2.3.1.10.2 Guida allo stile

Per garantire la coerenza tra i progetti, consigliamo di seguire queste guidelines:

- utilizzare **snake_case** per cartelle e nomi dei file (ad eccezione degli script C#). In questo modo si evitano problemi di case sensitivity che si possono verificare dopo l'esportazione di un progetto su Windows. Gli script C# rappresentano un'eccezione a questa regola poiché, per convenzione, devono avere lo stesso nome della classe al loro interno, scritto in PascalCase

- utilizzare **PascalCase** per i nomi dei nodi, in quanto corrisponde al node casing built-in

- in generale, mantenere le risorse di terze parti in una cartella `addons/` di primo livello, anche se non sono plugin dell'editor. Ciò rende più facile tracciare quali sono i file di terze parti. Ci sono però eccezioni a questa regola; ad esempio, se si utilizzano asset di terze parti per un personaggio, ha più senso includere questi asset all'interno della stessa cartella della scena e dello script del personaggio

2.3.2 Organizzazione del progetto

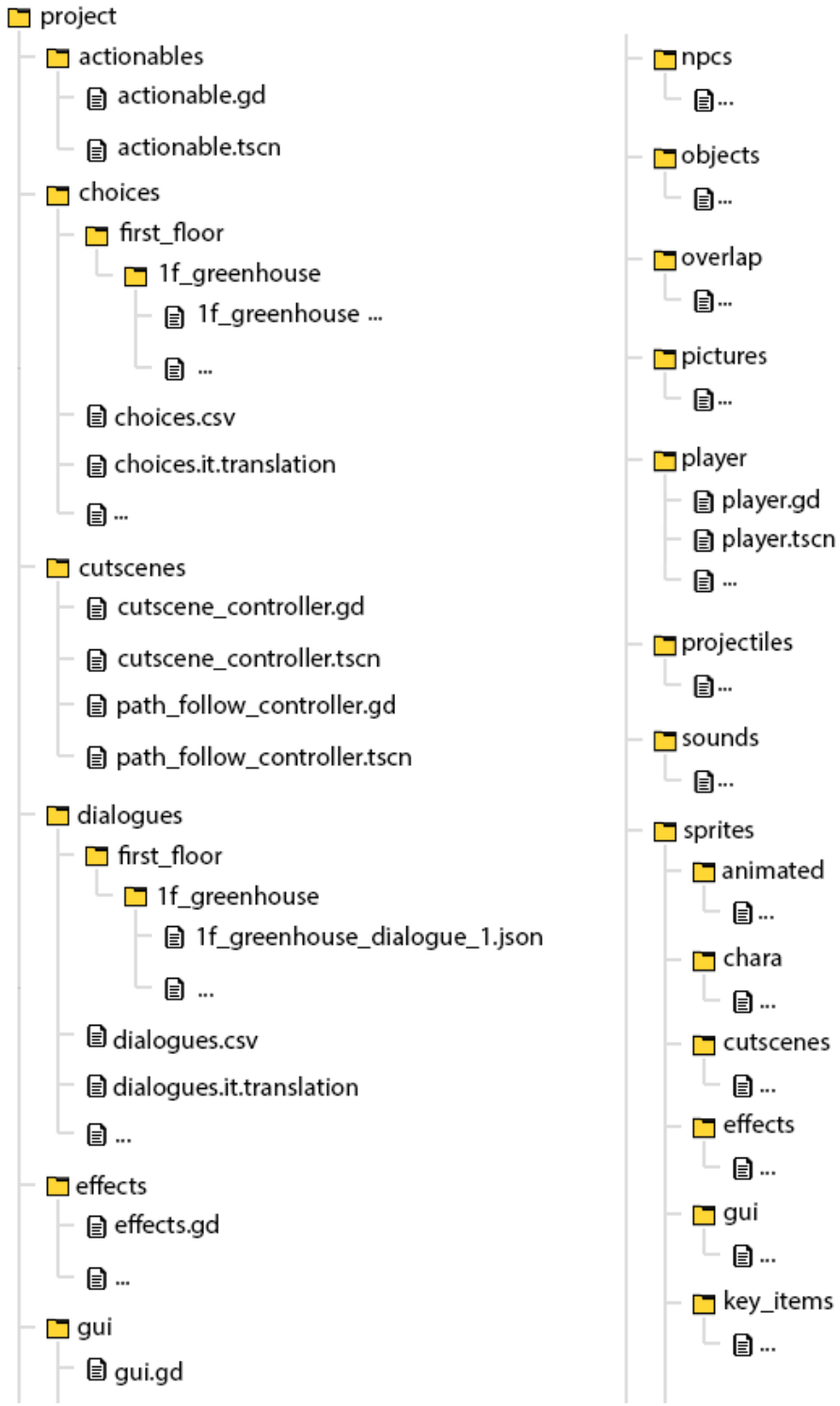
Prima di creare un nuovo progetto, ragioniamo insieme su quella che sarà la *project structure*, ovvero, la struttura, nel senso di organizzazione di file e cartelle, del progetto.

