my NPCs	10-19-16
Wood	Added Class Diagrams and Descriptions	10-19-16
Nick	Added some Class Descriptions	10-19-16

Table of Contents

Section 1 - Requirements

Part 1.1: Introduction

1.1.1: Motivation

Our motivation behind choosing to develop a video game arose from a collective interest in video games, programming, and a desire to increase the visibility of video game developers at The University of Alabama. In following this motivation, we plan to produce a viable product that is fun to play and present to others. Furthermore, we intend for our project to showcase the Unity engine as an empowering tool for small teams developing games. We hope that our project will provide a concrete example of a fun, complete video game produced by students at The University of Alabama.

1.1.2: Purpose

The purpose of this project is to create a two dimensional role-playing video game that delivers an immersive and engaging narrative experience for players.

1.1.3: Scope

The completed game will have a start point and an end point, while also having optional content for the player to explore and experience. The game will have key role-playing elements such as an explorable environment, realistic and interactive characters, items which affect the player's stats in battle sequences, and items which play different roles in the story. The scope of this project will be limited in that the goal will be to squeeze as much fun gameplay and content into a reasonably small amount of playtime. In this way, the game should be able to be experienced and understood fairly quickly, and our project should be completed on schedule.

1.1.4: Goals

The goals of this game development project include producing a high functioning interactive graphical environment to represent the game world, and creating a highly reliable user control scheme that allows the user to seamlessly explore this environment. In creating an interactive world, our goal is to develop several unique and entertaining non-playable characters for the user to engage with, while also providing enjoyable dialogues and story arcs for the player to experience. The finished game should also provide a fun and well-designed battle system that functions with coherent and effective game mechanics, and that allows the user to be challenged and improve their skill.

1.1.5: Glossary

- **Game Engine:** A software development framework for creating video games.
- **Unity:** A game engine that supports development across many platforms with the C# programming language.
- **NPC:** A non-playable character in a video game
- **Assets:** Art models, audio effects, animations, or any element that is created outside of the game development environment in order to be utilized during development.
- **FPS:** The number of frames rendered every second.
- **Sprite:** A two dimensional graphical image.
- **Game State:** The state of all the game's elements and parameters at any given time during gameplay.
- **Colliders:** An area of space on the game screen that represents the shape and form of a game object in order to organize physical collisions on the screen.
- **Map:** The entire graphical representation of objects and space within the loaded scene.
- **GameObject:** A generic object in the Unity game engine.
- **The Main Game Loop:** The looped code sequence which updates the game each frame. In Unity, each GameObject calls its Update function every frame.
- **Coroutines:** A type of function in the Unity framework which acts as a new thread and can run asynchronously alongside the main game loop.
- **Prefab:** A GameObject which has been constructed in the Unity editor and saved to a file. This prefab object can then be referenced later in code.
- **Top-Down Perspective:** A graphical paradigm in which the the player object and the immediate surroundings are portrayed from above.
- **Shooter:** A broad genre of gameplay where the primary player activity is shooting projectiles at enemies.
- **360° Shooter:** A sub-genre of video games where the player can rotate 360° in order to shoot projectiles at enemies that approach from all directions. Usually implementing top-down perspective.
- **RPG:** A genre of video games characterized by narrative arcs, character dialogue, and world exploration in which the player assumes a fictional character's role in a fictional universe.
- **Battle System:** The algorithms, rules, and mechanics that determine the statistical system for a game's combat gameplay.
- **Play-Through:** A single completion of a video game's main story from start to finish.
- **Spawn:** The initialization of a graphical game object on the current map.
- **Replayability:** The degree to which a game is enjoyable to play through more than once.
- **Critical Path:** The sum of all the mandatory actions and events that must take place for the player to reach the end of the game from the initial starting point.

Part 1.2: Description

1.2.1: General Description

This project is a science fiction story based two dimensional role-playing video game. It is composed of a main storyline and multiple optional side quests. The main storyline begins with the main character, a space cadet in training, waking up in their room in a space station. The main character then does optional side quests by talking to different characters and exploring the space station, and then eventually goes to the space cadet training room to continue the main storyline. After doing the space cadet turret battle training, the main character then returns to their room and goes to sleep. Upon waking up, the main character realizes that their dog is missing. In searching for their dog on the space station, the main character may do more optional side quests. When the player finally finds their dog, the main storyline concludes with the space station being attacked by aliens. If there is sufficient development time, the Player will experience different endings based on which side quests they complete.

1.2.2: Gameplay Description

The gameplay entails exploring the current map, interacting with NPCs, discovering game items, and battling enemies. The battle system gameplay is both designed as a turret based shooter, as well as a top-down, 360-degree shooter where the player aims and shoots projectiles as enemies approach from all directions. Another component of the gameplay is interacting with the player statistical system in order to customize and improve a character's attributes, which is connected to the basic reward system for engaging in the game's battle system. Some portion of gameplay which involve item discovery and character exploration are intentionally designated as optional game scenarios, which the player is free to experience during their play-through of the main storyline.

1.2.3: User Control Description

The project implements a two-dimensional, top-down perspective, in which the user moves the player object around the map and navigates game menus using a joystick touchscreen GUI. The user interacts with game objects in the environment and selects player options in menus using a touchscreen button GUI. These ergonomic GUIs allow the gameplay to be controlled on a smartphone seamlessly without any need for the user to gain practice or develop motor skills.

1.2.4: Device Requirements

The device which the game will be running on will need to be running a version of Android 2.3.1 (Gingerbread) or higher to function. The pixel dependent user interface of the game will be scaled relative to the pixel density of the device it is on during runtime, so the screen size is independent from the functionality of the game. However, since the game is being developed for a screen ratio of 16:9, any screen ratios which are significantly different than 16:9 may have scaling issues with the user interface. The device which the game will be playing on will also need to have a touch screen for user input, and a large enough pixel resolution for the text displayed on the screen to be readable.

Part 1.3: Functional Requirements

1.3.1: System Startup Requirements

1. The system will load the title menu upon game launch.
2. The system will allow the user to start a new game from the title menu.
3. The system will allow the user to load a saved game state from the title menu and the pause menu.
4. The system will initialize the start state of the gameplay sequence upon the user creating a new game.
5. The system will properly load the player's saved game state upon the user loading a game.
6. The system will display the functional joystick and button interfaces on the screen during gameplay.

1.3.2: User Control Requirements

1. The system will allow the user to move the player object with the on-screen joystick interface during gameplay sequences which involve exploration.
2. The system will allow the user to interact with interactive game objects by utilizing the touch screen button interface.
3. The system will allow the player to select text options from selection menus during the proper game states using the touch screen button interface.
4. The system will allow the user to shoot the selected player weapon with the touch screen button interface during the game's battle sequences.
5. The system shall allow the user to aim the player weapon using the touch screen while in the game's battle state.

1.3.3: Graphical Map Requirements

1. The system shall load all the game objects that make up the graphical environments in the proper locations upon the user starting a new game or a saved game.
2. The system shall render the game object colliders in the proper spatial positions throughout the user's gameplay.
3. The system will allow the user to access the proper map locations dependant on the user's game state.
4. The system shall render the separate map sprites properly such that sprites that move will interact properly with the correct layers, and multiple sprites do not overlap on the same layer.

1.3.4: Game State Requirements

1. The system will load the correct menus as the user interacts with the specific game objects corresponding to those menus.
2. The system shall transition to the game's correct battle state at the proper times as the user engages enemies.
3. The system shall spawn and eliminate enemy game objects properly as the user triggers the battle state.
4. The system shall calculate health and damage parameters properly for all items and enemies as the user engages the battle system.
5. The system will allow the user to pause the game using the graphical pause button.
6. The system will freeze the current game state upon the user pausing.
7. The system shall allow the user to load a game state, save the current game state, quit the game, or resume the game state from the pause menu.
8. The system will execute the player death sequence when user reaches zero health.
9. The system will execute the endgame sequence when the user reaches the proper game state.

Part 1.4: Non-Functional Requirements

1.4.1: Optimization Requirements

- There will be a limit to the number of sprites on the screen.
- Input will be guaranteed to be responsive on most machines through extensive testing and script optimization.
- Script optimization will include moving cpu heavy code out of the game loop to coroutines where possible, so that the cpu heavy code will run at fixed time intervals as opposed to running every frame update.

1.4.2: Documentation Requirements

- Project documentation will be created using Visual Studio's XML syntax. This syntax allows for instant generation of intellisense, which will speed up the development of the game significantly.

- Documentation will be written for every function where the name of the function does not entirely describe what the function does.
- Code will be commented significantly, at least at every point where the developer has to research what a code segment does.

1.4.3: Coding Standards and Reusability Requirements

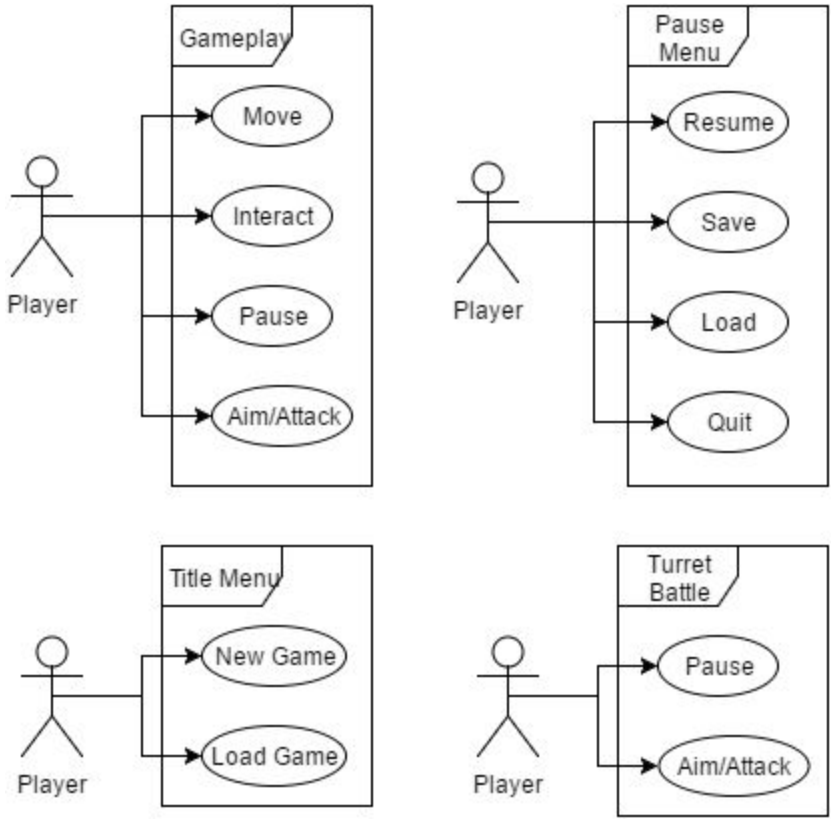
- The inversion of control design pattern will be applied by acquiring frequently used dependencies in a GameSceneController class in each scene, and referencing them through a single GameSceneController instance in each scene.
- Prefab GameObjects will be used to instantiate UI objects and other complicated objects, as suggested in the Unity documentation.
- If modules are identified as more code is written, they will be extracted to their own package as development continues. These extracted modules will be designed to be reusable in future projects.

1.4.4: Project Backup Requirements

- This project is in a git repository hosted on bitbucket.org.
- Each group member has their own branch.
- Any changes to the repository will be committed frequently.
- If an error is made which makes the game unplayable, the game will be able to be rolled back to its last playable commit.

Part 1.5: Use Case, Activity, and High Level Class Diagrams and Descriptions

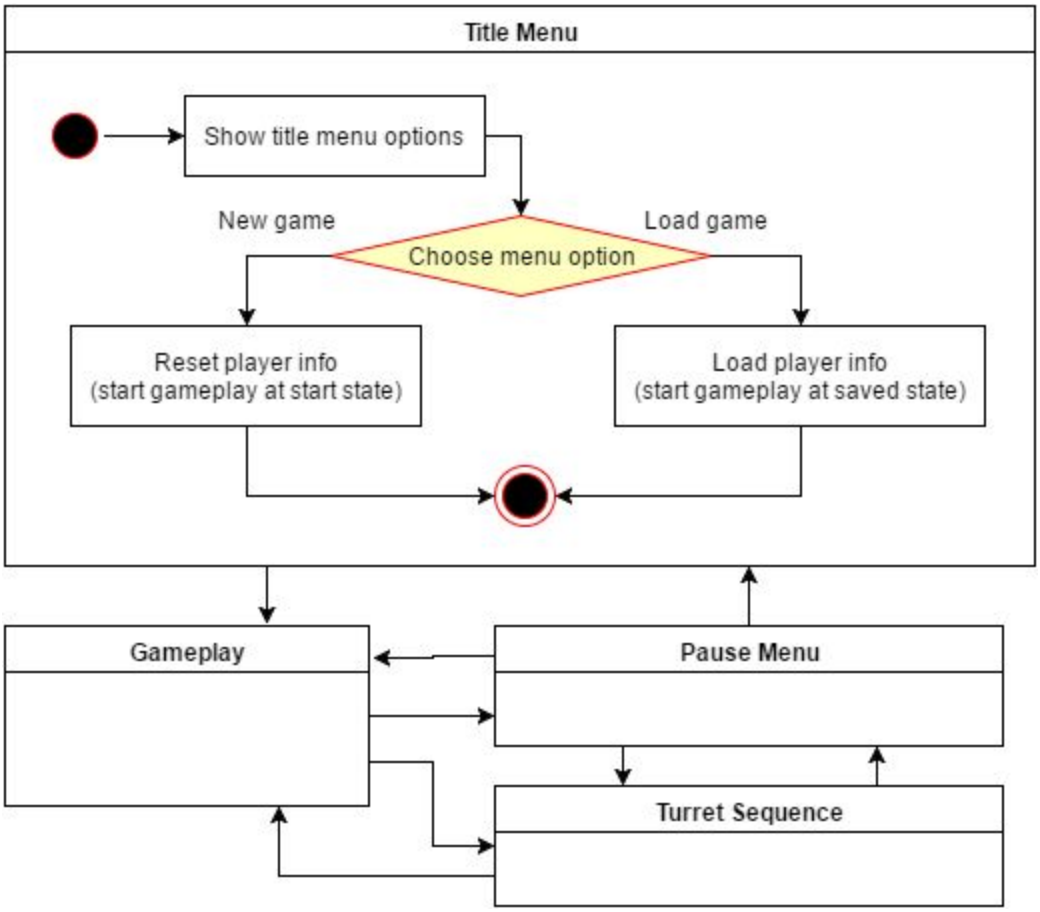
1.5.1: Use Case Diagrams



There are four use case scenarios in this project.

