Glorious Nippon Project

Technical Design Document

2017/10/11

Contents

[Project Location 2](#_Toc495503189)

[Development tools 2](#_Toc495503190)

[Naming conventions 3](#_Toc495503191)

[Folder structure 4](#_Toc495503192)

[Code Architecture 5](#_Toc495503193)

[Introduction 5](#_Toc495503194)

[Toolbox 5](#_Toc495503195)

[Managers 5](#_Toc495503196)

[Controllers 6](#_Toc495503197)

# Project Location

Repository:

https://github.com/LeeviPaa/GloriousNipponProject

# Development tools

Source control:

GitHub

SourceTree

Engine:

Unity

Programming tools:

Visual Studio

2D Graphics tools:

Photoshop

3D Graphics tools:

Blender

3dsMax

Unity Plugins:

VRTK

SteamVR

PostProcessing Stack

# Naming conventions

Use camel case (first letter of each word is capitalized, e.g. ThisIsCamelCase) when naming any assets for the project.

Underscore is the preferred way to separate different parts of the name from each other.

When suitable, make up descriptive “subcategories” for asset names. Meaning, instead of having eight different doors with descriptive but long names, they should be shortened from “DoorMahoganyWorn\_Mod” to “DoorWooden01\_Mod” and from “DoorSteelRusty\_Mod” to “DoorMetal01\_Mod” for example. Categories and subcategories should always start from the broadest and least descriptive (e.g. object type, like “Door”) and end with the strictest and most descriptive (e.g. detail type, like “Worn”).

When faced with a situation where multiple assets would either the same name or the names would have to be considerably longer (e.g. DoorWood\_Mod -> DoorMahoganyWorn\_Mod) and creating new categories is not viable, adding numbers to the names is preferred. For example: DoorWooden\_Mod -> DoorWooden01\_Mod.

Pre-assigned three letter post-fixes should be used in most asset names. Post-fixes are separated from the rest of the asset name with an underscore. With some asset types, such as scripts and prefabs, these post-fixes are unnecessary. If a post-fix is not included in the following list and you’re not sure if the asset type should have one, consult one of the team leads. Post-fixed currently in use:

Texture Tex

Material Mat

Model Mod

Dates are written in year/month/day form. For example, the 21st of September 2017 would be 2017/09/21. Use slash (/), dot (.) or underscore (\_) to separate the numbers if deemed necessary. When an asset name includes a date, non-separated format is preferred (20170921 vs 2017\_09\_21) to prevent unnecessary long and complicated names. In general however, dates should only be included in build names, not in basic assets’ names.

Example asset names:

Example 01: A prefab of a lootable golden chalice

**LootableChalice01**

“Lootable” is the broadest category, “Chalice” gives a more detailed description, whereas “01” differentiates this prefab from other lootable chalices. Prefabs do not use post-fixes, so none is necessary.

Example 02: A texture for an exterior stone wall.

**WallExteriorStone01\_Tex**

“Wall” is the broadest category, “Stone” gives once again a more detailed description. The optional “Exterior” subcategory should be added if deemed necessary for extra visibility (if the wall texture count increases to more than just a few for example). The number “01” differentiates this texture from other stone wall textures. Since textures should use “Tex” post-fix, the “\_Tex” is added to the end of the asset’s name.

# Folder structure

Directly inside the main root folder are only the Unity project folder, the documents folder and git related mandatory files. Additional folders may be added to the root folder if necessary (these should only include files that are considered necessary to be included in the repository, but not in the actual Unity project files).

The Unity project folder (and the Assets folder within) contains most of the actual files used in the project, such as models, textures, scripts and scenes.

The documents folder contains all the available documents related to the project (Technical Design Document, Game Design Document, etc.). Subfolders can be added inside the documents folder when deemed necessary.

Unless you know what you are doing, do not add or modify any files outside of the “Assets” and “Documents” folders.

**GloriousNipponProject** Repository root folder

**Castle Raid** Unity project root folder

**.vs** Auto-generated Visual Studio files. Ignore when pushing to repository

**Assets** Unity assets folder. Contains most of the actual assets used in the project

**\_Scenes**, **Materials**, **Scripts**, etc. Separate folders for different asset types

**Library** Local Unity files and settings. Ignore when pushing to repository

**Temp** Temporary Unity files. Ignore when pushing to repository

**Documents** Contains all available documentation related to the project

# Code Architecture

## Introduction

Biggest classes are separated into two categories: Managers and Controllers. In addition to these categories, other smaller scripts are also included in the project, though in most cases they can be classified as “Controllers”. The final class type is the Toolbox.

## Toolbox

Toolbox is a singleton which handles the management of the Global Managers. One and only one Toolbox always exists in the game. Reference to any of the Global Managers can be found through the Toolbox.

## Managers

Managers are classes that handle bigger sections of the games functions, such as a “UIManager” or an “EffectManager” would. Managers can be separated into two different groups: Global and Local Managers.

Global Managers control sections of the game that move between scenes, such as UI, game state and events between scripts. There will be a very limited number of Global Managers overall and creating a new one should always be discussed with the rest of the programmer team. Reference to all the Global Managers can be found through the Toolbox.

Current Global Managers:

**Game Manager**

Handles score keeping, gameplay timer, most gameplay events, etc.

**Event Manager**

The information highway between classes, that allows scripts to communicate without having an actual reference to each other. Functions on a subscription / broadcast event basis.

Local Managers on the other hand are more of a mix between Managers and Controllers. While they control individual actors instead of larger cross-scene features, separating them from the Controllers is the fact that Local Managers can control multiple actors simultaneously. Alternatively, Local Managers may be managing a single gameplay system during gameplay in a level, such as “global alert state” for all the enemies in the scene. Example Local Managers could include classes like “AlertStateManager” or “CastleStateManager”. Currently the difference between Local Managers and Controllers is too blurry, and the Local Manager classification might be merged with Controllers later on, if deemed obsolete.

## Controllers

Controllers handle the controlling of individual actors within the game. Controllers would include classes such as “EnemyGuard”, “Door” or “TrappedClimbable”. Multiple Controllers can exist simultaneously and generally they should not be controlling more than one actor (an entity in the game world, such as an interactable object or a patrolling enemy) per controller.