Notes

Created for Unity Learn Junior Programming Theory In Action. Completed on xxxxxxxxxxxxxxxxxxxxx

During this, my beginner’s journey to Unity and game development, I've heard the following message many times, 'just build something, FINISH it, and PUBLISH it; that is the best way to learn. For this last assignment, I decided to expand my effort and try to finish a more complete game. In this way, I hoped to reinforce skills development in the UnityLearn Pathways while building a body of work that could serve as reference material for my future questions and efforts. I am also hoping that documenting this process serves to exhibit some of the challenges encountered, problem-solving undertaken, and solutions developed.

Assignment Details

ABSTRACTION - Probably the easiest of the pillars to put into beginner’s practice. Used descriptive method names to act as surrogates for the details within. Attempted to break down complexity and sequester that complexity to methods that executed one primary function. Collections or sequences of method calls with these descriptive names forming a coherent expression/summary of the actions the code was executing. Abstraction used to assist ‘reading’ the code (perhaps like prose, if not well-written).

INHERITANCE – Three Tables includes two major classes that other scripts inherit from: ‘Food’ and ‘TableTracker’. Food was the first implemented and served as the base class for all the different food objects. This base class defined how food objects behaved in the scene, interacted with triggers, and outlined defining common variables. The TableTracker class was unanticipated and emerged during development to avoid code duplication that separate ‘Table1’, ‘Table2’, ‘Table3’ sections of code would have propagated. Inheritance from the Food class was probably the best use of the inheritance pillar; TableTracker would probably have been unnecessary if code was structured differently.

The original game concept was crafted with inheritance in mind. By having foods categorized into two dimensions of classes: protein (main courses) or non-protein (sides) and vegetarian or non-vegetarian. Vegetarian main dishes such as veggie-burger or tofu meat substitutes would drive complexity and polymorphism. With development underway, it became clear that the scope of menu and concept for inheritance was larger than it needed to be while still trying to figure out how to execute it. By the sounds of it, a common beginners mistake; too large in scope.

POLYMORPHISM – Method overrides were used to customize beef and salad object properties that varied from the base class. Method overrides implement different interaction properties at the grill and steamer stations for meats versus vegetables. One of the better examples is probably that ‘beef’ food type has three levels of ‘cooked’ for rare, medium, and well done. Method overrides are needed to implement multiple new colors beyond the scope of the base class.

ENCAPSULATION – Assigning variable and methods as private, protected, or public, was generally straightforward (though this example likely has ‘too many public variables for good practice’. While food temperature was set up with get; set and a private backing field, the use of getters and setters is still not very clear. I suppose getters and setters should be used to protect important variables like hit points (in my case food temperatures) and score. This more advanced encapsulations, get;set would make reading available (in most cases; for non-security variables) and writing more purposeful and controlled. The UnityLearn JuniorProgrammer section on this topic was brief. More research and use of this pillar will be needed in the future to better understand the get;set tools.

About Development

While developing this game, code was refactored several times. The first major refactor was to be purposeful/thoughtful about resetting variables after use. Methods for calculating a score are called to evaluate player performance after delivering food to guests at a table. The same methods are used to calculate the potential score of any given order when the order is generated. Because each of the three tables operate asynchronously and can call for a new order (and max score possible calculation), ensuring variables were cleared after use was a strategy to assist debug efforts. I remain unsure if clearing variables just before use or after use is a better practice. Clearing before use would initialize them ensuring they are reset; clearing after use prevents artifact values from persisting and potentially confuse debug activities.

The second major refactor was to establish the TableTracker class and remove specific ‘Table1, Table2, Table3’ references where possible. This refactor was a total breakdown of a previous OrderManager script, splitting code between the new TableTracker and existing GameManager scripts.

A third refactor was smaller in nature and removed ‘Table1, Table2, Table3’ duplication that had grown in TableTracker. Again, code was reduced to common elements to remain in Table tracker while exporting the table specific variable changes to GameManager. TableTracker had many GameManager.Instance.foo calls, so most of these were relocate to GameManager.

A fourth code refactor removed duplication from food scripts. Initially the methods to set color and item names were included in each food type. Really the variable values were the only thing that had to be defined individually, while the common method(s) were moved back to the base class.

The methods written to create orders and evaluate score after customers are served are quite verbose. Additionally, there are many If-statements and Booleans across the different scripts. It often felt as though ‘there must be a better way’; better coding practices and tools that could have been used, had I known what they were and how to implement them. Switch statements instead of If-Statements, multi-variable arrays to track food names and properties, maybe dictionary to store all the food properties associated with each food type and status?

As new features (scope creep) were implemented and code was refactored, maintaining ‘clean(ish)’ and ‘efficient’ code became a larger challenge. The tutorial layer, stats tracking (session and all time), food smashing/cleaning up with Scrubbi, UI flyout animations, perfect order tracking etc., were all added during development to add further detail, engagement, and interactive experience to game play loop. Reviewing all the code to verify/assess the access level for each variable will be needed. Many variables were moved around during refactoring and some public variables can probably be made private or protected.