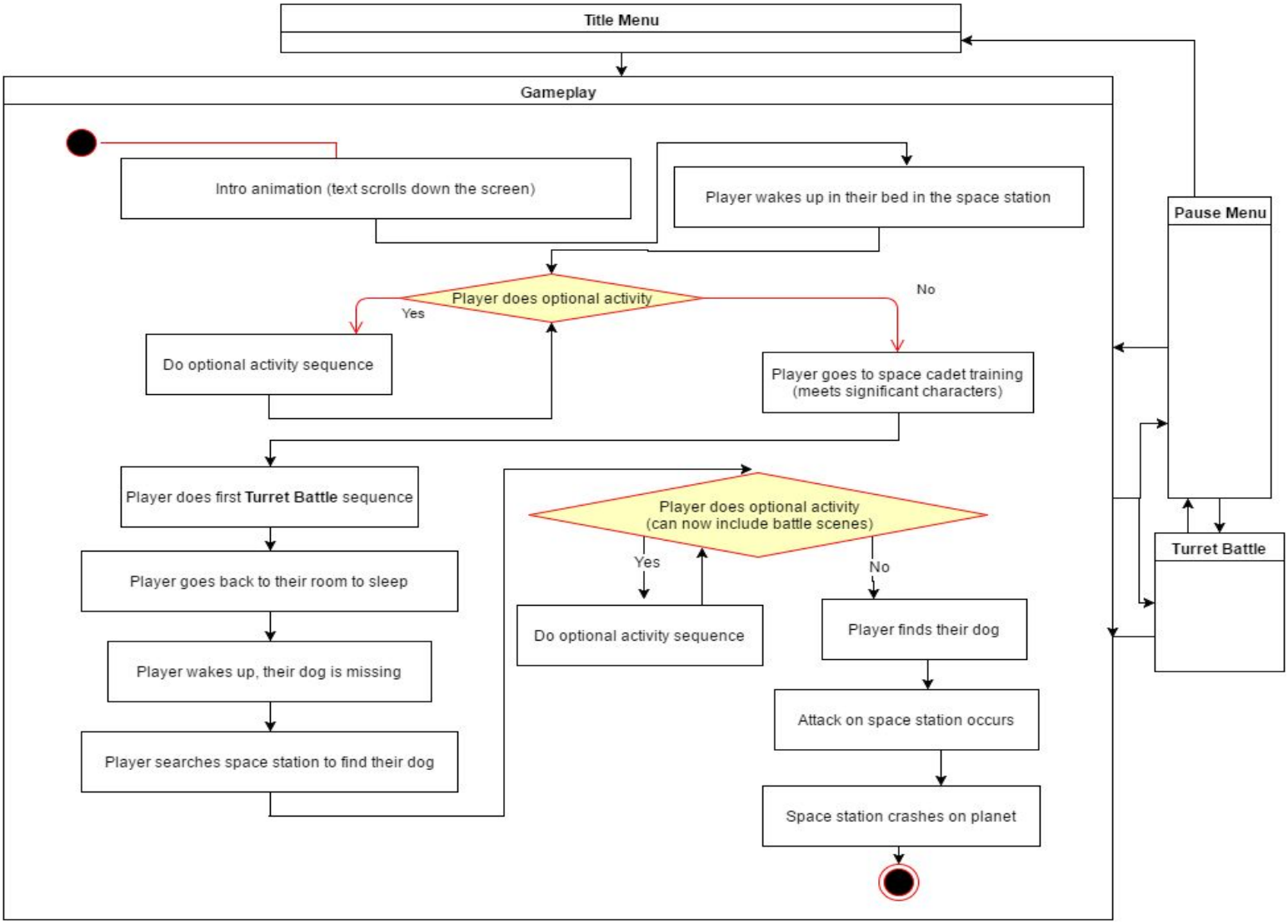
- The Gameplay scenario involves moving, interacting, pausing, and aiming/attacking. This scenario exists when the player is in full, direct control of the player GameObject.
- The Player can resume, save, load, or quit the game in the Pause Menu scenario. This scenario is loaded when the Player pauses the game.
- The Player can start a new game or load a game from the Title Menu scenario. This scenario is loaded whenever the game is opened.
- The Player can pause the game or aim/attack in the Turret Battle scenario. This scenario is loaded when the Player enters a turret battle.

1.5.2: Activity Diagrams



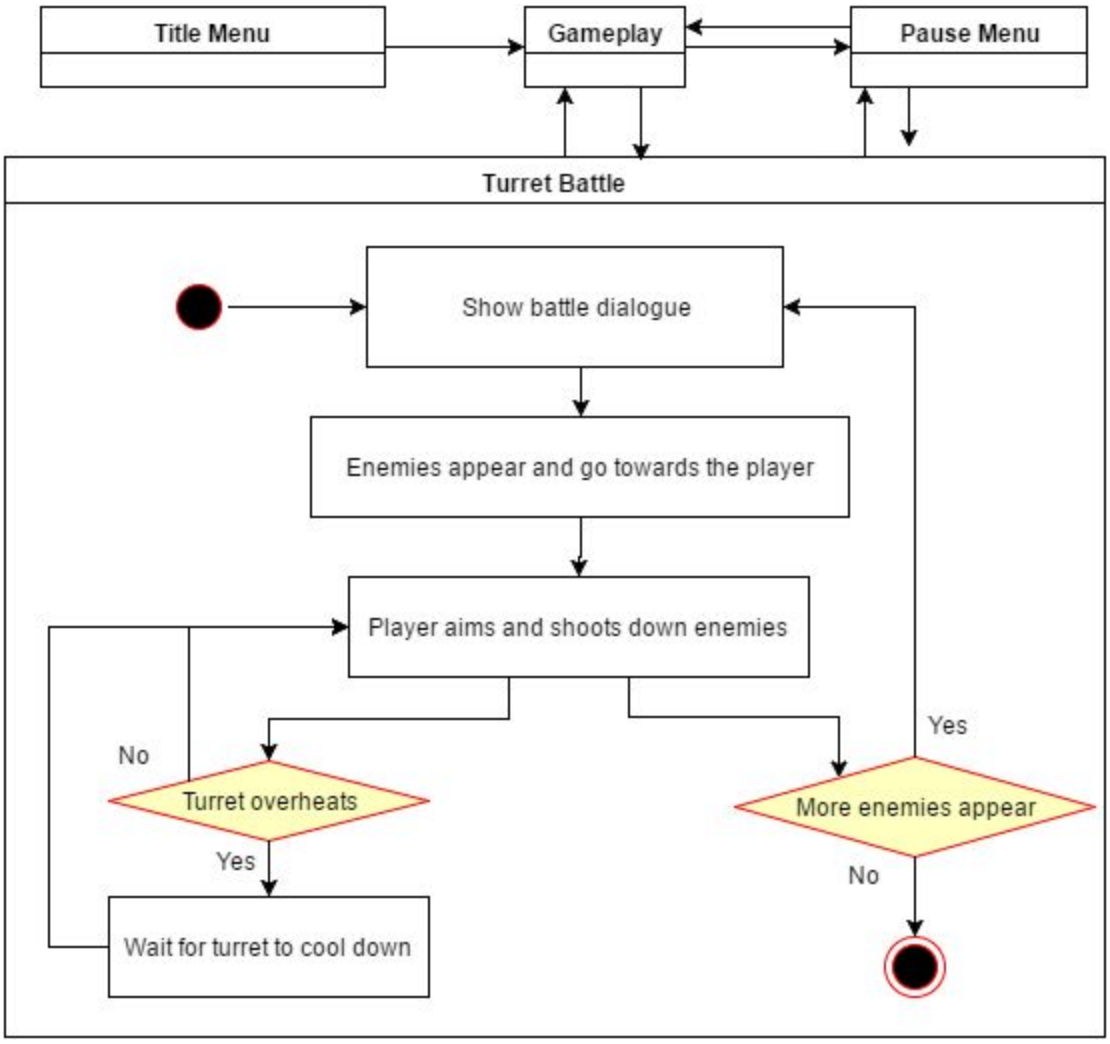
Title Menu

The Title Menu is loaded whenever the game is opened. When the Title Menu is loaded, the New Game and Load Game buttons are shown. If there are no saved game states, the Load Game button is disabled.



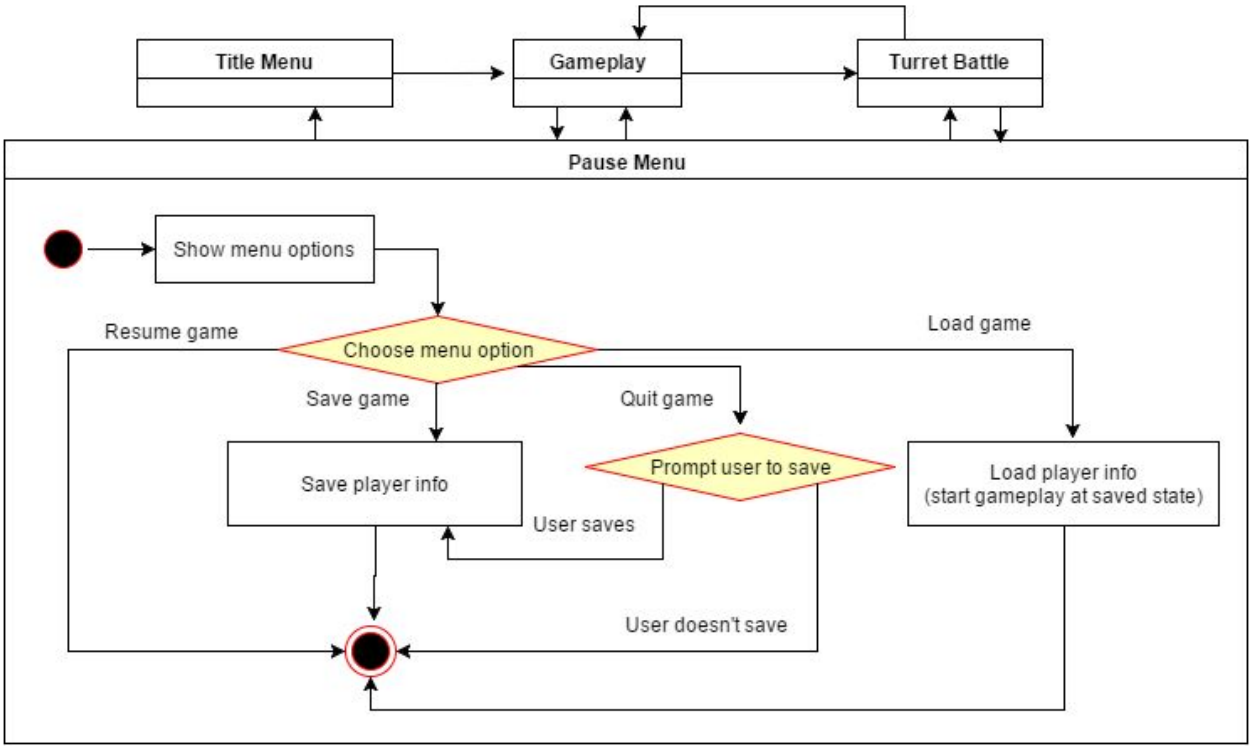
Gameplay

The initial state of the Gameplay sequence is loaded on the start of a new game. If a saved game state is loaded, the Gameplay sequence begins at that game state. The Player progresses through the Gameplay states as the story progresses. When the pause button is pressed, the Pause Menu sequence is loaded, and when the Player goes to space cadet training, the Turret Battle sequence is loaded.



Turret Battle

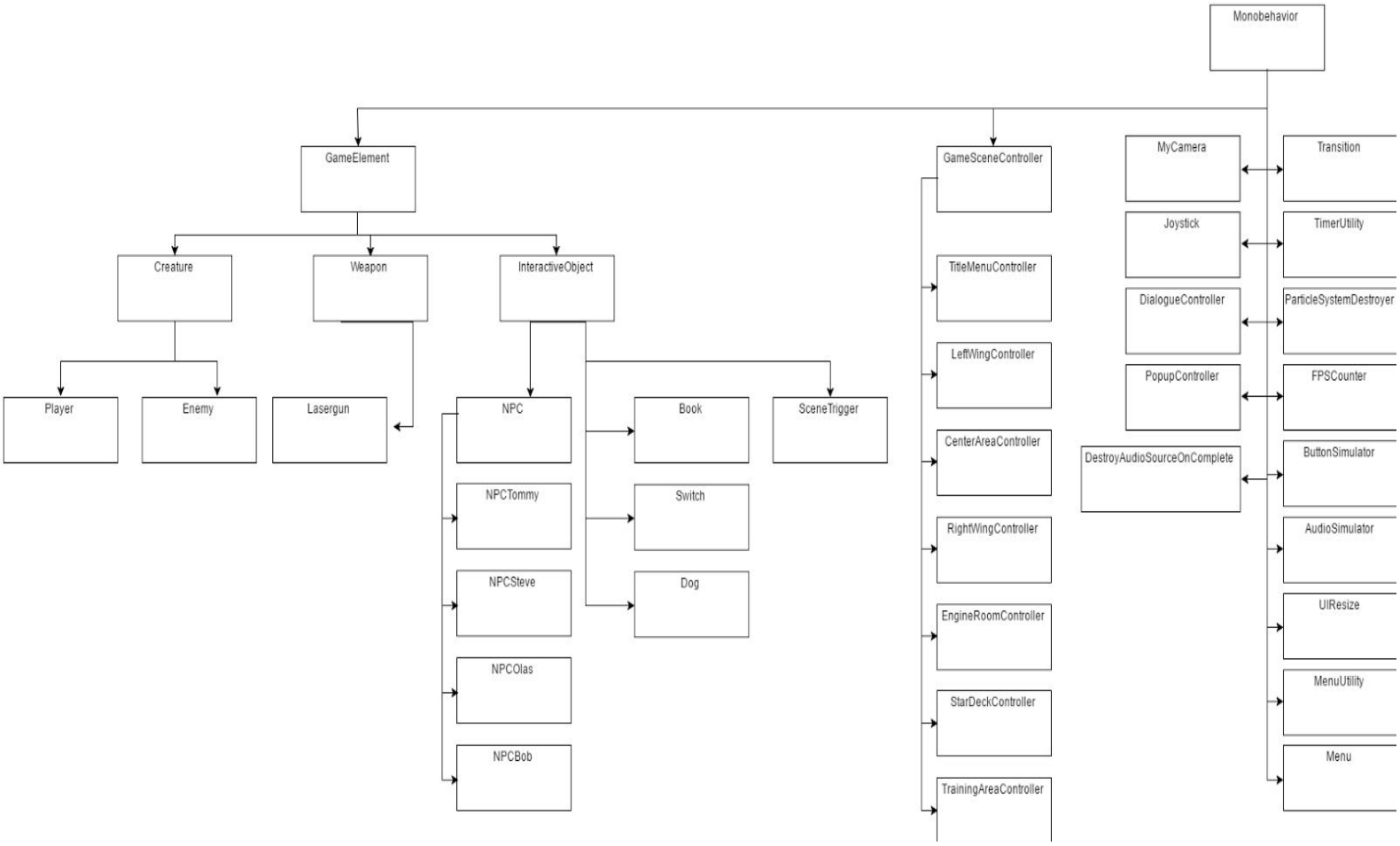
The Turret Battle sequence is loaded when the player gets to certain points in the Gameplay sequence. If the turret overheats while the Player is aiming and shooting down enemies, the Player must wait for the turret to cool down before continuing to shoot down enemies. When the pause button is pressed, the Pause Menu sequence is loaded. Also, when the Turret Battle sequence is complete, the Gameplay sequence is reloaded and the Player resumes the game at the state where the Turret Battle sequence was initialized.

