Open Projects | Conventions

Useful links
Chop Chop Github repo | Open Projects forums | Chop Chop Roadmap

Code

We used the .NET standards as a starting base, and the following are our own changes.

Wording

- Use **descriptive** and **accurate** names, even if it makes them longer. Favor readability over brevity.
- Do not use **abbreviations**.
- Use **acronyms** when they are an accepted standard. Ex: UI, IO
- Method names should be **verbs** or **verb phrases**.
- Property names should be **nouns**, **noun phrases** or **adjectives**.
- Boolean properties should be **affirmative** phrases. You can prefix a Boolean property with “Is”, “Has”, “Can”. Ex: IsActive, CanJump.
- If **multiple properties** relate to the same item, use the **item name** as a prefix and add the property type, or role. Ex:

```
Color _gameTitleColor;
String _gameTitleString;
TextMeshProUGUI _gameTitleText;
```

- Avoid using **numbers** for names, if they are not part of an inherent list. Ex: *animator1*, *animator2*. Instead **explain** the difference between both properties -eg *playerAnimator*, *enemyAnimator*

Capitalization

Definitions
camelCase: first letter is lowercase. First letters of the following words are uppercase.

PascalCase: The first letter of every word is uppercase. If a word is an acronym with two letters, both letters are capitalized. If a word is an acronym with more than two letters, only the first letter is capitalized.

- Classes, methods, enums, namespaces, public fields/properties are written using **PascalCase**. Ex: `ClassName`, `GetValue`.
- Local variables, methods parameters use **camelCase**. Ex: `previousValue`, `mainUI`.
- Private fields/properties are camelCase, but start with an underscore. Ex: `_inputReader`.
- Constants are all-caps, and they use underscores to separate words. Ex: `GRAVITY_AMOUNT`.

Programming

- Keep fields and methods **private**, unless you *need* them to be public.
- If you want to expose fields in the Inspector without actually making the variable accessible to other classes, use the attribute `[SerializeField]` and `private`, instead of making them public.
  Note: Doing so, you might get the warning "Field is never assigned to, will always have its default value". Assign the default value to the field with `= default`.
- Try to avoid the usage of Singletons. Explore usage of ScriptableObjects (1, 2) for a similar, centralised class that can be accessible from multiple objects.
- Do not use `var` when declaring a variable. Always write its type explicitly.
- Avoid using static variables. If you absolutely need them, make sure they are compatible with *Fast Enter Play Mode* as detailed here.
- Do not use hardcoded "magic numbers" in your code. Ex: The player is moved by `xInput * 0.035f`. Why that number? Instead, store the number in a field with a clear name - and maybe a comment on why you chose that specific number.
- For "using" directives (Ex: `using System;`), remove all the unused ones before committing code.
- If you want to enclose your code in a namespace, use "UOP1" to denote code and assemblies specific to this project.
  For example: `namespace UOP1.Dialogues.Helpers;`

Formatting

- Use **1 Tab** per column to indent code, not spaces.
- Curly brackets: if they are empty, they should be on the same line. If not, they should be on their own line and aligned in the same column. Ex:

```
public class EmptyBraces(){ };
public class NonEmptyBraces
{
      //...
}
```

- Logical units which are contained within each other need to be indented to indicate the hierarchical relationship. Ex:

```
public void FunctionName()
{
    if(somethingHappened)
    {
        //...
    }
}
```

Comments

- **Important:** Don't be redundant. If you think anyone could understand what the code does by just looking at it, don't add a comment. Instead, name your variables, classes and methods so that they explain themselves!
- Use inline comments to provide additional context over individual lines of code.
- Write a summary above every class that describes the class' purpose. Optionally, include details about how the class works, especially if it's not particularly intuitive or readable. Ex:

```
/// <summary>
/// This class manages save data
/// </summary>
```

Tip: IDEs usually auto-generate a summary when typing the "/" symbol 3 times.

For more information on summaries, check the official Microsoft specification.

- Write a comment before a method, to explain what it does, in case the name is not self-explanatory or you want to add important details. You can also use an inline summary. Ex:

```
/// <summary> This function does this... </summary>
public float CalculateBoundingBox(){ }
```

- Use a comment beginning with `//TODO:` to indicate something that needs to be picked up later, so you don't forget about it. **Note:** This is not an invite to push broken functionality.
- Do not use `#region` dividers, or "line separator" comments like `//--------`.

