EasyEditordisplay

Kernm. GDV 2

Ruben Bergshoeff

Use of the tool

Simple display

1. Derive from the Editor_EasySimpleDisplay class instead of Editor
2. Override OnEnable and execute base instead of using regular OnEnable
3. Whenever change is required to the view, clear queue in Display and add objects

```
Display.AddGameObject (
    top.objectReferenceValue as GameObject,
    new Vector3 (0, 4, 0),
    Quaternion.identity);
```

Interactive display

1. Derive from the Editor_EasyInteractiveDisplay class instead of Editor
2. Override OnEnable and execute base instead of using regular OnEnable
3. Whenever change is required to the view, clear queue in Display and add objects
4. To add buttons or images, call the AddButton or AddImage method in the GUISystem

```
Display.GUISystem.AddButton (
    SetDayMode,
    (Texture2D)EditorGUIUtility.Load ("sun.png"),
    new Rect (-80, 5, 50, 50),
    GUISnapMode.TopCenter
);
Display.GUISystem.AddButton (
    SetNightMode,
    (Texture2D)EditorGUIUtility.Load ("moon.png"),
    new Rect (30, 10, 40, 40),
    GUISnapMode.TopCenter
);
```

Examples

In the project you can find a few examples of the usage of the tool in the folder `Editor/RubicalMe/Examples`

Explanation UML

EditorRenderer

This class is the center of the system, is responsible for keeping track of objects to render and draws the result in the specified rect. It contains methods for adding objects into the window; at this point in time GameObject and Texture2D types are supported.

InteractiveEditorRenderer

To keep functionality clear and the basic renderer as clean as possibly, interactive functionality has been added in this derived class of EditorRenderer. For now it is limited to a GUI overlay which is possible in two forms: a normal layover GUI which uses the display rectangle for placement, and a GUI which scales with the scale of the world rendered by the base class.

GUISystem

This class manages the GUI created. It could be used separated from the EasyEditor Display tool as it has no references to any of the other classes. It contains methods for adding objects to be drawn; for now just a button or image. To create a button, a method is required to delegate to when the mouse interacts with the button. For this an argument of type RM_ButtonEvents is required. The Draw (Rect) method draws the GUI at the specified position, the ProcessEvents (Event, Rect) checks if any interaction has been initiated and returns true if so.

Editor_EasySimpleDisplay || Editor_EasyInteractiveDisplay

These UnityEditor.Editor derived classes provide an integrated EditorRenderer or InteractiveEditorRenderer respectively into the preview area of the basic Editor. This makes it really easy to add a display to a custom editor as one just needs to derive from these classes instead of the basic Editor class.

EditorWindow_EasySimpleDisplay || EditorWindow_EasyInteractiveDisplay

Similar to the classes above, these derived classes provide an integrated display functionality on top of the UnityEditor.EditorWindow class.

Design Patterns

Factory Method in the GUISystem

To some extend one could argue that the GUISystem uses the Factory Method. While it is itself aware of all GUIItem classes as the creation of objects is done through methods within the GUISystem class, the GUIItem list contained within it is not aware of these and the Draw and ProcessEvents methods are equally unaware. This makes it easier to add new types of GUIItems without changing the basic methods of the GUISystem.

Observer Button

The Button class uses the observer pattern as a way of communication. Whenever the state of the button changes, it fires a delegate to let listeners know of its state.

Changes along the way

From PlatformEditor to EasyEditorDisplay

Initially I started out wanting to create an editor for platform instances (scriptableObject derived class) in my hybrid spaces project. For this I used a scene where a prefab could be selected, after which a series of cubes were spawned which were clickable and represented the relative grid positions of the platform. While this worked perfectly and was set up relatively quick, I wanted the designer to be able to change the platforms without having to load a different scene.

But having a preview display were multiple objects were visible, not all of them actual parts of the object itself, turned out not to be natively supported. That's were I decide to create my own display tool which in turn got to be my main focus for tool development. The PlatformEditor is still included in the project, to demo the use of the EasyEditorDisplay tool and for a look if you're interested.