La project structure è uno dei tanti aspetti che vengono sempre sottovalutati dai programmatori novelli nonostante essa, in realtà, influisca significativamente sull'intero progetto. Pensate alla vostra casa, se siete disordinati e non mettete mai i vestiti nello stesso posto, non riuscirete mai a trovare quel calzino che state cercando; se invece siete ordinati, e mettete sempre i vestiti in posti precisi, catalogando e differenziando, quel famoso calzino lo troverete in un batter d'occhio, e anzi, addirittura li troverete entrambi. La stessa logica può essere applicata con i progetti. Se un progetto non ha una solida project structure, non si saprà mai né dove andare a posizionare nuovi file, né dove andare a cercarne di vecchi. Un progetto con una project structure solida e ben definita, invece, non solo renderà la vita del programmatore molto più semplice, ma accorcerà notevolmente anche i tempi di sviluppo.

Come abbiamo visto in precedenza, esistono molteplici approcci e template per progettare una buona project structure. Quella che adatteremo noi in questa parte di documento incentrata su Godot, è molto simile al tipo di project structure comunemente utilizzata nei progetti di OOP, ma con qualche piccola accortezza e differenza che renderanno il tipo di programmazione che andremo a fare, ovvero, lo scripting, molto più chiaro e conciso.

Prima di presentare la project structure vera e propria, cerchiamo di capire il pensiero guida che ci ha portato a questa struttura. L'obiettivo è la "*separation of concerns*" da un punto di vista progettuale, non solo dal lato del codice: vogliamo far sì che tutto sia separato e ben delineato anche sotto un punto di vista di file e cartelle. Per fare un esempio, qualcosa di strettamente legato come le cutscenes e i dialoghi, potrebbero normalmente andare sotto la stessa cartella, questo perché, appunto, cutscene senza dialoghi sono molto rare. Ma i dialoghi senza cutscene invece? Questi possono capitare, e non solo, ogni stanza avrà dei dialoghi ben definiti, dunque perché riempire la cartella dedicata alle cutscene con i file per i dialoghi? Non ha senso. È dunque meglio creare due cartelle separate: una per le cutscene e una per i dialoghi. In questo modo, tutto ciò che è legato alle cutscene in senso stretto, ovvero, tutto ciò che definisce solo e

soltanto la logica del concetto di "Cutscene" e nient'altro, vivrà all'interno della cartella `cutscenes`. Stessa cosa dicasi per la cartella `dialogues`, che conterrà invece tutti i file dei dialoghi del gioco. Alla luce di tutto questo, la project structure che utilizzeremo per la realizzazione del videogioco su Godot in questo capitolo è dunque quella riportata in figura.



Di cui ci accingiamo adesso a descriverne file e cartelle.

2.3.2.1 `.godot`

La cartella `.godot` viene generata automaticamente da Godot e può essegnere ignorata mediante `.gitignore` da Git. Contiene risorse importate e altri file che verranno utilizzati al momento dell'esportazione del videogioco.

2.3.2.2 `actionables`

Partendo dall'alto verso il basso in ordine alfabetico, la prima cartella in cui ci imbattiamo è `actionables`, in cui andranno, come già preannunciato dal nome della cartella stessa, tutta la logica legata agli `Actionable`.

