Cody Kagawa

Final Write-up

Class Breakdown:

Many of the classes are small intermediaries and are not especially worthwhile to examine on their own. Input set and its subclasses were created as a means of passing data around. Most of the actual MonoBehaviors in the code exist as a way of tying visual elements to code values and identifiers. Builders exist to encapsulate all of the information that would conceivably be removed from release builds of the game. These would be used to produce a loadable file that would instead be included in the game.

Assessor: The assessor was designed to function as an intelligent input reader. While it was originally conceived as an application of pattern recognition, implementing it as such proved to be too difficult for the amount of time that remained when it was created. Its primary functionality is to serve as an intermediary between the various input sources in the game and the judgment being used to analyze the player’s actions. As one of the 2 major focuses of the project, and consequently one of the largest unknowns, it was one of the most difficult classes to structure.

The class does not operate at the level I would have liked for this project. While it does function, there are few real elements of “intelligence” to this code. Ideally this class would have included pattern recognition and basic path prediction, neither of which is currently present. While this could be accomplished by replacing the current judgment with more sophisticated one given that much of the needed data and framework is there, creating such a judgment would not be a simple task.

CommandSet: CommandSet is a structural element of the code. It is conceptually based on the idea of a dynamic game script, a set of behavior patterns that would expand and improve itself as it received information. It was designed to serve two functions: to manage a collection of fitness evaluated commands, and to facilitate the creation of new compound commands based on provided base components and previously generated combinations. Commands of notably high fitness would be checked against preceding actions for commands with a high rate of preceding successful actions and be coupled together. This comes from an assumption that the supporting action likely creates a situation where subsequent actions are more likely to succeed, based on some metrics built into the implementation.

This class also does not function at the level that I would have liked for this project, though this comes as a result of other factors. The functions that it would have needed to serve its purpose existed at one point and still exist to a lesser extent in the current code, but they needed to be scaled back and reduced due to the insufficiencies of the evaluation code it relies on. Command compounding does not operate at all in its current state so it acts as a glorified ArrayList of commands currently. With improved evaluation code this could be used to potentially add an overwhelming degree of complexity to the functionality of actors, but attempting this with the current evaluator would make a mess of the existing AI functionality.

Evaluator: This is the core of the other major focus of the project and one of the major divergences from the original designs of the project. It was originally intended to be an implementation of supervised learning, but the need for a supply of training data made this impossible or at least useless for the purpose of the project. It is instead based on reinforcement learning, with the idea of fitness evaluating a command input against a player status output. It functions by taking a continual flow of input data from the stage and feeding it to an Evaluation that attempts to attribute this stage data to commands that caused it. The Evaluation then decides if this data is desirable and adjusts the value of the associated commands accordingly;

This class functions but only provides enough subtlety to direct command patterns, not improve them. The evaluation used by the evaluator does not possess enough nuance to really discern what actions actually support subsequent successful actions as opposed to merely preceding them. This is in part due also the great lack of variety in attacks, as this distinction would be easier to make if attacks were significantly less likely to appear mutually beneficial by coincidence. I have tried to permit command combination with this level of evaluation but with such a small number of possible enemy actions it becomes impossible to tell if the AI is even functioning in that state.

Unified AI: This is a basic structural class that manages the various actors in a scene and their respective command sets. It is meant to additionally manage the multiple command sets each actor would have for how to its behavior should vary based on what it is trying to accomplish.

This class does not perform switching based on changes in player target. While this would be an important function in a completed system, this function was a low priority for the project as it would not serve a significant purpose in the test environment. Had there been time to create heavily populated and highly complex stages, as one might expect in an actual game, this functionality would have been a much higher priority.

UnitManager: This class serves as the base for most general functionality. This class loads, destroys and monitors most actors, generates the player and user interface, and provides general stage information to the Assessor. It is also used to handle thread management for Evaluator processes and non-location based scene transitions.

The UnitManager is a major part of the project but a relatively minor part of the goals. Its only function with relation to project goals is providing stage information to the Assessor, which it does do. It does admittedly relay a relatively small amount of information to the Assessor but this is due to the Assessor’s weakness at processing rather than the UnitManager’s ability to provide.

StateMachine: This class is designed as a finite state machine for actors. It serves as a way of controlling what may be done to or by an actor, and at what times it may be done. While not technically a part of the project goals, it was desirable both as a way of creating clear distinctions for the AI to use in evaluations. It also allows for a less intensive command structure for the AI scripts.

This component was not explicitly necessary for the project, but was the most practical way to structure the game elements. Being able to easily restrict both the player and the enemy from taking various actions at various times greatly reduced the amount of work needed to maintain a manageable command pool and greatly simplified the requirements for the CommandSet class. These benefits do not show in the project very well as they would have help mitigate the load of assessments and evaluations, but neither reached the level of complexity that would have benefited from this class’ inclusion.