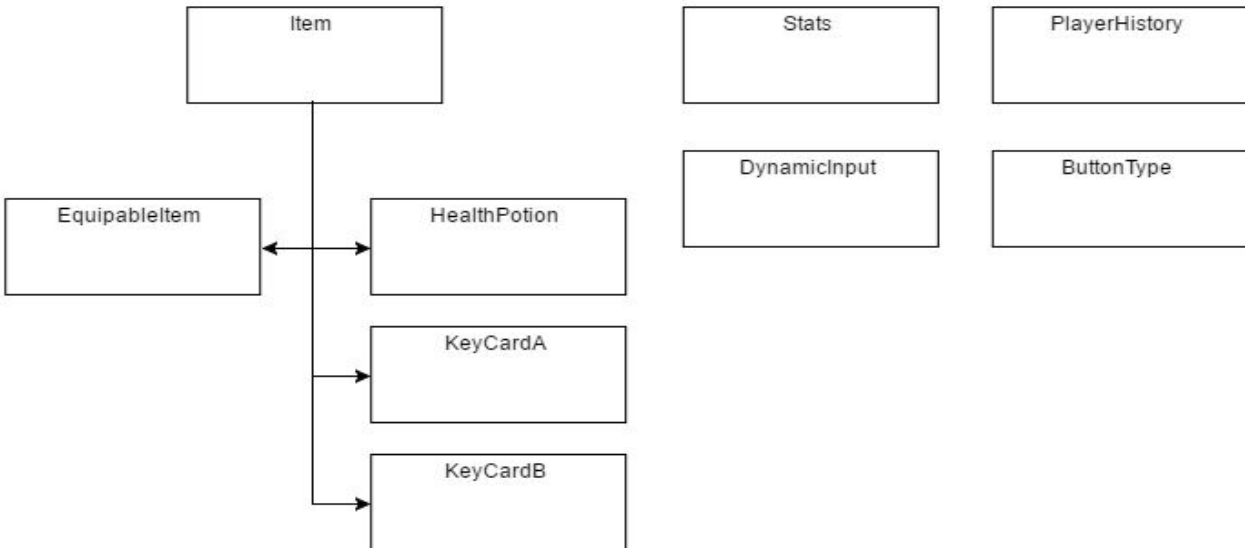


Pause Menu

The Pause Menu is loaded whenever the pause button is pressed during the Gameplay sequence or the Turret Battle sequence. When the Pause Menu is loaded, the Resume Game, Save Game, Quit Game, and Load Game buttons are shown. If there are no saved game states, the Load Game button is disabled. The Quit Game button prompts the user to save, then loads the Title Menu sequence.

1.5.3: High Level Class Diagrams





Class Sections

The classes in this project can be clearly divided into two sections. One section includes classes which inherit from Monobehaviour, and the other section includes classes which do not inherit from Monobehaviour. Monobehaviour is a class in the Unity api which is necessary for a class to inherit from if the class is to be a component of a GameObject. As a component of a GameObject, these classes may have fields exposed which can be changed and filled from the Unity inspector. This ability is useful in many cases. For example when instantiating menus in the MenuUtility class, GameObjects are created in the Unity editor and passed in as fields to classes which can instantiate them as needed. In this way, classes which inherit from Monobehaviour often deal directly with gameplay mechanics, and classes which do not inherit from Monobehaviour often represent other data.

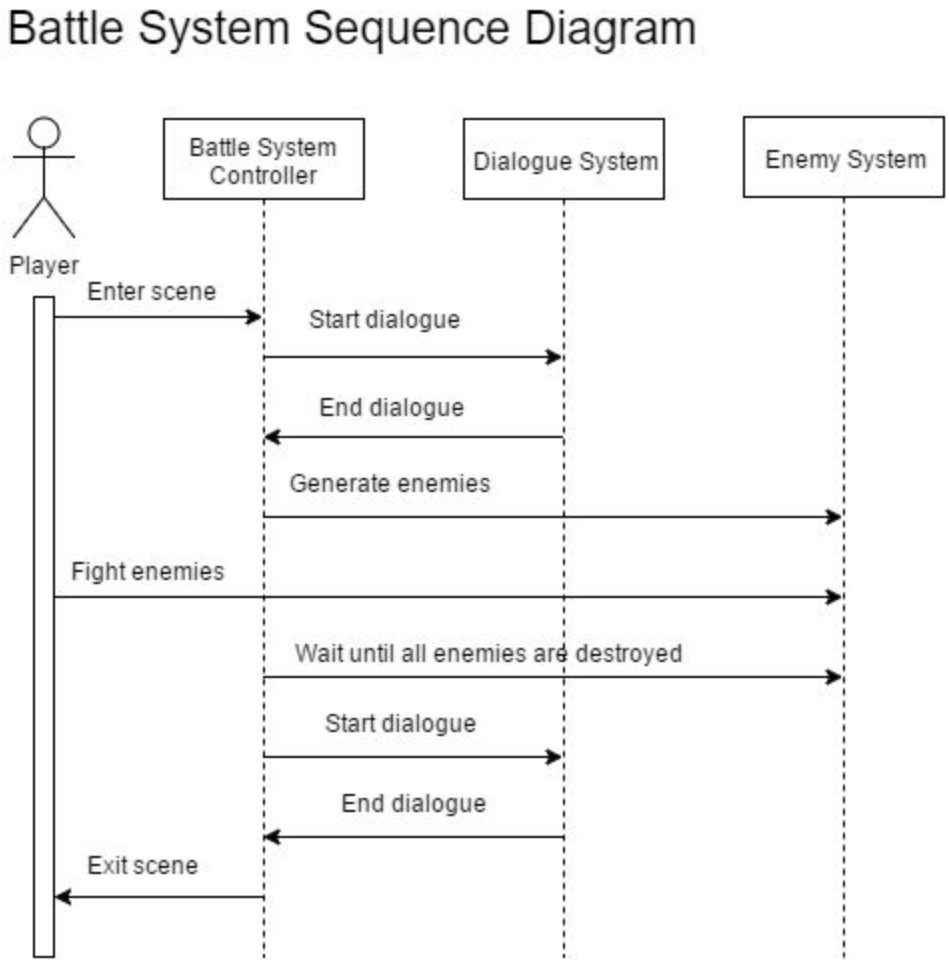
Important Classes

- **GameElement:** An abstract class which requires every subclass to have an elementName, elementIcon, and elementDescription.
- **Item:** An abstract class which represents an item, and requires an itemName, itemDescription, and itemIcon. This class is distinct from the GameElement class since Item objects do not have a physical representation in Unity as a GameObject (this is because Item does not inherit from Monobehaviour).
- **DynamicInput:** A wrapper class for Unity's Input class. This class is used instead of Unity's Input class to allow for different types of input for the same button.
- **GameSceneController:** An abstract class which contains references to other GameObjects (such as the Player GameObject). Each scene has a class that inherits from GameSceneController which contains the unique gameplay sequence for the scene.
- **InteractiveObject:** An abstract class which requires each subclass to implement the Interact function. This function is called whenever the Player is within a predefined distance from the InteractiveObject and presses a predefined button.

Section 2 - Design

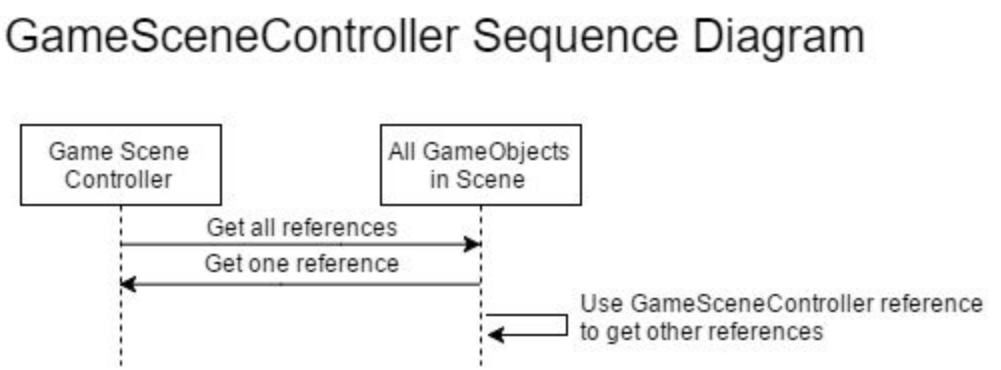
Part 2.1: Sequence and Detailed Class Diagrams and Descriptions

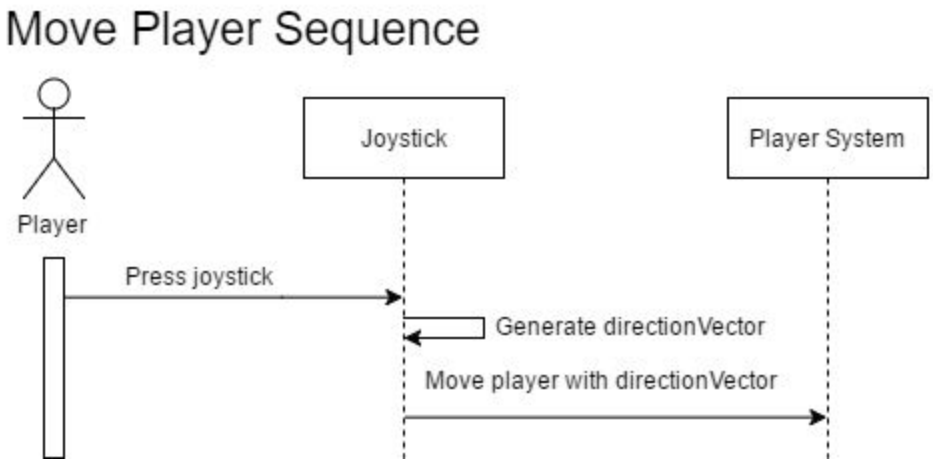
2.1.1: Sequence Diagrams



Battle System Sequence Diagram Description

The Battle System Controller (BSC) in this sequence diagram is an arbitrary implementation of the GameSceneController class for a battle scene. When the player enters a battle scene, the respective BSC for that scene initiates dialogue through the Dialogue System to give context for the battle. After the introduction to the battle is complete, the BSC generates enemies on the screen using an Enemy System, which the player can then fight. Once all enemies are destroyed, another dialogue is started and completed by the BSC, and then the scene transitions to exit the battle scene.



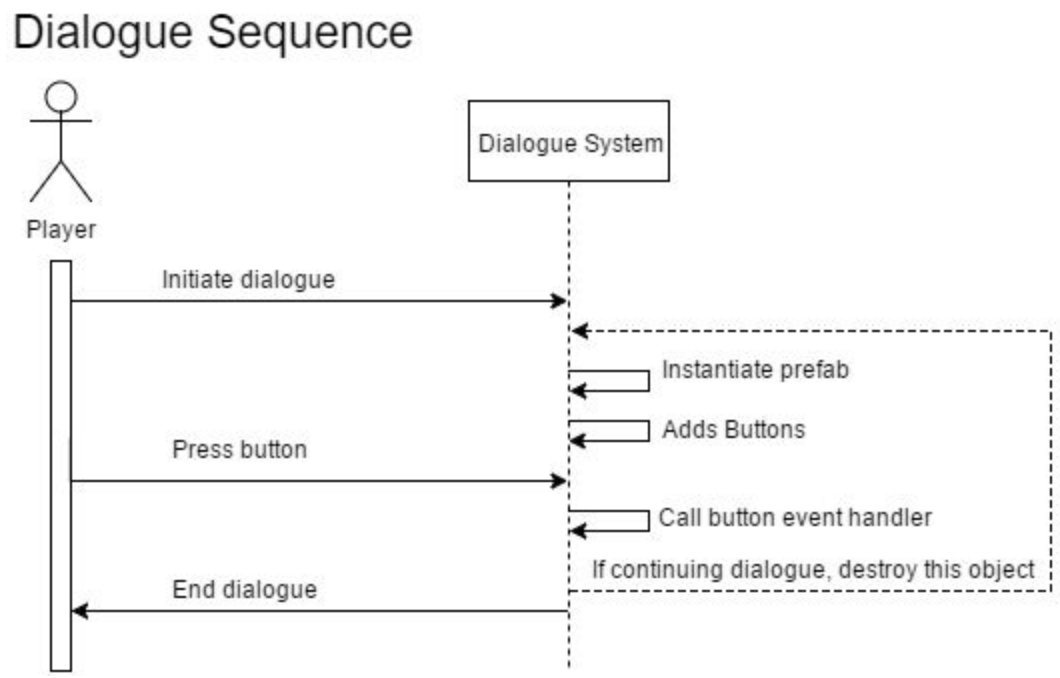


GameSceneController Sequence Diagram Description

The Game Scene Controller (GSC) sequence begins with the GSC for the active scene finding references to all the necessary GameObjects in the scene. There is one implementation of the GSC per scene. Once the GSC has references to all the GameObjects in the scene, each GameObject only needs one reference to the GSC to get a reference to any other GameObject in the scene, so each GameObject then finds a single reference to the GSC. This sequence utilizes the Inversion of Control design pattern to reduce the amount of references each class needs, since all dependency acquisition and error handling are handled in a single abstract class.

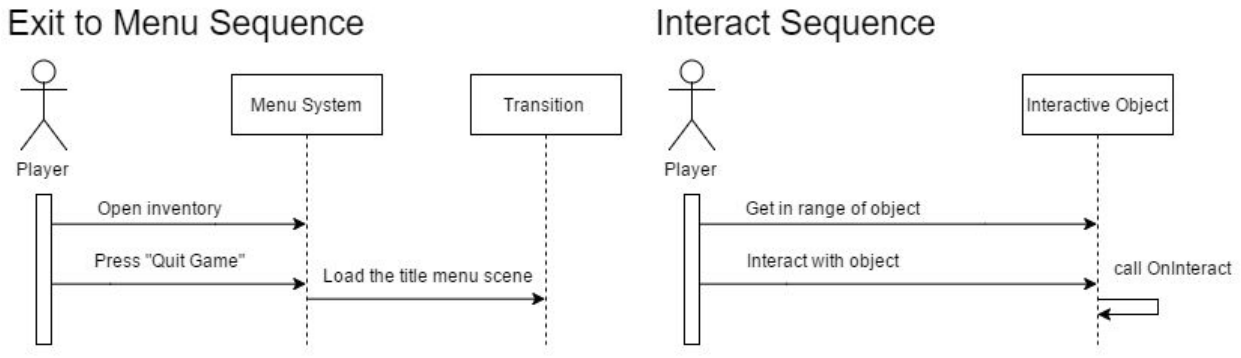
Move Player Sequence Diagram Description

There is a simulated joystick in the HUD of the game. When this joystick is pressed, a directionVector is generated by the Joystick class in the direction of the press. The magnitude of this directionVector is constrained to match the size of the sprite representation of the joystick. The directionVector is then used by the Player class to move the player object on the screen.



Dialogue Sequence Diagram Description

When dialogue is initiated by the player through any means, the dialogue system instantiates the dialogue prefab on the screen, and then adds button prefabs to it. These buttons have unique event handlers and labels. When the player presses a button on the screen, the event handler for that button is called. At this point the Dialogue System could either show a new dialogue by destroying the current dialogue object and creating a new one through the steps described above, or end the dialogue by simply destroying the current dialogue object. From the point dialogue is instantiated to the point dialogue ends, the game state of the GSC in the active scene is set to IN_DIALOGUE, and the button objects are disabled. This is to prevent the player from moving about while in dialogue.

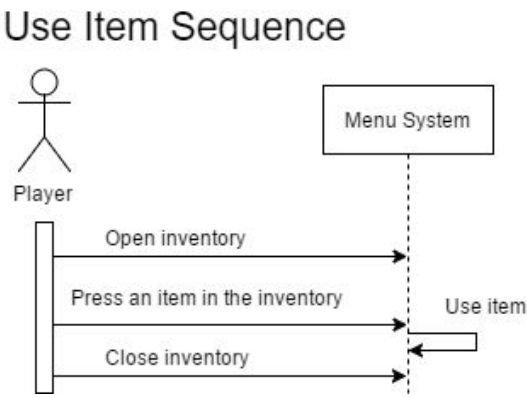


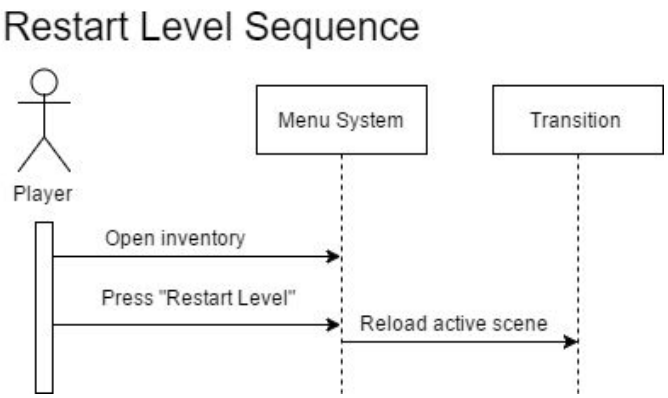
Exit to Menu Sequence Diagram Description

The player can access the “Quit Game” button by opening the inventory menu. This action pauses the game, and sets the game state of the GSC for the active scene to PAUSED. When the player presses the “Quit Game” button, the game resumes so that the title menu scene can be loaded, and the game state of the GSC for the active scene is set to PLAYING. The transition object then fades in so as to ease the transition between the active scene and the next scene, and the title menu scene is loaded.

