Heliobiotic Project/Code Style Guide

Project

Folder Structure

Assets

- _Dev
    - Dev Name 1 (ex. Marcus)
        - (Exploratory stuff)
    - Dev Name 2
        - (Exploratory stuff)
    - … more dev folders
- _Zenith
    - Art (anything that's not a script or a prefab)
        - Materials
        - Models (just fbx and such, would prefer prefabs in prefabs/level folder)
        - UI
    - Scripts (all the scripts)
        - Gameplay
        - Managers
        - UI
        - Input
        - ScriptableObjects
    - Scenes (all the scenes)
        - Production (build scenes and milestones only)
        - Test (any other scenes for testing)
    - Prefabs (all the prefabs separated by general purpose)
        - Level
        - Gameplay
        - Managers
        - UI
- EVERYTHING ELSE
    - All the asset bundles and the FMOD project will have their own folders in the root asset folder. We keep them here to avoid issues when updating and such

Folder Organization

There should be no bare hanging files in the root Asset folder or any of the folders one level below (Art, Scripts, etc). Everyone should have their own space in the _Dev folder to mess

around and play with anything. When something becomes more final, move it into the main Zenith folder.

General File/Folder Naming

There should be no spaces in any file or folder names, including all assets and scripts and folders. Use pascal case for all folders and scripts, when numbers are involved or pascal case obscures naming you may use dashes.

Prefabs

Make prefabs all the time! Whenever you think a prefab can be beneficial, make one. You can even make prefabs within prefabs and prefab variants. Honestly pretty much everything should be a prefab, it allows us to work concurrently.

Git / GitHub

Tasks

Create tasks with proper naming and tags. Names should be relatively short and only as descriptive as necessary. Descriptions should ideally list goals and subgoals required to complete the task, preferably with simple concise bullet points. Make sure to apply the appropriate tags.

Branching

Make branches based on tasks. If a task doesn't exist for your branch or the work you want to do then make it. Tie your branch to your task, in the task menu there is an area where you can select a branch to associate with that task.

Name your branch based off of the task name or what work you are trying to do. If the work ends up being different than expected, feel free to rewrite the task a little or make a new task / branch to continue in a new direction. DO NOT use names like "{your name}-branch" as it does not convey anything about the work you are doing.

Merging

Always create pull requests for merges, especially if you are merging into more central branches with other people's work. Use common sense as to whether you should simply approve the merge yourself, send a quick message over Discord about the merge, or ask someone to review the pull request thoroughly before merging. Creating pull requests allows for others to easily review your changes, another clear way to see what files will be changed in the merge, and makes our branch history much more obvious to look over.

If there are changes you would like to incorporate into your branch or you have merge conflicts I highly recommend using the “git rebase” command in the command line instead of merging from another branch into yours. For example to get updates from main simply pull then run “git rebase main” while on your branch. However, if you are just trying to get other people’s work then merging also works. Please make sure you commit everything and have fetched / pulled the other remote branch before rebasing or merging. If this confuses you just ask someone else to merge or rebase or solve merge conflicts or make pull requests for you.

Selective Change Discarding

Please look at what changes you are about to commit before committing. Oftentimes you will not realize you changed something that you did not mean to change. If a file is marked as changed that you don’t think you meant to change, revert that change before committing.

Look out for some pesky files that like to change by themselves and be annoying:

- Font assets (anything under Zenith\Assets\TextMesh Pro\Resources\Fonts & Materials)
- Probuilder settings

Code Style

Accessibility

Always include summaries for public functions. Always include tooltips for variables exposed in the editor. Preferably include headers as well and ranges if applicable. PLEASE MAKE YOUR ATTRIBUTES AND METHODS PRIVATE BY DEFAULT. If it does not need to be accessed by anything outside your class it should be private.

Indentation and Spacing

In general: Use spaces (4 space tab) - default in VS Code and Visual Studio
Expressions: Include spaces between operators and values, ex:

```
var x = myDict[2(y + 1) / (1 - z)];
```

Naming Style

Classes are `PascalCase`. Generally attributes and variables are `camelCase`. Methods and functions are `PascalCase`. Publicly accessible attributes are `PascalCase`. Do not use names in the forms of “myVariable” or “m_Variable”. Only use one letter variables for quick mathematical expressions.