State: Representation of a State in the StateMachine. The class itself is not particular interest so much as the State instances that exist within the project. States were used to achieve the desired restrictive feeling of both player and enemy actions. They were intended also to send actor information the Evaluator regarding things like successful attacks and unit deaths.

States, particularly setting them up, took longer than they were expected to and probably should have. They also are not fully functional in a sense, as they were intended to be used for things like notifying the assessor of successful attacks. This functionality ended up being a part of the attacks themselves, though this was done in the interest of time rather than as a design shift. Putting the functionality in the states would have been cleaner and more effective, but the amount of time it would take to properly do so was not feasible at the time.

The remaining scripts are not insignificant to the project, but mostly exist for the purpose of gameplay or player convenience. Basic camera was made help reduce visual annoyances that occurred from merely child-ing the camera to the player and the UI elements were made to assist in debugging.

As a whole, the structure exists within the program for it to function as it was envisioned but several components are too weak to produce those results. In addition, the weak components make it such that parts that interact with them must also be scaled down to prevent the entire system from breaking.

Issue Summary:

The project was more troublesome than anticipated, thought that is not to say it was expected to be easy. Figuring out a system for making game objects visible to one another was surprisingly challenging when collision was not involved. It took a lot of time to understand how game objects are constructed, how best to interact with them, and what can and can’t be done with them.

Physics should not have been an issue but unfortunately the easy solution took too long to find, and by the time I learned how to stop rigid bodies from rotating and the various quirks of how colliders work I had ripped out most of that functionality. Collision currently works worse than it did previously, minus issues of rotation, and significantly works than it could be had I known about the mechanics of Unity’s 2D physics engine before. It still would not have been easy, as forcing the behavior I needed still violates some of the rules for normal physics, but it would have performed a lot better than it does currently.

Other issues included figuring out how to manage threads within singletons, as they originally failed to terminate when projects did and would run almost indefinitely. The ray casting that is currently used for collision also took a bit of work to function.

Identifiable Programming Influence:

Prof. Chambers was very helpful in providing me with an alternative when it turned out I could not structure my Evaluator as I had originally intended. I sent her a few emails regarding methods of building an AI without training data and she directed me towards reinforcement learning. I was not able to put her suggestion into practice as well as I would have liked but I hope to continue this project on my own in the future and will hopefully be able to put this knowledge into practice someday. Singleton construction was a concept I was introduced to in a mobile coding class I took at Hawaii Pacific University and it helped to avoid extensive loading in between each scene transition. The implementation still needs to be tweaked slightly but it has functions about as well as I would have hoped. Ray casting is taken from graphics classes I have taken, and thought it was included as an alternative in the time when I could not get collision to behave the way I needed it to it nonetheless allowed me to continue making progress with the project. Finite state machines are taken from Network Programming. While arguably a bit dated for game design, they are intuitive way of producing the sort of rigid behavior needed by this project.

Remaining Issues:

The project still has a handful of bugs and quirks that make certain things difficult to test or identify. Because of the way ray casting is used in calculating collision behavior, especially around corners, is inconsistent. For similar reasons attacks and collectables do not collide in as they might be expected to, given that a single ray will only create collision along a single line. This issue is particularly noticeable with the red boxes on startStage. Because there is only one enemy in the project the kill state cannot be entered normally. Because projectiles fizzle on hit and reset the enemy’s refire flag, Enemy 0 starts firing at faster intervals whenever a projectile successfully connects. There is also a minor quirk with the way movement is handled when if the player performs a moving jump too quickly after landing they will continue in their original direction, in spite of the direction the player is holding. There is also an issue with entering the boss stage in a state besides “Don’t Die” where it is unclear when the target naturally reverts. Because the boss room does not contain any functional stimulus for the Assessor, the target is essentially guaranteed to eventually revert to the “Don’t Die” target state, but because this is reflected poorly by the game interface it will appear that the player is being targeted when they should not be. Other minor issues include the player heads-up display not adjusting to resolution and on some setting appearing weirdly close to the player, and the lack of a termination trigger that causes the player to be stuck in the final stage with nothing to do if the boss is killed.

Concluding Remarks:

This project has not yet reached the level that I wanted to achieve, but it has been a valuable learning experience. I have gained a great deal from working with new things and exploring AI concepts I never had the chance to previously. While I regret being unable to realize all of the goals I set out to accomplish with this project, I am glad that I had the opportunity to attempt them. Now that it has become an issue of making things more robust rather than putting the project together I look forward to seeing what more I can bring to this project.