Victor Haskins

Game Programming II

Lab 03 Report

Introduction

This lab expands on the use of tool creation, focusing on presenting and modifying information in close to real time on the Inspector window. We started with a proof of concept, showing how instance variables can be presented with automatic layouts. The class then broadened their view to include specific layout information as well as displaying arrays of instances and all their variables using serialized property information. Finally, the on your own section included two parts consisting of rounding out the lab tutorial's work and then creating two new editor tools, each with their own information instances and enums.

Methods

The lab work started off with using a basic class called Monster.cs with an enum list and several variables that are accessed via properties. As before, it is important to note the beginning information. Our Editor of the class information must have the “using UnityEditor” library, and be labeled an Editor subclass as opposed to the normal Monobehavior. We also label the whole class as a custom editor of the Monster class.

We start with a variable placeholder for the script and then in an Awake() function, we set that to our target Monster script. Now we can use it later. Other boolean variables are created now for later use. As opposed to just using the OnGUI() function that we were introduced to in previous labs, now we are using the OnInspectorGUI() function to house the editor on the Inspector window. The biggest takeaways were the foldout option for the automatic layout which can condense multiple pieces of information at once, the indentLevel property, that allowed the user to indent the displayed information so it was easier to read and understand, and adding a serialized property to the window, like that of an element of a list. The last idea becomes the central focus of the next part of the lesson.

We start with practicing foldout options, on three different sections of the code. As we progress, we offer toggle, (check boxes,) to change how information is stored in the functions. As the boolean value is read by the inspector, the available fields and surrounding labels change using if-else statements. On the last foldout option, we also see our first use of serialized properties. Using an automatic layout affords the array that is passed in it's own foldout and its basic information layouts for each element. It must have the serialized.Update() function before it so as to modify its information, display, and loadout. Considering that this is a list inside of our editor, with information contained there, we indent the array. Finally, an extra space is added for better readability, and then the function serializedObject.ApplyModifiedProperties(); is set at the bottom to send any changes made to the variable allocations.

From there, we move to the bigger part of the lab, which is to set up levels of data holding and inheritance. Like the last class, the ObjectTypes.cs class is provided for us, but there are more variables, less properties, and more enumerations used for several of the variables. We are then asked to create a controller class which is nothing more than a holder for an array of ObjectTypes class instances. Afterwards, we start on creating an editor class for our *controller* class, not our type class yet. We start similar to the monster editor where we create a script variable and initialize it in the Awake() function. Because several serialized objects are about to be used, we encapsulate the code with the Update() and ApplyModifiedProperties() functions. Finally, we create a serialized property for the array in the controller, just calling it controller, and display it using the function EditorGUILayout.PropertyField(controller). This will show the array in a similar fashion to the array in the previous experiment, but we take it one step further by first grabbing and displaying the property field for the controller array size. This will let us modify the list on the fly, something we previously would not have been able to accomplish. We then add a for loop to cycle through every element and show the item's information. At this time, we are basically done with the controller and finally get to move onto the object *type* information. The key difference between the controller and the type editors is that the type isn't just an editor. It still needs the Unity Editor library, but it is actually a Property Drawer and that type will replace the Editor base class, as well as having to change the attribute at the top to [CustomPropertyDrawer(typeof(ObjectTypes))]. The biggest steps for this new class are on the OnGUI(Rect position, SerializedProperty property, GUIContent label) function. Whatever GUI it is being called from, it will produce new information there to view. Information that will be presented and/or manipulated by the OnGUI() need to be sandwiched between the functions, EditorGUI.BeginProperty(position, label, property) and EditorGUI.EndProperty(). The parameters for the Begin property are from the OnGUI() function.

Every variable for the array element needed a new serialized property so all of those needed to be created. After that, we would take in data depending on a choice in the type enumeration variable for each element, so we encapsulated choices in a switch statement based off of that. Using a series of property fields, label fields, and spaces, each with their own distinct Rect() to house them, the appropriate information would be housed in each element. Many items in each item are left unused because it is a base class for a bunch of objects.

Lastly, since the information needs space to be seen, we add a public override function called GetPropertyHeight() to modify the display size for the item. This is for the Inspector node and if an array or foldout is displayed inside of the node, then the size is automatically expanded by that amount.

After the events of the lab are shown, we are tasked with filling out the final elements of the ObjectTypes tool and then set on creating two new tools. One for creating and storing enemy types, and the other will create and store weapon types. These will work similarly to the ObjectTypes class and container, as each tool will have both in addition to the base class. Certain extra constraints are added, such as if one property is selected, then other ones cannot be. Another task was to limit the options available in certain enum properties depending upon other choices. This last one I could not get to work and frankly ran out of time at the writing of this report.

We were also asked to write a script that would print the container information of the tools to the console. I wrote two special functions that I housed in the container classes, and called them with a button in the editor. They grab the appropriate spots and present them, but not quite to my liking.

Conclusion

This was difficult to complete because of time constraints with other classes and work as well as losing days worth of time due to being sick. After spending so much time hammering out the code for the lab in a condensed time frame, I decided to stick with using the inspector window instead of creating a new editor window. I was unable to figure out how to modify the initial opening size of a new editor window and was quickly losing patience with the process. At least the Inspector GUI was more clearly defined and afforded me a more logical and easier to use infinite canvas.

Once the ball got rolling and it finally clicked how to use the Rect's and offsets, the generation of the layouts practically wrote itself. I just tweaked a little bit of placement and squeezed some offsets to fit more information on fewer lines.

Overall, I found this project difficult to wrap my head around. It was slow moving, up until the end, but this tool has immense opportunities for the future and I can see myself abusing it mercilessly.