Ma cos'è un `Actionable`? Per `Actionable` intendiamo tutto ciò con cui il player può interagire, come può essere un vaso, un barile, una chiave, una porta, ecc. . . .

Un `Actionable` definisce dunque, in poche parole, un oggetto interagibile dal giocatore.

Questa cartella è in realtà molto scarna di contenuto; conterrà difatti solamente la scena e il file di scripting in GDScript dell'oggetto `Actionable`.

2.3.2.3 `choices`

La seconda cartella nell'ordine è `choices`, la quale conterrà tutti i file legati alle scelte che il player è chiamato a compiere. Ad esempio, utilizzare o no una chiave per aprire una determinata porta è una scelta che apparirà al giocatore sotto forma di domande come "*Vuoi usare la chiave?*". Queste domande, e i risultati (outcome) delle rispettive scelte (ovvero aprire o no la porta, ecc...), rientrano tutti nella logica che definisce una scelta.

La cartella `choices` conterrà dunque il file CSV con i dialoghi di tutte scelte (domande, risposte, ecc...), e una serie di file JSON, organizzati prima in livelli e poi in stanze, che definiscono le azioni e le conseguenze di ciascuna scelta presente all'interno del gioco.

2.3.2.4 `cutscenes`

La cartella `cutscenes`, come già accennato nell'esempio, avrà al suo interno solamente tutto ciò che è strettamente necessario per una cutscene, e solo per quella, nient'altro.

Questa cartella conterrà dunque la scena e lo script dell'oggetto `Cutscene Controller` del nostro videogioco, che vedremo più avanti, ma che per adesso basti sapere che si occupa di controllare la buona riuscita di una cutscene e di comunicare all'"esterno" tutti i cambiamenti che la riguardano, come, ad esempio, inizio e fine; e la scena e lo script dell'oggetto `PathFollowController`, il cui unico compito è quello di far muovere i personaggi secondo percorsi specifici e predeterminati.

2.3.2.5 `dialogues`

La cartella `dialogues`, proprio come la cartella `choices`, conterrà tutti i file legati ai dialoghi di gioco, nello specifico, il file CSV con i dialoghi dei personaggi, e una serie di file JSON, organizzati prima in livelli e poi in stanze, che definiscono l'ordine dei dialoghi e li suddividono in micro-conversazioni.

2.3.2.6 `effects`

La cartella `effects` conterrà al suo interno lo script e le scene che estendono dall'oggetto `Effect`, come ad esempio può essere l'effetto che compare quando un nemico viene sconfitto, quando si utilizza un attacco, oppure ancora quando si viene colpiti, ecc....

L'oggetto `Effect` è un oggetto generico che contiene tutta la logica necessaria al corretto funzionamento degli effetti di gioco. Tutti gli effetti del nostro videogioco, come gli esempi appena citati, infatti, estendono da questa classe.

2.3.2.7 `gui`

`gui` è una delle cartelle più affollate del progetto.

Come si può ben intuire dal nome, essa conterrà al suo interno tutti gli oggetti che formano la GUI del nostro videogioco: barra della vita, barra del mana, controlli mobile, inventario, ecc....

In pratica, tutto ciò che rientra nel concetto di "GUI" si trova all'interno di questa cartella.

Tra i vari file di scene e script che compongono i singoli oggetti della GUI, prestiamo particolarmente attenzione al file CSV con tutte le scritte e i titoli che compaiono nei vari componenti della GUI (scritte come "Item", "Inventario", ecc...); e al file di script della GUI vera e propria, incaricato di gestire ogni singolo elemento di quest'ultima e di occuparsi anche di altre azioni, come ad esempio il nascondere i controlli mobile quando si è su desktop e viceversa.

2.3.2.8 `key_items`

Nella cartella `key_items` andremo invece ad inserire tutti quei file che riguardano solamente quegli item dell'inventario del player che possono essere letti tramite un'apposita schermata dell'inventario stesso, come ad esempio lettere, diari, fogli di carta, ecc....

I file degli item che non possono essere letti non verranno contenuti in questa cartella.