Accessibility

Always include summaries for public functions. Always include tooltips for variables exposed in the editor. Preferably include headers as well and ranges if applicable.

Programming Architecture

Referencing Other Scripts

If you need a script or component to communicate with another script or component here is a guideline you can follow. This is not strict and based more on my intuition than anything, always do what is best for your use case.

If your dependency is on the same game object or a child / parent object (and presumably in a consistent prefab)

- Same game object
    - Use GetComponent together with the RequireComponent attribute to guarantee the dependency component exists on that object
- Child or parent game object
    - Use serialize field and drag reference in editor
        - You likely have a prefab so you can just set references in the prefab editor which will be preserved when the prefab is instantiated in a scene
    - OR use GetComponentInChildren / GetComponentInParent
        - This will also work but less recommended since there is a chance the component doesn't exist

If your dependency is on a completely different game object

- If this script is not going to be instantiated during gameplay (all the instances of this script that will ever exist are already in the scene) and there aren't that many of them
    - Use serialize field and drag the reference in the editor
- If new items will be instantiated that absolutely require a central dependency
    - Create a service (see below)

Services

If you think a centralized manager any script can access is the best way to implement your feature (or if you want to make a singleton), then you can make a service object.

1) First make a service script (inherit from the Service class instead of Monobehaviour)
    a) I recommend naming it in the form "{Purpose}Manager"
    b) The service class is simply a wrapper over monobehaviour so you can still access all your Unity event functions (Awake / Start / Update / etc.)
    c) You now also have access to new event functions which you probably don't need:

    i) OnRegister() which is called when the service is registered with the service locator
    ii) OnUnregister() which is called when the service is unregistered with the service locator
2) Then for each scene your service needs to be accessible from (really it's probably only the main game scene) add the script to an empty game object and turn that into a service prefab (please put it with the rest of the service prefabs)
3) Then make sure there is only one instance of that game object in the scene and add that service object to the services list in the scene's service bootstrapper object
4) Make sure to set whether the service should be persistent or not with the checkbox on your service object script in the editor (if checked the service will persist across scenes, you probably don't need this unless you really need to preserve state)

Now you can access your service from any script in this form:

```
fooManager = ServiceLocator.Instance.Get<FooManager>();
```

Where "FooManager" is the class of the service you are looking for (the service class you made in the previous steps). Please create a private variable for your service reference and then call this service locator function in the Awake() call.

Null Checks

Refrain from doing null checks when a null value is not expected behavior. We want to see errors, not suppress them. In other words pretty much don't use null checks unless it is OKAY if the value is null. So really you should almost never use null checks.

UNACCEPTABLE null check example uses:

- When accessing services/managers
- When accessing input actions
- When accessing components that should always be there

Do not use Unity null coalescing (anything in the form of "?." or "{type}?" or "?==") unless you are using it to invoke event actions (where this is standard practice):

```
Action?.Invoke("message");
```

Event Subscribing

When you do something like this:

```
service.Event += OnEventTriggered;
```

or like this:

```
button.onClick.AddListener(OnButtonClicked);
```

you are subscribing to events.

Do these subscribing operations in the OnEnable() function. OnEnable() is called after Awake() and before Start() in the monobehavior initialization chain.

Then you MUST put the corresponding unsubscribing operations in the OnDisable() function. OnDisable() is called before objects are disabled and then destroyed. If you do not do this then objects that are disabled then re-enabled will resubscribe their events and your subscribing functions will be triggered multiple times.

For general events / action unsubscribing:

```
service.Event -= OnEventTriggered;
```

or for listeners:

```
button.onClick.RemoveListener(OnButtonClicked);
```

or even better (I usually use this):

```
button.onClick.RemoveAllListeners();
```

Unity Event Function Usage

Check out the Unity event function execution order here

Fun fact: “Awake is only called before OnEnable within the context of a single script. If you have multiple scripts with these methods, both Awake and OnEnable can get called for one script, before both get called for another one.”