Delegates instead of an EventHandler method

At first the GUISystem would process the current event and would fire a single delegate with event information. This would in turn execute the virtual method EventHandler, that the user of the tool would have to override. To know what button was interacted with, the event information would contain an ID of that button, which would be provided to the user on creation of said button. But soon this turned out to be a hassle as each button created would also require the editor of the user to have a variable for the button ID. Also, the EventHandler method would soon be a huge mess of if-else statements as the button ID variables are not constant values.

To solve this, I decided for the buttons to have their own delegate. On creation of the button, instead of having to save the ID and extending the EventHandler method to be able to handle the new button, the user now simply adds a method to be called when the button is fired. This makes for much cleaner and better to read code.

GUISystem class separated

When creating the InteractiveEditorRenderer I added a GUIItem list into this class and the overridden OnGUI method would execute relevant GUI elements. While this worked I wanted the GUI functionality to be more easily changed and read separately from the normal rendering.

To solve this I created the GUISystem class and added an instance of this to the InteractiveEditorRenderer. Now I can change the GUI entirely without having to change much to InteractiveEditorRenderer, which will make backwards compatibility a lot easier too. On top of that, the InteractiveEditorRenderer has become much more readable.

What I would have liked to add

Unity GUI items

I think for the GUISystem to be more complete, it would be great to add support for Unity's own GUI items. This way people can add the elements they are used to.

Collision handling / Ray casting

It would have been great to have a simple collision handling working to be able to determine weather someone has clicked an object in the rendered world. I wasn't able to get to that yet. This could allow for all kinds of great functionalities, like object / sub mesh selection or world-space GUI.

Instance management instead of re-creating the entire screen

Right now the whole "scene" that needs to be drawn has to be recreate every time an object change occurs. I wanted to find a way to acces individual objects but couldn't come up with a very user friendly way before the deadline.

Reading incompatible XML files

Input Data || Sergi's output data

The output of Sergi's tool, an XML file with data for his detective game consisting of various clues. A fragment:

```
    <clues>
      <Quest_Clues>
        <ID>333</ID>
        <clue>
There's shoe marks on the wall next to the fence.
       A cheap brand of sneakers.
       </clue>
        <found>0</found>
        <isKeyClue>false</isKeyClue>
      </Quest_Clues>
        ...
```

PlatformEditor

I decided to use the Platform Editor I started out with for the buddy-system as I thought this could add a dimension rather than just visualizing incompatible data (which I still do somewhat, don't worry)

Converting to relevant information

For importing the clue XML, I added a button in the PlacablePlatform_Editor class which would execute methods from PlacableObject_XMLimporter. This tries to convert the inner text of the XML nodes to either positions (if it's a string) or adds it to the variable that will decide the color (if it's possible to parse it to an integer).

After this, based on the Vector positions derived from the strings by using string.getHashCode (), a Mesh and material is generated and a prefab is saved via a prompt to the user. To decide which parts of the platforms should have collider in the grid, all vectors of the model are rounded, and at these rounded positions the platform will fill the grid of the game.

Then, to finish things up, another list of integers has been saved while processing the XML file. This list of integers determines on which platforms the should be deadly.

And another thing…

PlatformEditor

While in the end I decided to share my EasyEditorDisplay tool as my main assignment, I want to share one particular thing I'm proud of in the PlatformEditor: corruption handling and version control (the methods can be found in the script PlacableObject_EditMethods).

Version Control

Right from the start I decided to use versions in my platform instances (PlacableObject class). This way, the editor will know when the platform is made, and if the editor has changed a lot, a conversion can be built in. While this isn't actively used, if this was to be relevant it would be right there available.

Corruption handling

When loading in a platform instance, it will make sure the object has valid data stored inside. For instance, if the positions are beyond a certain scope that is not shown in the editor anymore, it will prompt the user if it's allowed to remove these. The same goes for events: if the event code is not recognized or if further information can not be verified, it will prompt the user instead of providing the user with an instance that will not be usable by the game.