Interact Sequence Diagram Description

When the player is in range of an Interactive Object (IO) and presses the A button to interact with the active IO, the Interact function of that IO is called. The Interact function in turn calls the abstract OnInteract function, which is unique for each IO implementation.



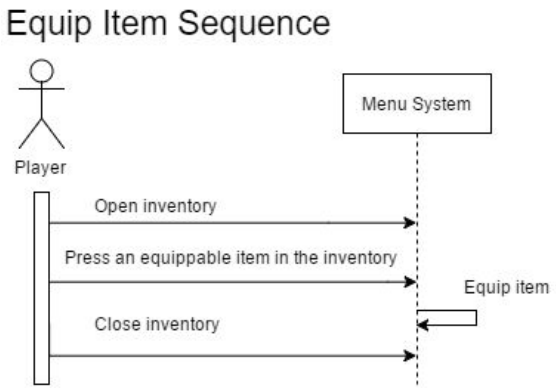


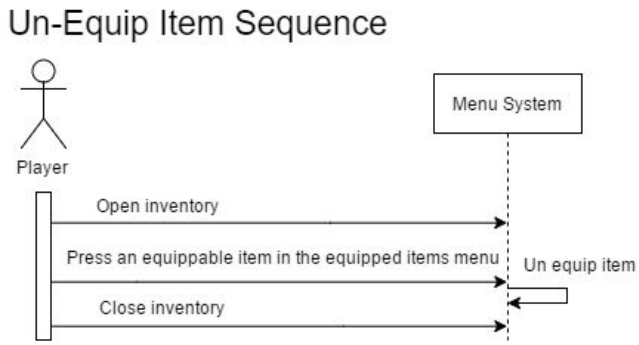
Use Item Sequence Diagram Description

The player can use an Item in the inventory by first opening the inventory menu. This action pauses the game, and sets the game state of the GSC for the active scene to PAUSED. When the player presses an Item in the inventory, the UseItem(Creature c) function is called on the selected Item with the Player class being passed in as the Creature to use the item on. When the inventory is closed, the game resumes and the game state of the GSC for the active scene is set to PLAYING.

Restart Level Sequence Diagram Description

The player can access the “Restart Level” button by opening the inventory menu. This action pauses the game, and sets the game state of the GSC for the active scene to PAUSED. When the player presses the “Restart Level” button, the game resumes so that the active scene can be reloaded, and the game state of the GSC for the active scene is set to PLAYING. The transition object then fades in so as to ease the transition between the active scene and the next scene, and the active scene is reloaded.





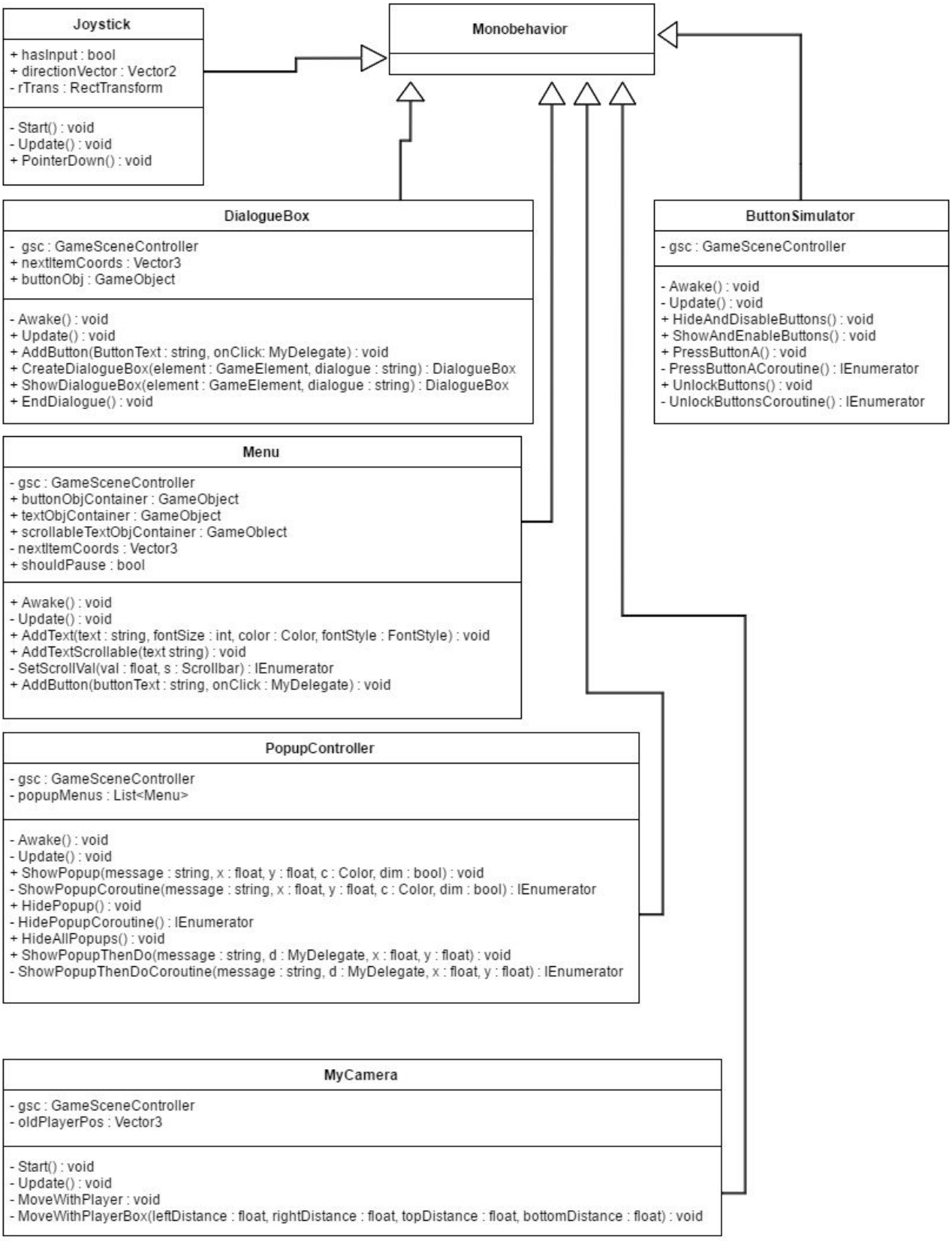
Equip Item Sequence Diagram Description

The player can equip an Equippable Item in the inventory by first opening the inventory menu. This action pauses the game, and sets the game state of the GSC for the active scene to PAUSED. When the player presses an Equippable Item in the inventory, the Equippable Item is removed from the Inventory and added to the Equipped Items List both in the Player class and in the UI. The UseItem(Creature c) function is then called on the selected Equippable Item with the Player class being passed in as the Creature to use the item on. The Equippable Item implementation of UseItem is empty, however this function can be overridden in derived classes so as to add additional functionality to specific Equippable Items. When the inventory is closed, the game resumes and the game state of the GSC for the active scene is set to PLAYING.

Un-Equip Item Sequence Diagram Description

The player can un-equip an Equippable Item in their equipped items by first opening the inventory menu. This action pauses the game, and sets the game state of the GSC for the active scene to PAUSED. When the player presses an Equippable Item in the equipped items, the Equippable Item is removed from the Equipped Items List and added to the Inventory both in the Player class and in the UI. When the inventory is closed, the game resumes and the game state of the GSC for the active scene is set to PLAYING.

2.1.2: Detailed Class Diagrams



Monobehavior: The abstract class in the Unity api which is necessary for a class to inherit from if the derived class is to be a component of a GameObject.

Joystick: Registers user input to the joystick UI control through the Unity UI system.

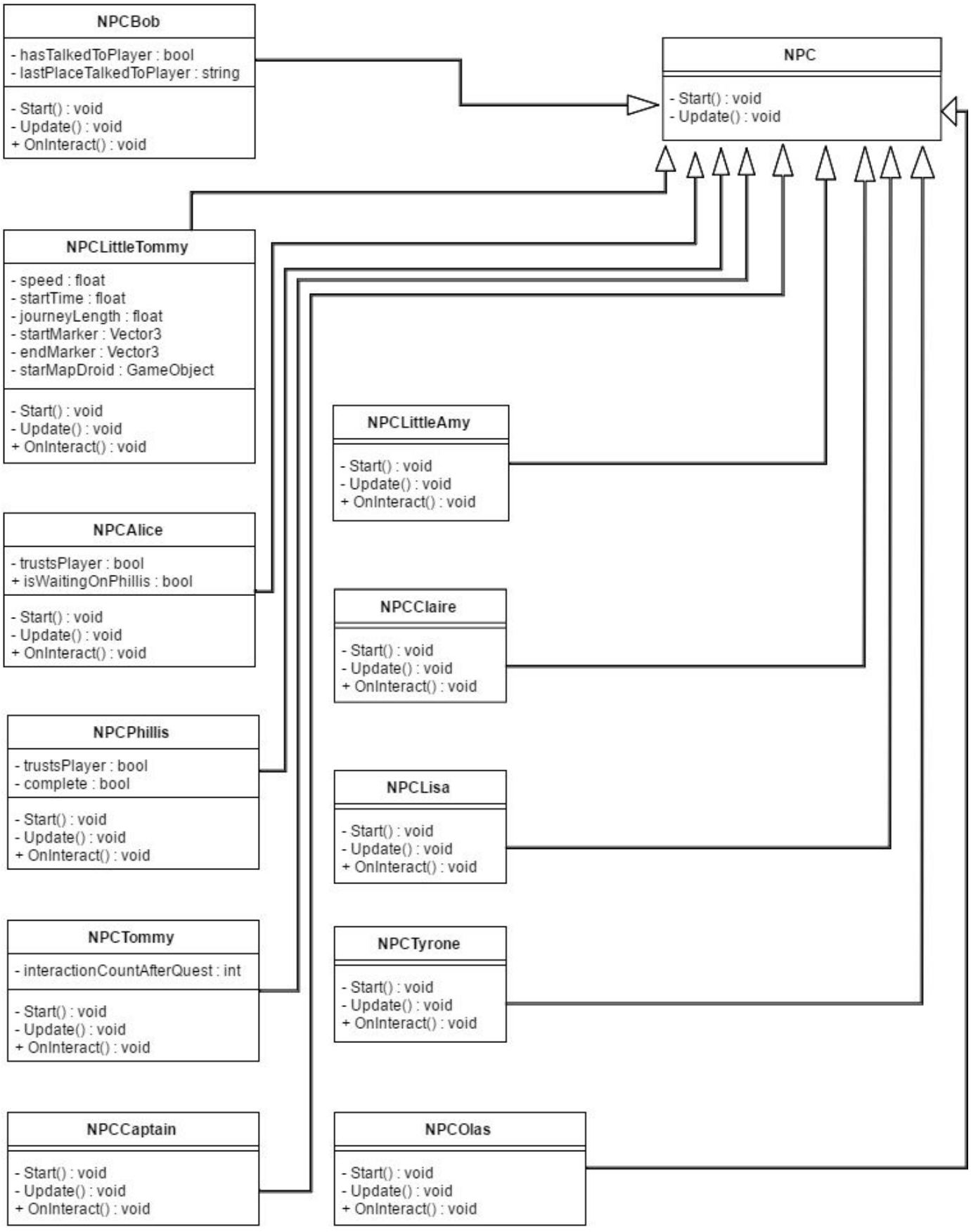
DialogueBox: A component on every instantiated dialogue object, the DialogueBox class is used to add buttons to a dialogue box and extend or end dialogue sequences.

ButtonSimulator: Registers user input to the button UI control through the Unity UI system.

Menu: A component on every instantiated menu GameObject, the Menu class is used to add text, scrollable text objects, buttons to menu GameObjects.

PopupController: Used to show quick popup messages to the user.

MyCamera: A component of the Main Camera GameObject, MyCamera makes the camera follow the player using a bounding box. The bounding box is defined in code as to not clutter up the Unity Editor view.


NPCBob
- hasTalkedToPlayer : bool
- lastPlaceTalkedToPlayer : string
- Start() : void
- Update() : void
+ OnInteract() : void
NPC
- Start() : void
- Update() : void
NPCLittleTommy
- speed : float
- startTime : float
- journeyLength : float
- startMarker : Vector3
- endMarker : Vector3
- starMapDroid : GameObject
- Start() : void
- Update() : void
+ OnInteract() : void
NPCLittleAmy
- Start() : void
- Update() : void
+ OnInteract() : void
NPCAlice
- trustsPlayer : bool
+ isWaitingOnPhillis : bool
- Start() : void
- Update() : void
+ OnInteract() : void
NPCClaire
- Start() : void
- Update() : void
+ OnInteract() : void
NPCPhillis
- trustsPlayer : bool
- complete : bool
- Start() : void
- Update() : void
+ OnInteract() : void
NPCLisa
- Start() : void
- Update() : void
+ OnInteract() : void
NPCTommy
- interactionCountAfterQuest : int
- Start() : void
- Update() : void
+ OnInteract() : void
NPCTyrone
- Start() : void
- Update() : void
+ OnInteract() : void
NPCCaptain
- Start() : void
- Update() : void
+ OnInteract() : void
NPCOlas
- Start() : void
- Update() : void
+ OnInteract() : void

NPCBob: A new recruit to the BUBs, Bob is found walking around Optimus 11 and talks to the player to help find his way around the space station.

NPCLittleTommy: Little Tommy is Tommy's son who lives on Optimus 11 with his father. Little Tommy has an abnormally high IQ and is very interested in all the technology on Optimus 11, which is why he is already knowledgeable enough to operate the BUB's Star Map Droid in the Left Wing.

NPCTommy: Within Tommy's innocent exterior, there lies an evil plot for galactic domination, however he has lost an old ring which is needed to complete his plan! If the player finds this ring, they can choose to give the ring back to Tommy and unknowingly aid in his pursuits, or deny Tommy the ring, and prevent a galactic power struggle.

NPCAlice: Alice is a BUB who was just rejected from a bouncer position on Planet M5337.

NPCPhillis: Phillis wanted to go to Planet M5337 with Alice, and is upset that Alice didn't get the bouncer position on that planet so she can't go with her there.

NPCCaptain: The Captain is the Commander of Optimus 11, and a retired starpilot.