`key_items` conterrà dunque un file JSON per ognuno di questi particolari oggetti dell'inventario, e un singolo file CSV con tutte le righe di testo che compongono ogni item.

2.3.2.9 `levels`

Il vero fulcro del gioco è la cartella `levels`, nella quale verrano immagazzinati tutti i file dei vari livelli di gioco.

Ogni livello di gioco avrà infatti la propria cartella, all'interno della quale andremo a salvare tutti quegli script utili per far fuzionare la stanza e altri script necessari, ad esempio, per interagire con gli oggetti e le cutscene di quel particolare livello.

Per comprendere meglio, consideriamo la stanza "*2f_nursery*", lì dove "*2f*" sta per "*second floor*", poiché questa è una stanza del secondo piano della magione del videogioco. La stanza "*2f_nursery*" avrà una propria cartella chiamata `2f_nursery`, dentro la quale vi sarà il file della stanza che si occupa del set up di ogni singolo elemento della stanza stessa, anche dopo che c'è stato del progresso nel gioco (come un oggetto che è scomparso/apparso); e tutti i file di script per gli actionable, le cutscene, i dialoghi, gli oggetti, il path following, ecc... che riguardano solo e soltanto la stanza considerata.

Nota

Si noti che i file di script per gli actionable, le cutscene, i dialoghi, gli oggetti, il path following, ecc... non sono gli stessi di cui abbiamo parlato per le altre cartelle, (poiché quelli sono contenuti, appunto, nelle loro cartelle specifiche), ma sono bensì dei file che descrivono le interazioni e i comportamenti di tutti gli oggetti actionable, cutscene, object e così via, che compongono la stanza in questione. Per esempio, se una stanza ha due oggetti `Actionable`, una porta e una chiave, cosa fare quando il player interagisce con la porta o con la chiave è scritto all'interno del file actionable contenuto nella cartella della stanza stessa.

2.3.2.10 `menu`

All'interno della cartella `menu` andranno invece tutti i file delle scene e degli script dei vari menu del videogioco; come ad esempio il menu di pausa, di salvataggio o quello di caricamento.

Nella cartella `menu` vi sarà inoltre una sottocartella chiamata `settings`, all'interno della quale andremo invece a collocare tutti quei file che si occupano di gestire impostazioni come il suono, la qualità grafica, ecc. . . , i quali, per rispetto della "*separation of concerns*", necessitano di essere mattoncini separati e indipendenti l'uno dall'altro. La logica che si occupa di alzare o abbassare il volume, per esempio, non deve curarsi di quello che sta facendo la logica che gestisce il cambio di lingua, per poter svolgere la propria mansione.

2.3.2.11 `mobs`

All'interno della cartella `mobs` metteremo tutti i file delle scene e degli script che compongono i vari nemici (o "mob") del videogioco.

In questa cartella vi saranno, inoltre, la sottocartella speciale `bosses`, che conterrà appunto solamente i file dei vari boss del gioco, ovvero particolare tipi di mob che hanno una barra della vita visibile e sono particolarmente difficili battere rispetto agli altri nemici; e un file CSV in cui sono contenuti i nomi di questi boss.

2.3.2.12 `notifications`

La struttura della cartella `notifications` è la stessa della cartella `choices`, con l'unica differenza appunto che si parla di "*notifiche*" e non di "*scelte*".

Nel nostro videogioco, con il termine di "*notifica*", intendiamo tutte quelle frasi come "*Hai ottenuto una chiave*", o "*"la porta è bloccata"*, o ancora "*Hai usato l'oggetto*", che compaiono a schermo e che hanno lo scopo di fornire del feedback visivo chiaro e conciso al giocatore, per meglio aiutarlo durante il suo viaggio all'interno del mondo del videogioco.

2.3.2.13 `npcs`

La cartella `npcs`, come indicato dal nome della cartella stessa, conterrà al suo interno tutti i file delle scene e degli script dei vari NPCs del gioco.

Come già definito in precedenza, un *NPC*, dall'inglese "*Non-player character*" o "*Non-Playable Character*", letteralmente "*personaggio non giocante*", è un personaggio del videogioco non direttamente controllato dal giocatore, con il quale il giocatore stesso può interagire e/o eseguire determinati tipi di azioni.