Places I recommend putting things in the event function execution order:

- Get your services in Awake()
- Put event subscriptions / unsubscriptions in OnEnable() / OnDisable()
- Put general initializations in Start()
    - Put input action initializations in Start()
    - Put GetComponent calls in Start()
- Put any physics APPLYING methods in FixedUpdate()
    - If a raycast then affects physics applying methods also put the raycast in fixed update

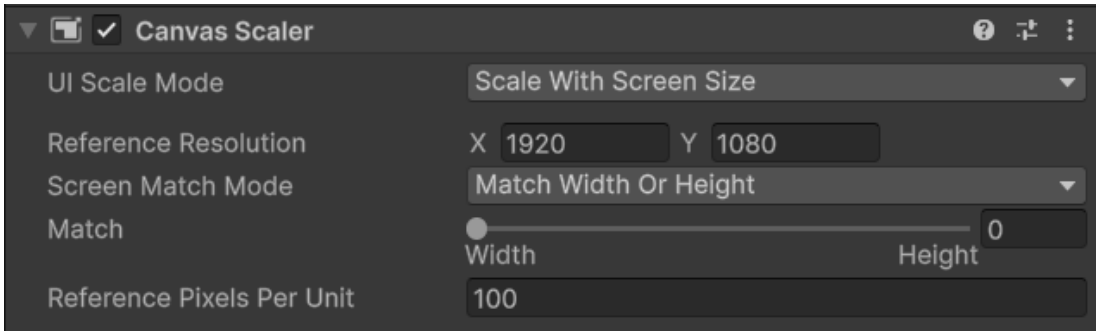
UI and Menus

If you need a Canvas, then use the new UI system. There are two types of screens you can make:

- Menu (UIMenu): Canvas that can be open or closed, only ONE menu can be open at any one time, can open the cursor
    - Recommended for anything interactive (need a cursor)
- Overlay (UIOverlay): Canvas that can be open or closed or hidden, any number of overlays are allowed to be open at once, cannot open the cursor
    - Recommended for anything non-interactive

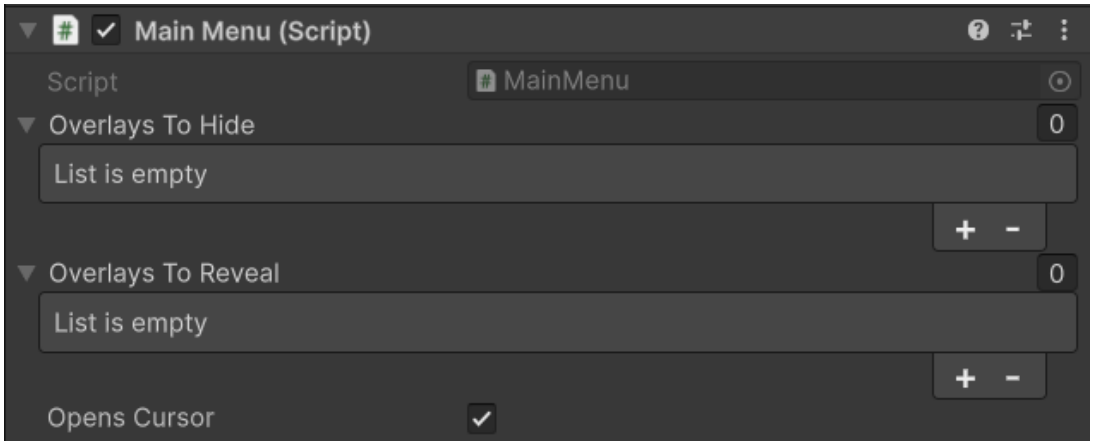
To make either simply make your Canvas like you usually would and put everything you need on it, then on the game object that also has the Canvas component add a UIMenu or UIOverlay script that you create (inherit from either of these classes instead of Monobehaviour).