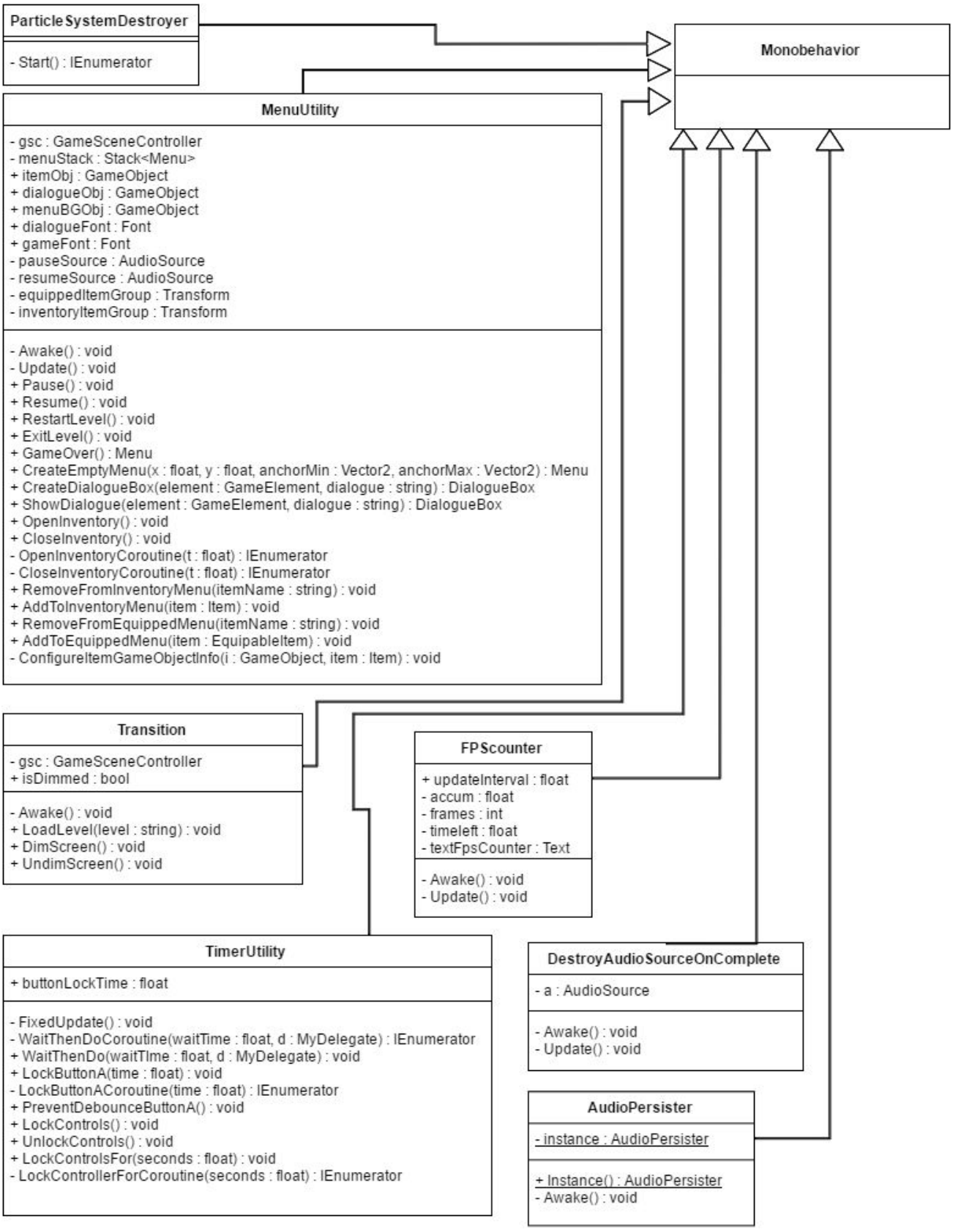
NPCTyrone: Tyrone is the Captain's Bodyguard and also does maintenance jobs around Optimus 11. Although his job is to intimidate those who may harm the captain, he is actually a very kind and wise individual.

NPCOlas: Olas is a salt-of-the-earth, Hungarian exchange recruit. Hard-working Olas is madly in love with the heartbreaking Lisa. Given his good nature, he hasn't expected a thing. But deep inside, Olas is not someone to be crossed. Should you choose to tell Olas about Lisa and the Captain's scandalous dealings, Bob will suddenly disappear from the game. I wonder where he went?

NPCLisa: Don't be fooled by Lisa's sweet exterior. She has a scandalous side that will be revealed by her journal entries. For 3 years, Lisa has been courted by Olas, the Hungarian exchange recruit. When you find the dirt, do you to tell Olas and ruin Lisa and the Captain's newly budding relationship? Or do you keep it to yourself and let them work it out?

NPCClaire: Claire is Lisa's overprotective roommate and a BUB cadet as well. She is also in charge of working with Tyrone on keeping the technology on Optimus 11 fully functional and up to date.

NPCLittleAmy: Little Amy is an eight year old orphan who was adopted by the Captain of Optimus 11 when she was five. Since then, she has traveled the galaxy on Optimus 11 learning about the universe from many BUB cadets and her best friend Little Tommy.


ParticleSystemDestroyer
- Start() : IEnumerator
Monobehavior
MenuUtility
- gsc : GameSceneController
- menuStack : Stack<Menu>
+ itemObj : GameObject
+ dialogueObj : GameObject
+ menuBGObj : GameObject
+ dialogueFont : Font
+ gameFont : Font
- pauseSource : AudioSource
- resumeSource : AudioSource
- equippedItemGroup : Transform
- inventoryItemGroup : Transform
- Awake() : void
- Update() : void
+ Pause() : void
+ Resume() : void
+ RestartLevel() : void
+ ExitLevel() : void
+ GameOver() : Menu
+ CreateEmptyMenu(x : float, y : float, anchorMin : Vector2, anchorMax : Vector2) : Menu
+ CreateDialogueBox(element : GameElement, dialogue : string) : DialogueBox
+ ShowDialogue(element : GameElement, dialogue : string) : DialogueBox
+ OpenInventory() : void
+ CloseInventory() : void
- OpenInventoryCoroutine(t : float) : IEnumerator
- CloseInventoryCoroutine(t : float) : IEnumerator
+ RemoveFromInventoryMenu(itemName : string) : void
+ AddToInventoryMenu(item : Item) : void
+ RemoveFromEquippedMenu(itemName : string) : void
+ AddToEquippedMenu(item : EquipableItem) : void
- ConfigureItemGameObjectInfo(i : GameObject, item : Item) : void
Transition
- gsc : GameSceneController
+ isDimmed : bool
- Awake() : void
+ LoadLevel(level : string) : void
+ DimScreen() : void
+ UndimScreen() : void
FPScounter
+ updateInterval : float
- accum : float
- frames : int
- timeleft : float
- textFpsCounter : Text
- Awake() : void
- Update() : void
TimerUtility
+ buttonLockTime : float
- FixedUpdate() : void
- WaitThenDoCoroutine(waitTime : float, d : MyDelegate) : IEnumerator
+ WaitThenDo(waitTIme : float, d : MyDelegate) : void
+ LockButtonA(time : float) : void
- LockButtonACoroutine(time : float) : IEnumerator
+ PreventDebounceButtonA() : void
+ LockControls() : void
+ UnlockControls() : void
+ LockControlsFor(seconds : float) : void
- LockControllerForCoroutine(seconds : float) : IEnumerator
DestroyAudioSourceOnComplete
- a : AudioSource
- Awake() : void
- Update() : void
AudioPersister
- instance : AudioPersister
+ Instance() : AudioPersister
- Awake() : void

Monobehavior: The abstract class in the Unity api which is necessary for a class to inherit from if the derived class is to be a component of a GameObject.

ParticleSystemDestroyer: A component which can be added to GameObjects with particle systems which destroys the GameObject once the particle system is complete.

MenuUtility: This class contains the dialogue creation, menu creation, and inventory management modules, which in the future may be refactored into their own classes. The dialogue creation functions return a DialogueBox class attached to the instantiated dialogue GameObject, the menu creation functions return a Menu class attached to the instantiated menu GameObject. The inventory management functions handle adding and removing UI item GameObjects to the inventory and equipped items menus on the inventory screen.

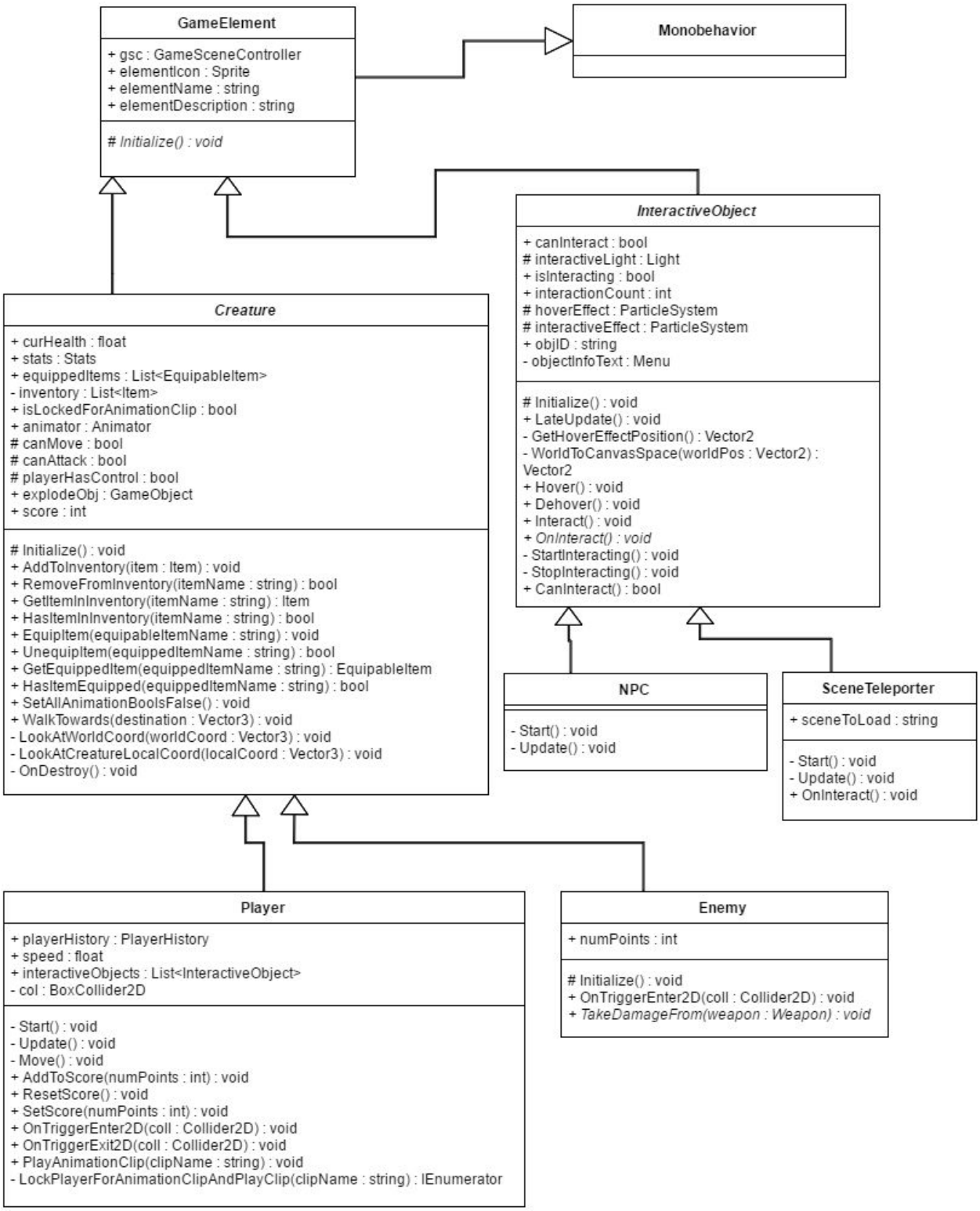
Transition: This class is used to load new scenes and fade the transition GameObject in and out between scenes to ease the transition.

FPSCounter: Included in the Unity Standard Assets, this class is used to profile frame rate performance.

TimerUtility: Used to delay the execution of delegates.

DestroyAudioSourceOnComplete: A component which can be added to GameObjects with audio sources which destroys the GameObject once the audio source is finished playing.

AudioPersister: A component which can be added to GameObjects which prevents them from being destroyed between scenes.


GameElement
+ gsc : GameSceneController
+ elementIcon : Sprite
+ elementName : string
+ elementDescription : string
# Initialize() : void
Monobehavior
InteractiveObject
+ canInteract : bool
# interactiveLight : Light
+ isInteracting : bool
+ interactionCount : int
# hoverEffect : ParticleSystem
# interactiveEffect : ParticleSystem
+ objID : string
- objectInfoText : Menu
Initialize() : void
+ LateUpdate() : void
- GetHoverEffectPosition() : Vector2
- WorldToCanvasSpace(worldPos : Vector2) : Vector2
+ Hover() : void
+ Dehover() : void
+ Interact() : void
+ OnInteract() : void
- StartInteracting() : void
- StopInteracting() : void
+ CanInteract() : bool
Creature
+ curHealth : float
+ stats : Stats
+ equippedItems : List<EquipableItem>
- inventory : List<Item>
+ isLockedForAnimationClip : bool
+ animator : Animator
# canMove : bool
# canAttack : bool
# playerHasControl : bool
+ explodeObj : GameObject
+ score : int
# Initialize() : void
+ AddToInventory(item : Item) : void
+ RemoveFromInventory(itemName : string) : bool
+ GetItemInInventory(itemName : string) : Item
+ HasItemInInventory(itemName : string) : bool
+ EquipItem(equipableItemName : string) : void
+ UnequipItem(equippedItemName : string) : bool
+ GetEquippedItem(equippedItemName : string) : EquipableItem
+ HasItemEquipped(equippedItemName : string) : bool
+ SetAllAnimationBoolsFalse() : void
+ WalkTowards(destination : Vector3) : void
- LookAtWorldCoord(worldCoord : Vector3) : void
- LookAtCreatureLocalCoord(localCoord : Vector3) : void
- OnDestroy() : void
NPC
- Start() : void
- Update() : void
SceneTeleporter
+ sceneToLoad : string
- Start() : void
- Update() : void
+ OnInteract() : void
Player
+ playerHistory : PlayerHistory
+ speed : float
+ interactiveObjects : List<InteractiveObject>
- col : BoxCollider2D
- Start() : void
- Update() : void
- Move() : void
+ AddToScore(numPoints : int) : void
+ ResetScore() : void
+ SetScore(numPoints : int) : void
+ OnTriggerEnter2D(coll : Collider2D) : void
+ OnTriggerExit2D(coll : Collider2D) : void
+ PlayAnimationClip(clipName : string) : void
- LockPlayerForAnimationClipAndPlayClip(clipName : string) : IEnumerator
Enemy
+ numPoints : int
# Initialize() : void
+ OnTriggerEnter2D(coll : Collider2D) : void
+ TakeDamageFrom(weapon : Weapon) : void

Monobehavior: The abstract class in the Unity api which is necessary for a class to inherit from if the derived class is to be a component of a GameObject.

GameElement: The abstract class which all Creatures, InteractiveObjects, and Weapons inherit from. GameElements contain a single reference to the GameSceneController of the active scene, and contain an ElementName, ElementDescription, and ElementIcon to describe each GameElement instance within the game.

Creature: This class inherits from GameElement and has functions related to inventory managment, equipping and unequipping items, and movement control. It also has Stats and current health fields to keep track of the status of the Creature.