2.3.2.14 `objects`

`objects` contiene semplicemente tutti i file e le cartelle che costuiscono gli oggetti presenti nelle stanze del videogioco; oggetti come possono essere ad esempio un baule, una statua, un fiore, una armadio, e così via dicendo.

Nota

Si noti che i file degli oggetti non sono salvati all'interno di cartelle che portano il nome della stanza in cui sono collocati. Questo perché nulla vieta a uno stesso oggetto, come ad esempio lo è una porta, di essere contemporaneamente presente in due stanze diverse.

2.3.2.15 `overlap`

La cartella `overlap` è leggermente più complicata da descrivere rispetto alle altre. Al suo interno, infatti, andremo a posizionare tutti i file degli oggetti che descrivono delle *zone* particolari di mappa.

Facciamo un esempio in modo da capire meglio di cosa stiamo parlando. Quando in un videogioco attraversiamo una porta, il nostro personaggio cambia di stanza. Questo cambio di stanza è dovuto al fatto che il giocatore ha appunto camminato attraverso la porta. Ma come fare a capire quando il personaggio controllato dal giocatore si trova effettivamente dentro l'*area* della porta? È a questo che servono gli *overlap*, ovvero, oggetti che vengono posizionati sopra delle aree specifiche di mappa, delimitando così delle "*zone*" particolari che, se attraversate da un giocatore, mob o qualsiasi altro oggetto desiderato, provocano una sorta di cambiamento.

Il termine "*overlap*", "*sovrapporsi*", deriva difatti proprio dal fatto che questi oggetti vengono *sovrapposti* a delle aree di mappa.

La cartella `overlap` conterrà dunque tutti i file e le scene che descrivono questi oggetti appena discussi, come ad esempio la zona dentro il quale il player può subire danno (hurtbox), quella che rileva se un giocatore l'ha attraversata e così via dicendo.

2.3.2.16 `pictures`

La cartella `pictures` conterrà tutte le immagini utilizzate all'interno del videogioco, come ad esempio i ritratti dei personaggi, le immagini a schermo intero o quelle del menu principale, ecc. . . .

2.3.2.17 `player`

La cartella `player` avrà al suo interno i file della scena e dello script del nostro personaggio principale controllato dal giocatore, oltre quelli che si occupano di memorizzare e gestire informazioni come la vita e il mana posseduti da quest'ultimo in ogni momento del gameplay.

2.3.2.18 `projectiles`

La cartella `projectiles` sarà popolata solo e soltanto da tutti quei file che descrivono oggetti catalogabili come "*proiettili*", ovvero oggetti il cui movimento imita il concetto di moto del proiettile.

Oggetti come proiettili di pistola, raggi laser, sfere di energia, ecc. . . , sono quindi memorizzati all'interno di questa cartella.

2.3.2.19 `sounds`

La cartella `sounds` ha la stessa struttura di quella `pictures`, con l'unica differenza che si parla di suoni e non di immagini.

2.3.2.20 `sprites`

La cartella `sprites` conterrà tutti gli spritesheet del nostro videogioco.

Uno *spritesheet* è un'immagine che consente di creare animazioni poiché composta da numerose immagini (*sprite*) più piccole integrate in una formazione a griglia.

La nostra cartella `sprites`, inoltre, è ulteriormente divisa, a seconda della tipologia di spritesheet che dobbiamo memorizzare, in:

- `animated`: gli oggetti "animati", ovvero che si muovono in loop, come ad esempio fanno i fiori mossi dal vento;
- `chara`: i vari personaggi;
- `cutscenes`: le animazioni che formano le cutscene;
- `effects`: gli effetti di luce, ombra, ecc...;
- `gui`: gli elementi grafici della GUI;
- `key_items`: le icone degli item dell'inventario;
- `map`: i tileset che compongono le mappe dei livelli del videogioco;
- `menu`: gli elementi grafici dei vari menu del videogioco;
- `mobs`: i vari nemici (la sottocartella `bosses` al suo interno è utilizzata per dividere con chiarezza i mob normali dai boss);
- `objects`: oggetti che per la maggiorparte sono già presenti all'interno di alcuni tileset della cartella `map`, ma che si ha necessità di averli separati per creare determinati effetti di gioco, come ad esempio il far comparire o scomparire un particolare oggetto in una stanza;
- `system`: elementi grafici che non fanno parte della GUI ma che aiutano a far capire al giocatore cosa deve fare. Rientrano in questa cartella sprites che rappresentano, ad esempio, i tasti della tastiera o le freccie direzionali.