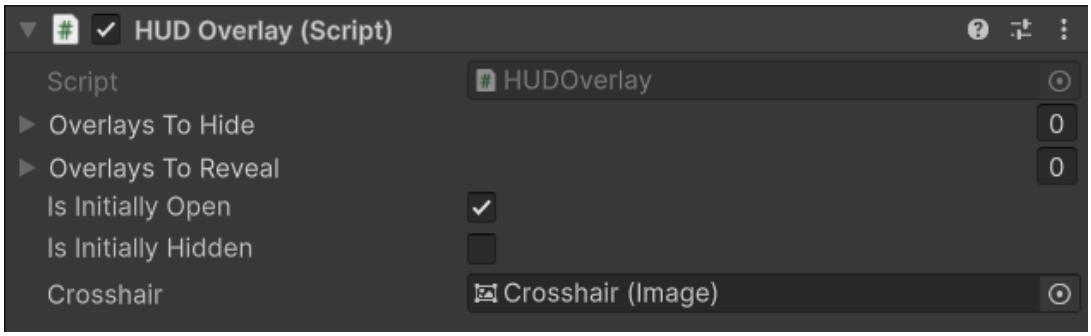
1) First make a menu or overlay script (inherit from the UIMenu class or UIOverlay class instead of Monobehaviour)
    a) I recommend naming it in the form “{Name}Menu” or “{Name}Overlay”
    b) Both classes are simply wrappers over Monobehaviour so you can still access all your Unity event functions (Awake / Start / Update / etc.)
    c) Menus and overlays can be in 3 different states
        i) Closed: The menu or overlay is inactive and invisible, does not receive update calls
        ii) Hidden: The menu or overlay is active but invisible, still receives update calls
        iii) Opened: The menu or overlay is active and visible, receives update calls
    d) Menus and overlays can have four state changes called on them
        i) Open: Opens the menu
        ii) Close: Closes the menu
        iii) Hide: Hides the menu
        iv) Reveal: Sets the menu to open from a hidden state
    e) You now also have access to new event functions which you may want to use:
        i) OnRegister() which is called when the menu or overlay is registered with the UI Manager
        ii) OnOpen() which is called when the menu or overlay is opened (great place to put your menu opening juice or setting default state)
        iii) OnClose() which is called when the menu or overlay is closed (great place to clean up or save things)
        iv) OnHide() which is called when the menu or overlay is set to hidden
        v) OnReveal() which is called when the menu or overlay is revealed
    f) You should make exposed fields for all of your UI Elements you need to access
2) Make a game object with a Canvas component, Canvas Group component, and your menu or overlay script. Add this canvas as a child to the scene canvas (should already exist), then make your canvas a prefab (this ordering is important).
3) You should now work in this prefab to make your menu. Within the prefab put all your UI elements as children of the Canvas as you normally would then plug in all your UI elements to the exposed fields in the menu or overlay script you made.
4) Make sure the following canvas scaler settings are set:



5) Choose your menu or overlay settings

a) “Opens Cursor” checkbox (for menus) sets whether the cursor should be open while your menu is open
b) “Is Initially Open / Hidden” checkbox (for overlays) sets whether the overlay starts closed, open, or hidden at the beginning of the scene
c) “Overlays to Hide” is a list you can drag overlays into so that while your menu or overlay is open, those overlays are set to hidden
d) “Overlays to Reveal” is a list you can drag overlays into so that while your menu or overlay is open, those overlays are revealed if hidden





6) Add your canvas in the scene to either the menu or overlay lists on the UI Service game object.
7) Set your canvas to inactive in the scene (uncheck the big check in the top left of the inspector so the canvas is greyed out in the hierarchy)

Now you can access your menu or overlay from any script in this form:

```
uiManager.GetMenu<FooMenu>();
```

or for listeners:

```
uiManager.GetOverlay<FooOverlay>();
```

Where “FooMenu” or “FooOverlay” is the class of the menu or overlay you are looking for (the menu or overlay class you made in the previous steps). You can get the UI manager using the service locator (see services section above).

NOTES

Logic separation: Please only reference UI elements (buttons, sliders, text, etc.) in a menu or overlay script. The point of making these scripts is so that you can get references to them and then call public functions you make which then interface with the UI elements instead of interfacing with the UI elements directly. I also recommend not storing too much general gameplay logic in the menu or overlay scripts. It is preferred that the only logic in these scripts is UI related.

Why hidden and reveal: Sometimes you might want to hide overlays but keep their update loops running (looking at the timer). This way you can decide to use your overlays with hidden/revealed states where your overlay keeps running while invisible rather than open/closed states where your overlay keeps is totally inactive when invisible. While the hidden/revealed style is good for things where you can start it up once and preserve state throughout the scene's lifetime, the open/closed style is good when you want things like Awake / Start / OnEnable to be called every time the menu or overlay is opened and you want Unity to clean things up for you every time the menu is closed.