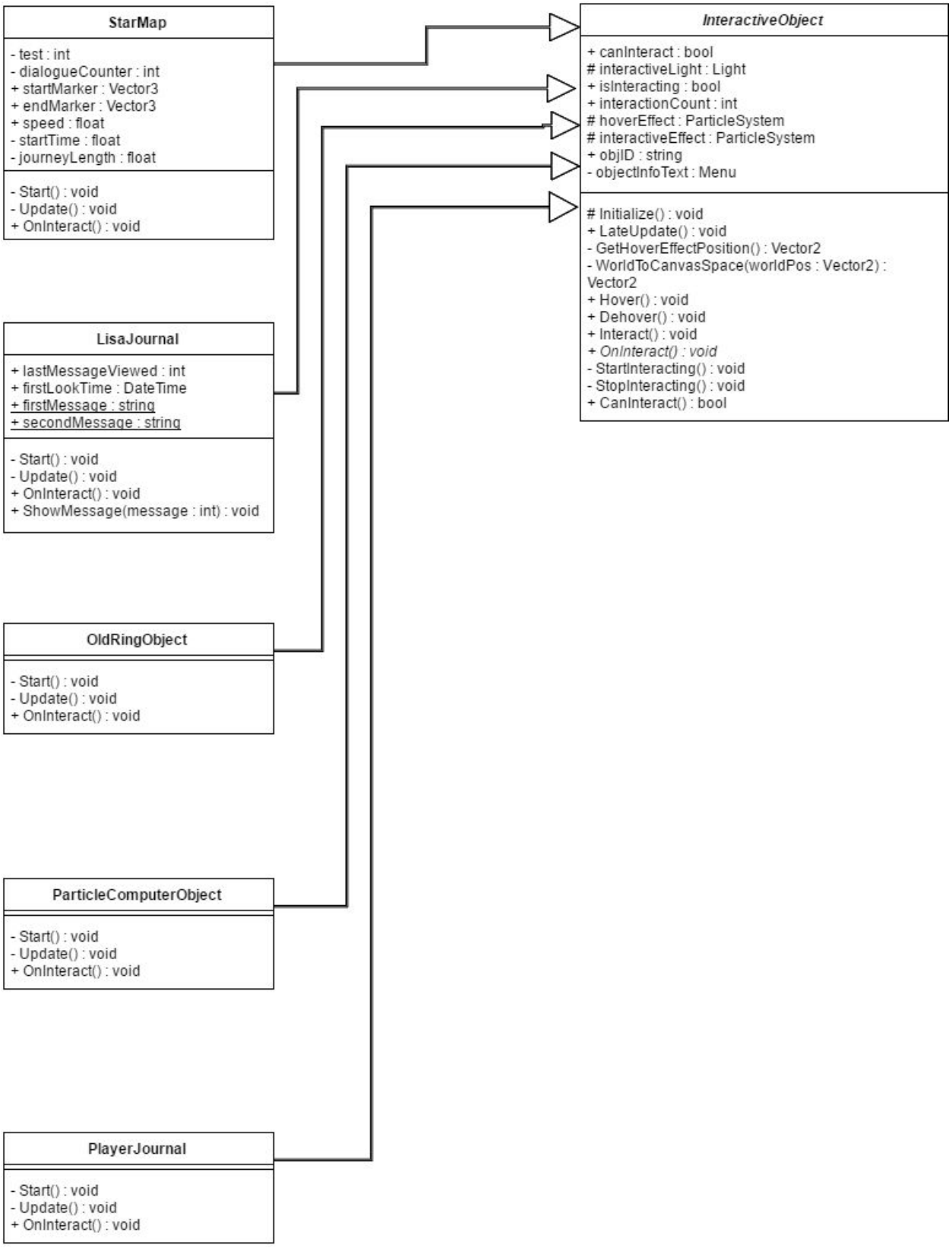
Player: This class inherits from Creature, and directly handles user interaction with InteractiveObjects, and movement based on user input.

Enemy: This class inherits from Creature, and will be used to implement AI controlled non-interactive enemies if there is time.

InteractiveObject: This class inherits from the GameElement class and represents game objects in the world that the player can interact with. The Interact function of the InteractiveObject class is called when the player approaches the object and pushes the action button. This Interact function will then call the OnInteract function in its derived class. When the Player is within range to interact with an InteractiveObject, the InteractiveObject will call the Hover function to display its ElementName and Element Description, and when the Player leaves the interactive range of the InteractiveObject, the DeHover function is called to remove the ElementName and ElementDescription from the scene.

NPC: This abstract class is used to distinguish NPCs from other interactive objects.

SceneTeleporter: The OnInteract function of this class uses the Transition object in the active scene to load the specified sceneToLoad scene.


StarMap
- test : int
- dialogueCounter : int
+ startMarker : Vector3
+ endMarker : Vector3
+ speed : float
- startTime : float
- journeyLength : float
- Start() : void
- Update() : void
+ OnInteract() : void
InteractiveObject
+ canInteract : bool
# interactiveLight : Light
+ isInteracting : bool
+ interactionCount : int
# hoverEffect : ParticleSystem
# interactiveEffect : ParticleSystem
+ objID : string
- objectInfoText : Menu
Initialize() : void
+ LateUpdate() : void
- GetHoverEffectPosition() : Vector2
- WorldToCanvasSpace(worldPos : Vector2) : Vector2
+ Hover() : void
+ Dehover() : void
+ Interact() : void
+ OnInteract() : void
- StartInteracting() : void
- StopInteracting() : void
+ CanInteract() : bool
LisaJournal
+ lastMessageViewed : int
+ firstLookTime : DateTime
+ firstMessage : string
+ secondMessage : string
- Start() : void
- Update() : void
+ OnInteract() : void
+ ShowMessage(message : int) : void
OldRingObject
- Start() : void
- Update() : void
+ OnInteract() : void
ParticleComputerObject
- Start() : void
- Update() : void
+ OnInteract() : void
PlayerJournal
- Start() : void
- Update() : void
+ OnInteract() : void

InteractiveObject: This class inherits from the GameElement class and represents game objects in the world that the player can interact with. The Interact function of the InteractiveObject class is called when the player approaches the object and pushes the action button. This Interact function will then call the OnInteract function in its derived class. When the Player is within range to interact with an InteractiveObject, the InteractiveObject will call the Hover function to display its ElementName and Element Description, and when the Player leaves the interactive range of the InteractiveObject, the DeHover function is called to remove the ElementName and ElementDescription from the scene.

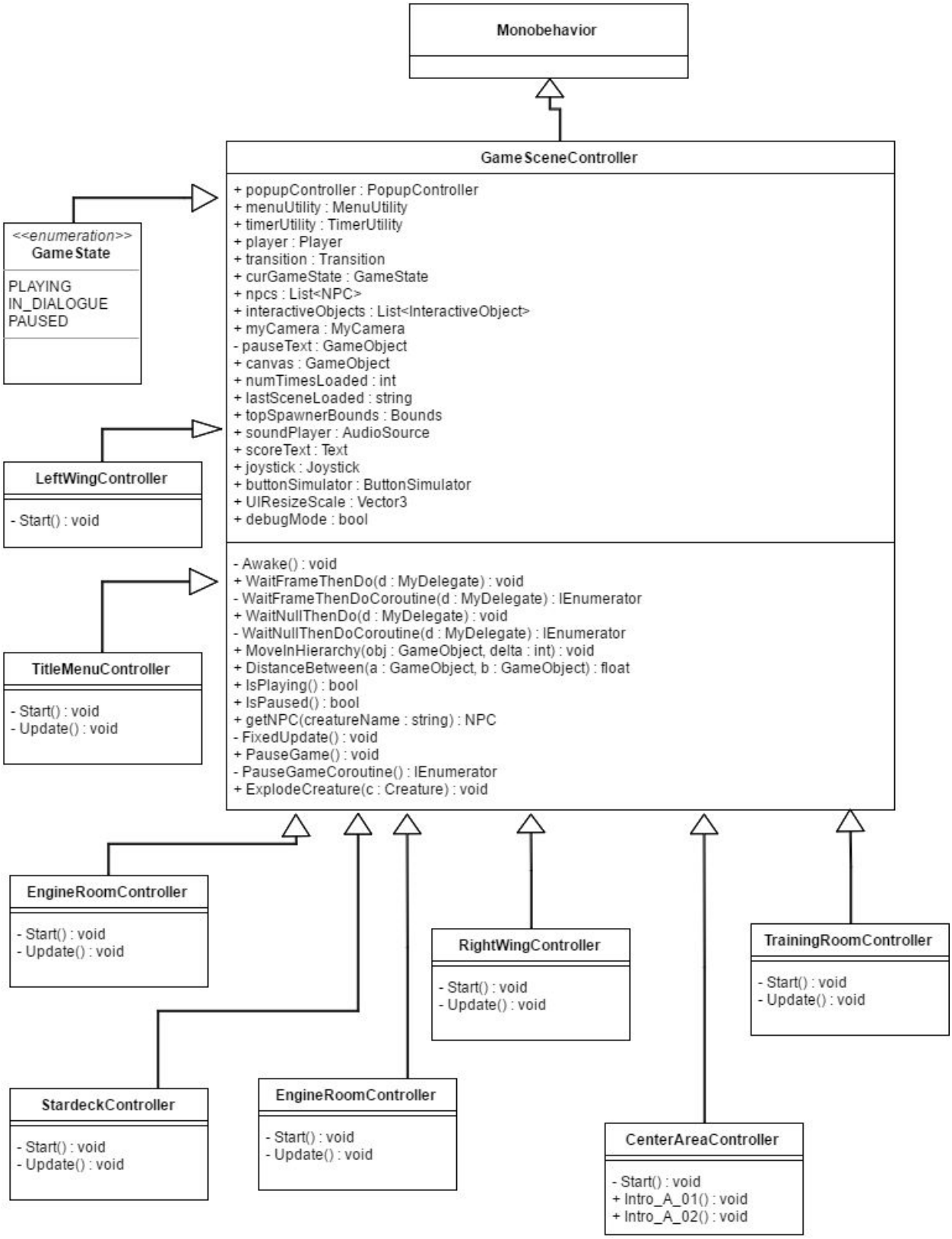
StarMap: This class controls the Star Map Droid in the Left Wing scene. This class inherits from InteractiveObject in order to script the player's interaction with the Star Map's game object, while also controlling the Star Map game object's movement around the scene.

LisaJournal: This class controls the LisaJournal game object in the Left Wing scene and inherits from InteractiveObject in order to interact with the player.

OldRingObject: This class controls the OldRing game object in the Left Wing scene and inherits from InteractiveObject in order to interact with the player. The OnInteract function on the OldRingObject destroys the gameObject of the OldRingObject, and adds a new OldRingItem to the Player's inventory.

ParticleComputerObject: This class controls the ParticleComputer game object in the Left Wing scene and inherits from InteractiveObject in order to interact with the player.

PlayerJournal: This class controls the PlayerJournal game object in the Left Wing scene and inherits from InteractiveObject in order to interact with the player.


Monobehavior
GameSceneController
+ popupController : PopupController
+ menuUtility : MenuUtility
+ timerUtility : TimerUtility
+ player : Player
+ transition : Transition
+ curGameState : GameState
+ npcs : List<NPC>
+ interactiveObjects : List<InteractiveObject>
+ myCamera : MyCamera
- pauseText : GameObject
+ canvas : GameObject
+ numTimesLoaded : int
+ lastSceneLoaded : string
+ topSpawnerBounds : Bounds
+ soundPlayer : AudioSource
+ scoreText : Text
+ joystick : Joystick
+ buttonSimulator : ButtonSimulator
+ UIResizeScale : Vector3
+ debugMode : bool
- Awake() : void
+ WaitFrameThenDo(d : MyDelegate) : void
- WaitFrameThenDoCoroutine(d : MyDelegate) : IEnumerator
+ WaitNullThenDo(d : MyDelegate) : void
- WaitNullThenDoCoroutine(d : MyDelegate) : IEnumerator
+ MoveInHierarchy(obj : GameObject, delta : int) : void
+ DistanceBetween(a : GameObject, b : GameObject) : float
+ IsPlaying() : bool
+ IsPaused() : bool
+ getNPC(creatureName : string) : NPC
- FixedUpdate() : void
+ PauseGame() : void
- PauseGameCoroutine() : IEnumerator
+ ExplodeCreature(c : Creature) : void
<<enumeration>>
GameState
PLAYING
IN_DIALOGUE
PAUSED
LeftWingController
- Start() : void
TitleMenuController
- Start() : void
- Update() : void
EngineRoomController
- Start() : void
- Update() : void
StardeckController
- Start() : void
- Update() : void
EngineRoomController
- Start() : void
- Update() : void
RightWingController
- Start() : void
- Update() : void
CenterAreaController
- Start() : void
+ Intro_A_01() : void
+ Intro_A_02() : void
TrainingRoomController
- Start() : void
- Update() : void

Monobehavior: The abstract class in the Unity api which is necessary for a class to inherit from if the derived class is to be a component of a GameObject.

GameSceneController: This class finds references to all necessary GameObjects in the active scene, and then controls the unique gameplay flow of each scene within derived classes. There is one implementation of the GSC per scene.

GameState: An enumeration that tracks the state of the current game within the GameSceneController.

TitleMenuController: This class inherits from GameSceneController and controls the unique gameplay flow for the Title Menu scene.

LeftWingController: This class inherits from GameSceneController and controls the unique gameplay flow for the Left Wing scene.

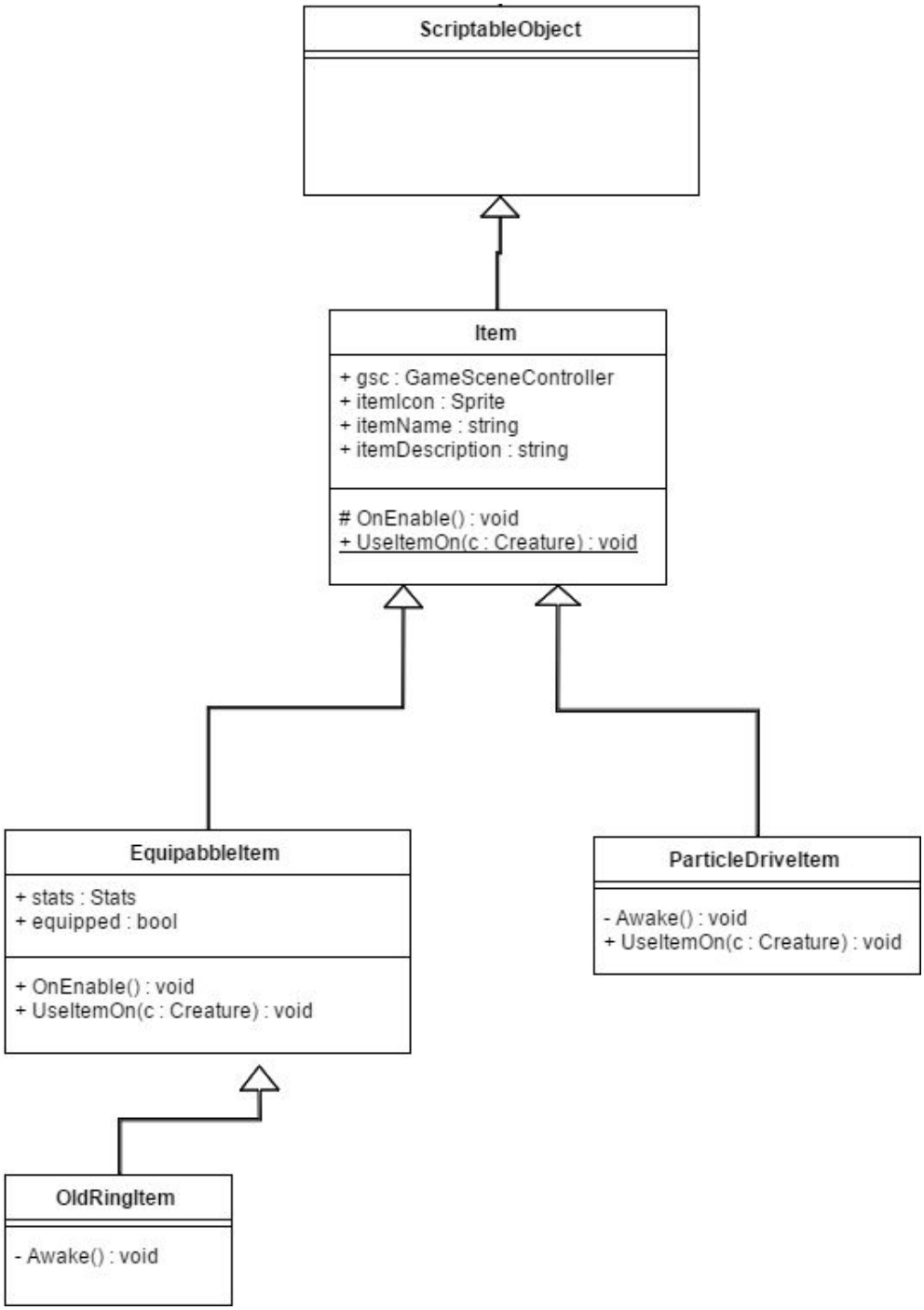
RightWingController: This class inherits from GameSceneController and controls the unique gameplay flow for the Right Wing scene.