2.3.2.21 `system`

Nella cartella `system` andremo a posizionare tutti quei file che si occupano della gestione del videogioco; come ad esempio fanno i file di logica della transizione da una stanza all'altra o quelli che memorizzano il progresso del giocatore.

Tutti ciò che impatta sul videogioco in senso generale, che sia esso un'impostazione grafica, la luce emessa dal giocatore o addirittura la musica riprodotta, finisce in questa cartella.

2.3.2.21.1 `inventory`

La cartella `system` ha inoltre al suo interno una sottocartella chiamata `inventory`, in cui, come da nome, andranno posizionati tutti quei file che si occupano della gestione dell'inventario del giocatore, come ad esempio il file CSV con i nomi degli item o il file JSON con le informazioni dettagliate degli item stessi.

2.3.2.22 `themes`

Nella cartella `themes` verranno salvati tutti i file generati da Godot che descrivono i nostri temi personalizzati.

2.3.2.23 `utils`

La cartella `utils` raccoglie al suo interno tutte le classi base da cui estendono oggetti più specifici come ad esempio può essere un NPC o una porta (si, può sembrare strano ma anche le porte hanno della logica dietro!).

All'interno di questa cartella è anche presente un file `utils.gd` (da cui il nome della cartella stessa) che contiene una serie di funzioni generali (come ad esempio funzioni che calcolano la somma tra gli elementi di due array, ecc. . .) utilizzate da più classi (magari completamente scollegate tra di loro) del nostro videogioco e che, quindi, proprio per questo motivo, non ha senso ripetere nel codice di ognuna di queste classi.

2.3.2.24 `.gitattributes`

Il file `.gitattributes` viene automaticamente generato da Godot se si sceglie Git come VCS nella schermata di creazione del progetto del Project Manager.

In poche parole, questo file è un semplice documento di testo che associa attributi e comportamenti a specifici tipi di file. Nella sua forma più elementare, esso configura come vengono gestiti quei file che fanno match con un certo pattern (solitamente estensioni di file). Scenari comuni includono anche la configurazione di come vengono gestiti gli avanzamenti di riga, se il documento viene trattato come binario (non differenziabile), e il fornire elaborazioni speciali subito prima di un commit o subito dopo un checkout.

2.3.2.25 `.gitignore`

Il file `.gitignore` viene automaticamente generato da Godot se si sceglie Git come VCS nella schermata di creazione del progetto del Project Manager. Questo file è utilizzato in una repository git per ignorare quei file e quelle cartelle non necessarie al progetto per funzionare, e che quindi non saranno memorizzate nella repository remota.

La tipologia di file che normalmente vengono ignorati sono principalmente file temporanei e quei file che non dovrebbero mai essere versionati.

2.3.2.26 `export_presets.cfg`

Un file generato da Godot che contiene tutte le informazioni necessarie all'engine per esportare il videogioco.

Il file contiene anche le opzioni di esportazione espresse da noi programmatori mediante l'apposita schermata di esportazione dell'editor.

2.3.2.27 `icon.svg`

L'icona del videogioco con il suo relativo file `icon.svg.import` autogenerato da Godot.

2.3.2.28 `project.godot`

Il file progetto di Godot che contiene tutte nostre le preferenze ed è utilizzato per aprire il progetto di un videogioco nell'editor di Godot.

2.3.2.29 `README.md`

Il file `README.md` viene automaticamente generato da Godot se si sceglie Git come VCS nella schermata di creazione del progetto del Project Manager.

Questo file funge da primo punto di contatto con quegli utenti e/o sviluppatori che vogliono comprendere lo scopo, il setup e l'utilizzo di un progetto. Per capirci, è l'ultima sezione che viene mostrata quando si apre la pagina di un repository di GitHub.