There are still many features that could be added; adding polish can be never-ending if allowed. High score ranking, player levels, achievements, customer comments board, additional cook stations (pasta, fryer…) and foods (and drink) could be unlocked as the player progresses.

One significant area of development that remains on the backlog is scene lighting. The outdoor directional lighting is not well suited to the scene. At times and in certain positions/orientations, the directional lighting makes it difficult to assess food colors while cooking. This indoor scene should probably be enclosed with a series of overhead spot lights and maybe some points lights to fill the space. Also related to lighting, it would have been nice to implement shaders to create a glow off the grill.

Another best-practice from UnityLearn pathways, that was not implemented, is object pooling. Food items are instantiated at dispensers and destroyed at the tables, waste, and under floor when smashed. Smashed food is instantiated and later destroyed with the help of Scrubbi. A large quantity of food objects could have been pre-instantiated, stored in a List, and individually SetActive(false/true) by index as needed. I was unsure if something like 200 objects would be sufficient to deal with order levels as well as players messing around dispensing a mountain of food for fun. If the pool runs out, are conditional instantiation calls needed? This was left on the backlog. Unity performance tracker seemed to show good performance with the memory calls and garbage collection that could have been avoided.

About Debugging Code

As a normal part of development, and increasingly necessary while refactoring code, a lot of debug problem solving was practiced. Not yet knowing how to use debug tools, execute code one line at a time, or add break points, debug for this program was entirely manual. Thinking through code logic, tracing code execution across the scripts, placing Debug.Log calls, and when appropriate, limiting code execution by commenting out unnecessary actions/methods. On at least one occasion, complicated game objects were disabled in the inspector and replaced with place holder primitives and code snippets to verify/understand behavior.

Play testing was critical to identifying edge case bugs. Bugs in logic related to opening and closing tables. When closing a table, the table should be ‘inactive’ for new orders, however, should still accumulate ‘isOpen’ time for StatsTracker until guests leave. The table ‘reserved’ sign (aka closed) was initially being SetActive when the player closed the table immediately after serving and while guests were still at the table (there was no isWaitingToClose state). In other areas, logic was added to prevent kitchen door from closing if all tables were closed and player remained in dining area.

Known Issues and Workarounds

As completed, this game has as least one known issue with work around implemented to mitigate adverse effects on player experience.

In ServeFood, when moving food objects from the players platter to a table, sometimes the translation is applied to an object that was already moved. There were initially several adverse results including the gameObject with tag ‘Food’ exiting a trigger leading to removal from lists and losing references. Additionally, when the same gameObject was translated more than once, gameObjects at the end of the list are stranded with no remaining index values to move them from the platter. A workaround was implemented whereby stranded gameObjects (childCount of the platter > 0) are quickly destroyed. A downside to this workaround occurs when there are many food items to move; the move food loop may not complete before the method to destroy the stranded gameObjects is called. If the method to destroy stranded gameObjects is held until the full loop completes, stranded objects remain on the platter and are noticeably out of place and then disappear moments later. It was a better experience to have most food items move (visually) to the table, even if not all of them; a few missing items on the table would likely go unnoticed. The root cause of the problem was not uncovered. Because the method loops through a list of GameObjects, it remains unclear how an already moved gameObject could be selected again as the loop index is incremented. I have attached an image that shows when is happening that was captured visually and with Debug.Log messages.

The most significant issue in play testing was the experience was not very fun. For some playthroughs, the first order received was large (12+ items). Implementing limits on order size that ramped up over time allowed for delivery successes and quicker learning cycle in case of less-than-optimal kitchen/service performance.

Sizing UI and text for different screens and resolutions is a potentially significant issue that has not been addressed. As can be seen in the Game View window when not in full screen, lengthy text such as these Game Notes may run off screen. Some type of reactive font sizing or scroll bars are potential solutions to allow text to perform on a wide range of display environments.

A Note About Assets

3D models are basic primitives with color-materials manually applied, all generated in Unity.

Textures, materials, and 3D models could have been implemented. There are many free assets/packages available that could have been integrated. Finding assets that coordinate well (similar fidelity, scale, similar style, color palettes, etc.) has been left on the backlog. This first ‘game’ has been focused on the code development process and reinforcing the building block skills acquired through the UnityLearn pathways.

Audio is a key aspect to setting the scene and providing feedback to the player. While Three tables tries to integrate a number of sound effects, using OpenGameArt.org and cc0 assets where possible, lead to fairly limited options. In the future, accounts can be set up at freesound.org or other sites to expand options. If suitable free assets had been identified during development, I would have implemented crowd/diners noise at populated tables, positive or negative expressive sounds to match order deliveries with positive or negative scores, and potentially some kitchen background sounds.