Scene/Hierarchy

Organisation

- Use empty GameObjects on the root as separators to break up visually different logical sections. Ex: --- Camera ---, --- Environment ---, --- Lighting --- ...
  Apply the tag EditorOnly to these objects so they get stripped from the build.
- Use empty GameObjects as containers when useful, but don't if they will only contain 1-2 objects.
- UI:
  - Use the same Canvas when possible, only create multiple Canvases when Canvas properties change.
  - Create a panel per screen (main menu, settings, pause...).
  - Use panels as containers to group parts that compose an element of the UI. Ex: a settings label and its options.
    Panels can also serve as a helper for elements that need to be anchored together. Ex: some energy/items UI in the bottom right part of the screen.

Naming

- Don't use spaces in names of GameObjects.
- Use **PascalCase**. Ex: MainCharacter, DoorTrigger
- Use underscores to join together two concepts when just using PascalCase would generate confusion. Ex: MainHall_ExitTrigger, BossMinion_AttackWaypoints.

- **Prefabs:** Rename instances if it makes sense. Ex: A Prefab Variant file is called Protagonist_Scene1Variant, but once you use it you could rename it just Protagonist.

Project files

Naming

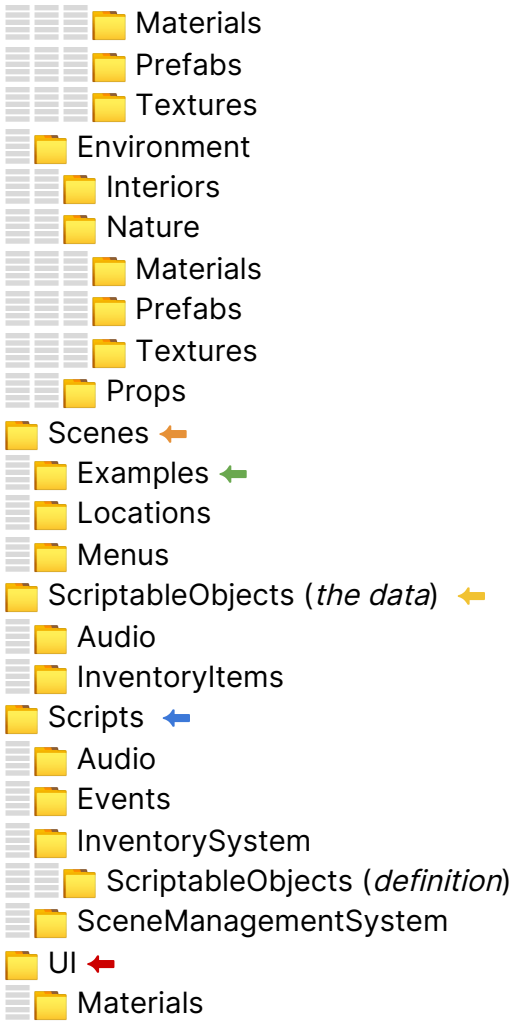
- Same rules as for the Scene/Hierarchy.
- Name objects so they **naturally group together** when they are in the same folder and are related.
    - Generally, you start the name with the thing that the object belongs to. Ex: PlayerAnimationController, PlayerIdle, PlayerRun, etc.
    - However, when it makes sense, you can name objects so that similar objects stay together even if they relate to different “owners” or if the adjective would group them differently. Ex: in a folder full of prop assets, you can use TableRound and TableRectangular so they stay close; instead of RectangularTable and RoundTable.
- Avoid filetypes in names. Ex: use ShinyMetal instead of ShinyMetalMaterial.

Folders

At the root level, put your assets in folders which identify areas/systems/locations of the game. In there, you can create sub-folders to separate different types of assets.
Scenes always go on a root folder
Scripts go in a root folder called Scripts, and then separated by system. You can create sub-folders in there to better categorise them, and the scripts that define ScriptableObjects should go in here.
The ScriptableObjects themselves should go in a root folder called accordingly.
When you make a new system, you can put an **example scene** within /Scenes/Examples. Try to gather the related test assets here (Prefabs, Timelines, ScriptableObjects) and don’t just place them in other parts of the project.

Example:

Art
Characters
PigChef


Materials
Prefabs
Textures
Environment
Interiors
Nature
Materials
Prefabs
Textures
Props
Scenes
Examples
Locations
Menus
ScriptableObjects (the data)
Audio
InventoryItems
Scripts
Audio
Events
InventorySystem
ScriptableObjects (definition)
SceneManagementSystem
UI
Materials