CenterAreaController: This class inherits from GameSceneController and controls the unique gameplay flow for the Center Area scene.

TrainingRoomController: This class inherits from GameSceneController and controls the unique gameplay flow for the Training Room scene.

EngineRoomController: This class inherits from GameSceneController and controls the unique gameplay flow for the Engine Room scene.

StardeckController: This class inherits from GameSceneController and controls the unique gameplay flow for the Stardeck scene.


ScriptableObject
Item
+ gsc : GameSceneController
+ itemIcon : Sprite
+ itemName : string
+ itemDescription : string
# OnEnable() : void
+ UseItemOn(c : Creature) : void
EquipabbleItem
+ stats : Stats
+ equipped : bool
+ OnEnable() : void
+ UseItemOn(c : Creature) : void
ParticleDriveItem
- Awake() : void
+ UseItemOn(c : Creature) : void
OldRingItem
- Awake() : void

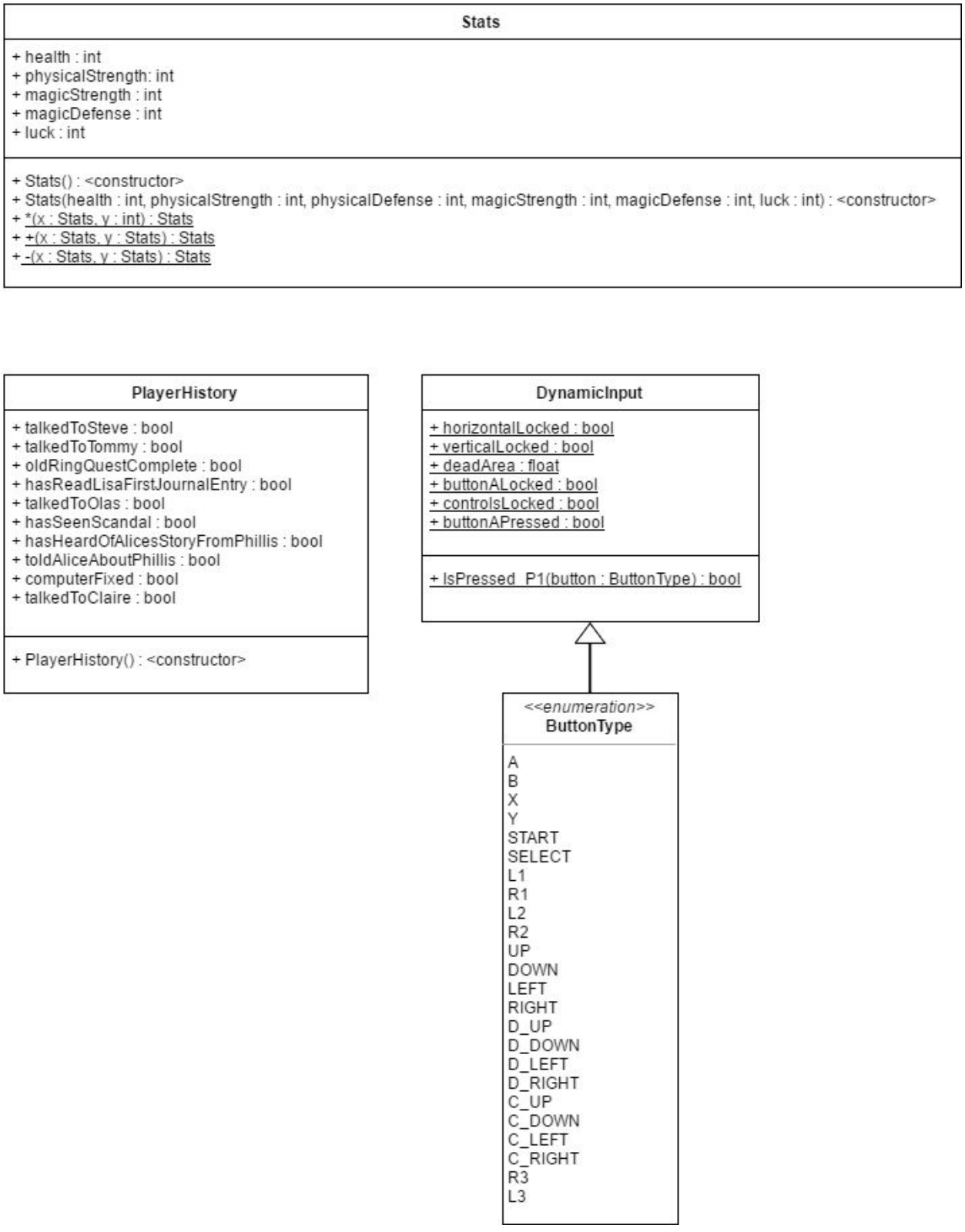
ScriptableObject: The abstract class in the Unity api which is necessary for a class to inherit from if the derived class is not the component of a GameObject but is still instantiated and used to interact with other objects in the game during runtime.

Item: This class inherits from ScriptableObject and contains an itemName, itemDescription, and itemIcon. When a Creature uses an item, the Creature calls the UseItem(Creature c) function on the Item, passing itself as a parameter to the function.

EquipabbleItem: A special type of Item which may be equipped and unequipped by Creature objects. Each EquipabbleItem has a Stats field which is added to the Creature's Stats field when the EquipabbleItem is equipped, and subtracted from the Creature's Stats field when the EquippableItem is unequipped.

ParticleDriveItem: This class inherits from the Item class, and stores a unique itemName, itemDescription, and itemIcon to be displayed on the inventory screen.

OldRingItem: This class inherits from the EquipabbleItem class, and stores a unique itemName, itemDescription, and itemIcon to be displayed on the inventory screen.


Stats
+ health : int
+ physicalStrength: int
+ magicStrength : int
+ magicDefense : int
+ luck : int
+ Stats() : <constructor>
+ Stats(health : int, physicalStrength : int, physicalDefense : int, magicStrength : int, magicDefense : int, luck : int) : <constructor>
+ *(x : Stats, y : int) : Stats
+ +(x : Stats, y : Stats) : Stats
+ -(x : Stats, y : Stats) : Stats
PlayerHistory
+ talkedToSteve : bool
+ talkedToTommy : bool
+ oldRingQuestComplete : bool
+ hasReadLisaFirstJournalEntry : bool
+ talkedToOlas : bool
+ hasSeenScandal : bool
+ hasHeardOfAlicesStoryFromPhillis : bool
+ toldAliceAboutPhillis : bool
+ computerFixed : bool
+ talkedToClaire : bool
+ PlayerHistory() : <constructor>
DynamicInput
+ horizontalLocked : bool
+ verticalLocked : bool
+ deadArea : float
+ buttonALocked : bool
+ controlsLocked : bool
+ buttonAPressed : bool
+ IsPressed_P1(button : ButtonType) : bool
<<enumeration>>
ButtonType
A
B
X
Y
START
SELECT
L1
R1
L2
R2
UP
DOWN
LEFT
RIGHT
D_UP
D_DOWN
D_LEFT
D_RIGHT
C_UP
C_DOWN
C_LEFT
C_RIGHT
R3
L3

Stats: A class used to contain and represent statistics of Creature objects. The “+”, “-”, and “*” operators of this class are overloaded. The implementation of this operator overloading allows for instances of the Stats class to be easily added together, subtracted from each other, or multiplied by a number.

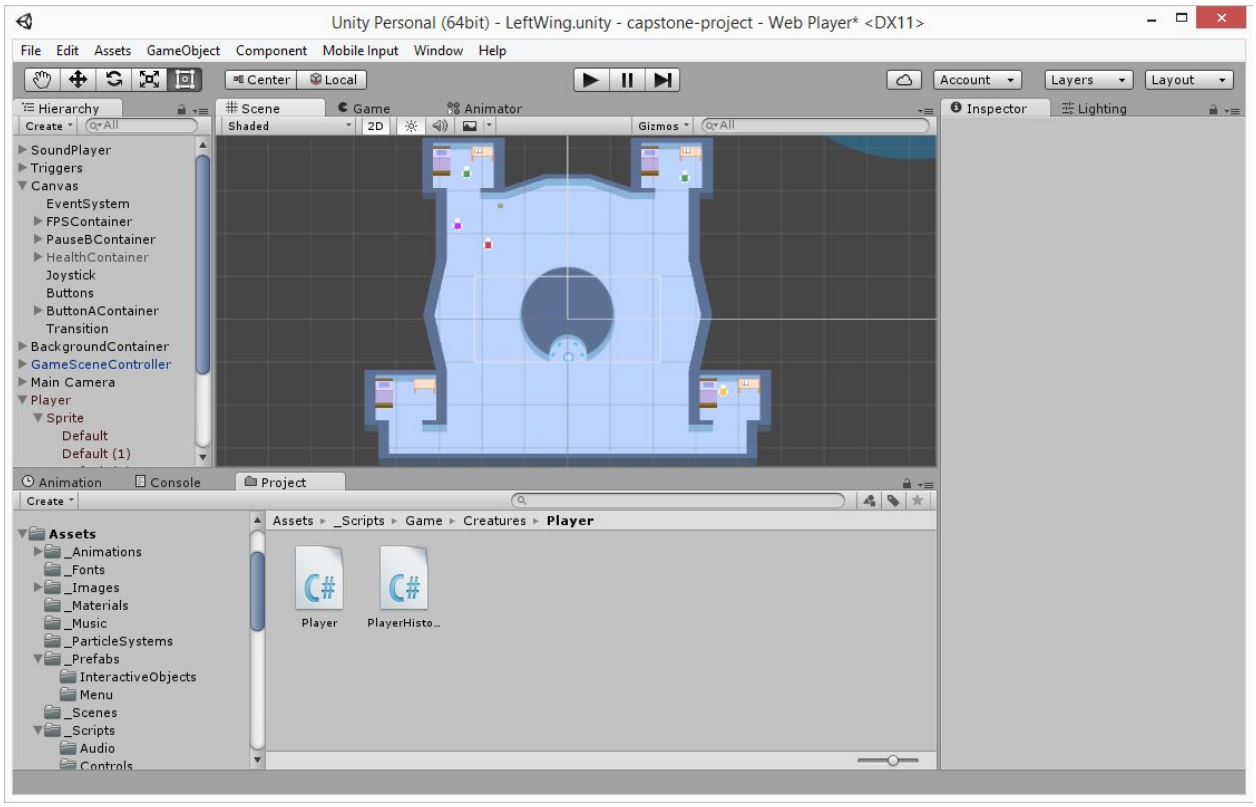
PlayerHistory: This class keeps track of which notable events the player has experienced through a list of modifiable fields organized in “regions” within Visual Studio.

DynamicInput: This class contains the function, IsPressedP1(ButtonType b), which returns whether the accepted input key(s) for the corresponding ButtonType have been pressed by the user for the specified ButtonType. This class acts as a wrapper for the Input class built into Unity, allowing for more than one key to be mapped to a single ButtonType.

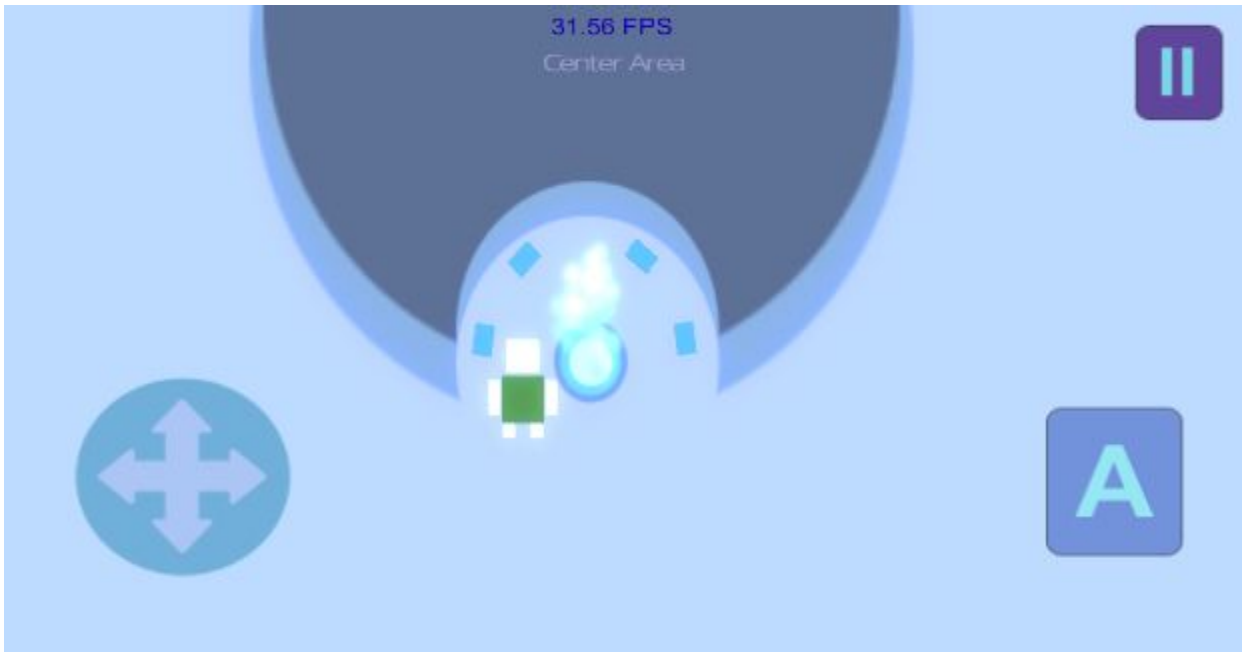
ButtonType: This enumeration contains the IDs of each potential button type.

Part 2.2: Screenshots, Links, Dialogue Trees, and Descriptions

2.2.1: Project Screenshots



Creating a Map in Unity and populating the map with sprites for game objects



Gameplay screenshot of a player and a teleporter GameObject

Early Dialogue Example of player interacting with an NPC GameObject

2.2.2: Project Demonstration Links

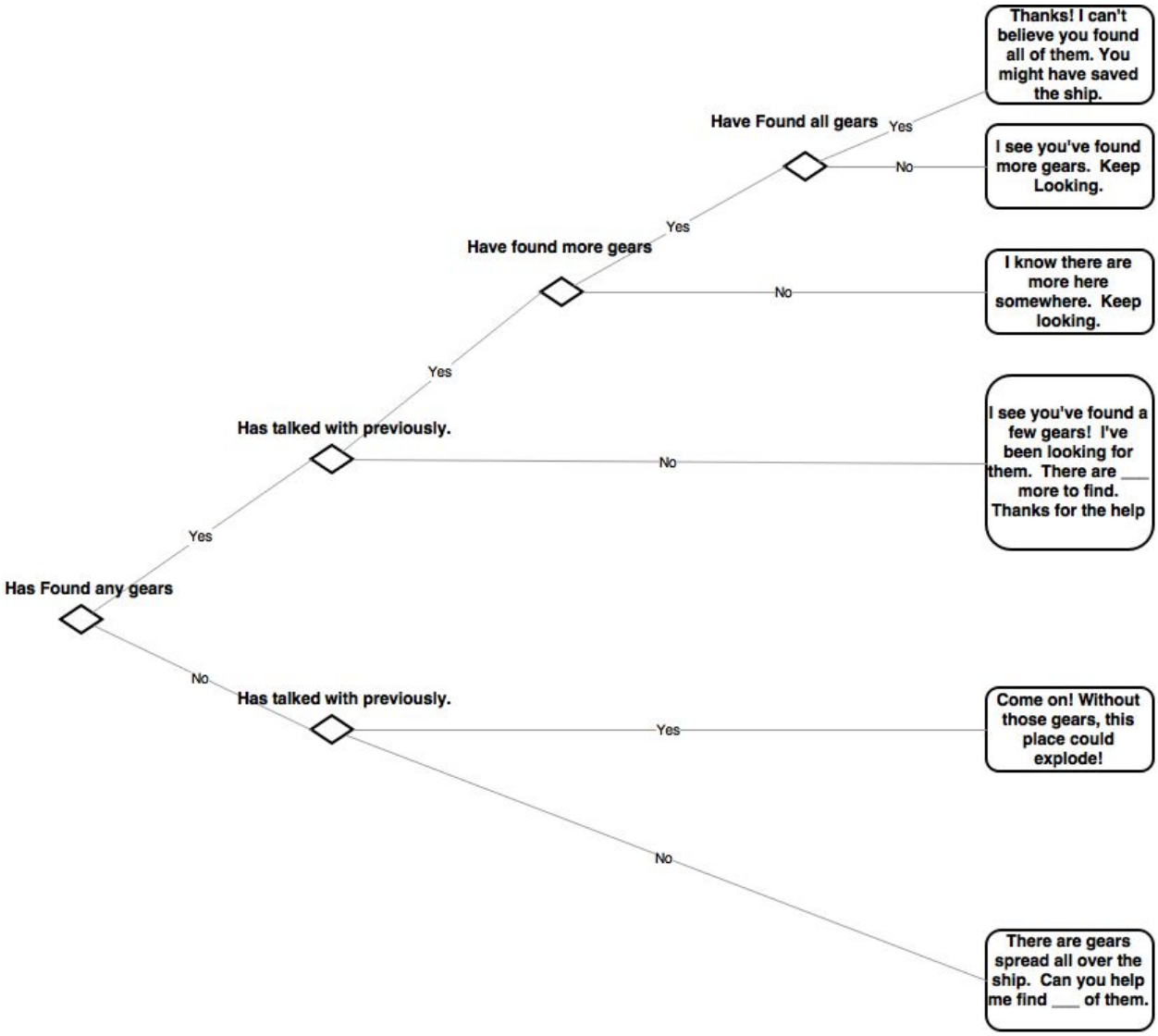
Website:

- https://gorbapork.github.io/

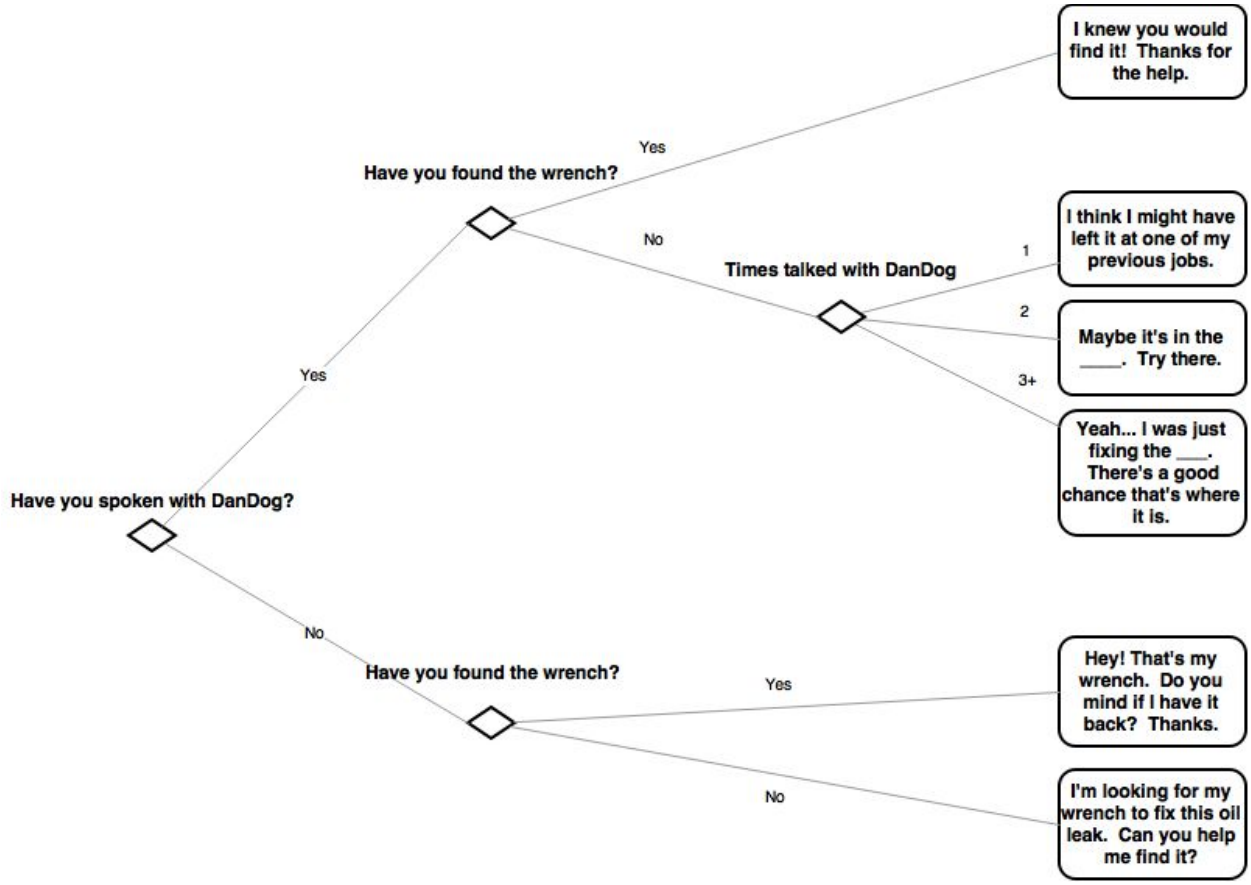
Demos:

- https://www.youtube.com/watch?v=SJ4ITSyG_gA
- https://www.youtube.com/watch?v=nLFAoSM9W-Q

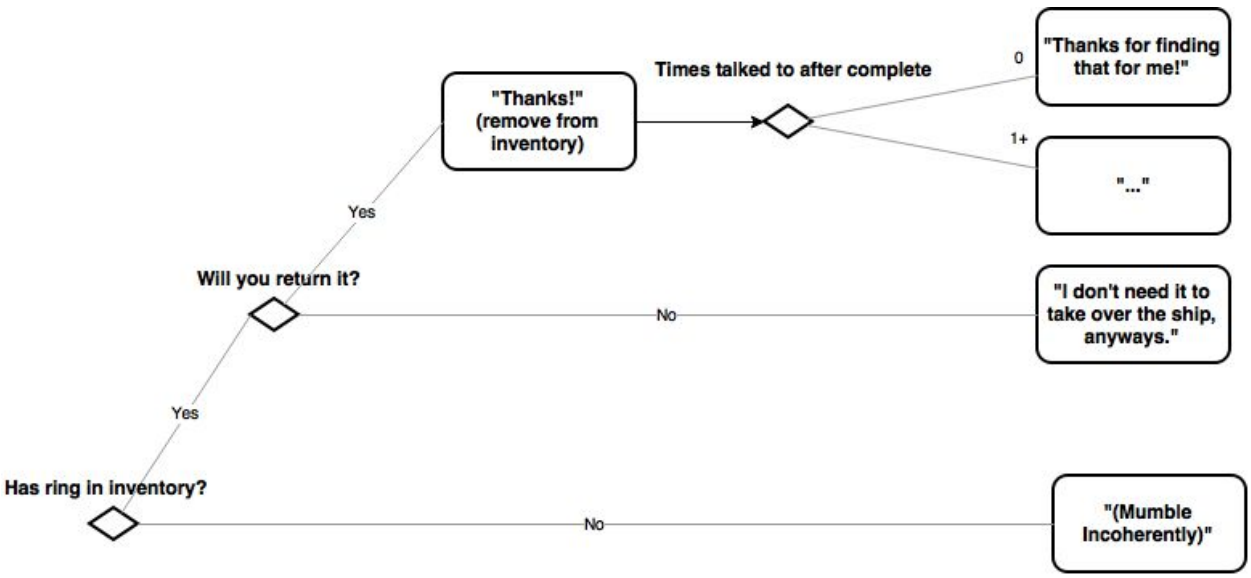
2.2.3: Dialogue Trees



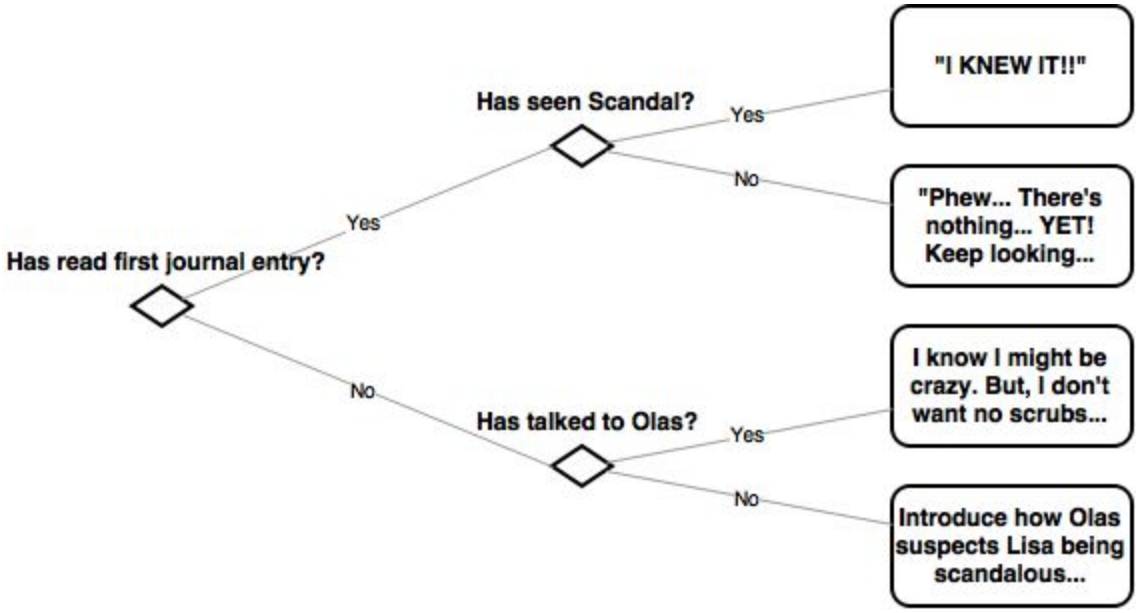
Decision Tree for the Gear-Search Mini-Quest



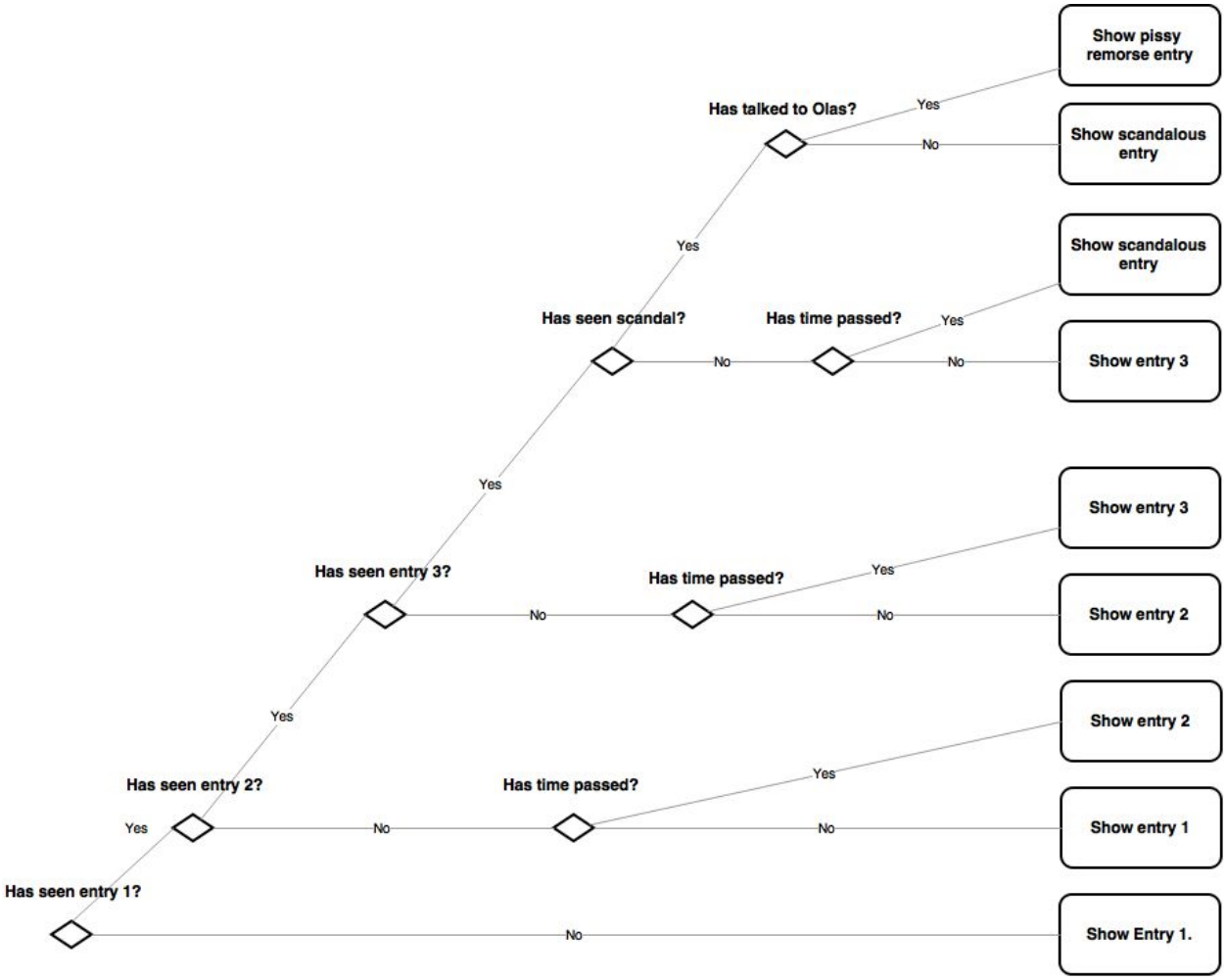
Decision Tree for the Wrench-Search Mini-Quest



Decision Tree for the Ring-Search Mini-Quest



Decision Tree for the Scandal Mini-Quest from Olas' Perspective



Decision Tree for the Scandal Mini-Quest from Journal Perspective