**PortraitGenerator**

**1. Pourquoi limiter maxSimilarAttributes entre 1 et 10 dans GeneratePortraits ?**

Cette limite évite des cas où la comparaison n’a pas de sens ou est trop permissive. En dessous de 1, on ne peut pas vérifier la similarité, et au-dessus de 10, la restriction serait trop faible, laissant passer trop de portraits presque identiques.

**2. Quel problème si on crée une nouvelle instance de Random à chaque appel de GetRandomName ?**

Créer plusieurs fois Random rapidement produit souvent les mêmes nombres car ils utilisent la même base (seed). Cela cause des répétitions non voulues dans la génération aléatoire.

**3. Pourquoi limiter le nombre d’essais dans GeneratePortraits ?**

Sans limite, si on ne trouve pas un portrait assez différent, la boucle tournerait sans fin, bloquant le programme. Une limite protège contre ce blocage.

**4. Quel est le rôle de selectedNames ?**

Il garde la trace des noms déjà utilisés pour ne pas les répéter, ce qui garantit que chaque portrait a un nom unique.

**5. Comment fonctionne la sélection pondérée dans GetRandomAsset ?**

Chaque rareté a une chance différente d’être choisie, proportionnelle à un poids. La somme des poids est ajustée pour que la sélection soit juste selon ces probabilités.

**6. Pourquoi utiliser filesByRarity plutôt qu’un simple filtrage ?**

Cela organise clairement les fichiers selon leur rareté, ce qui facilite la sélection pondérée et évite des erreurs de mélange.

**7. Pourquoi au moins 48 noms dans les fichiers ?**

Pour pouvoir créer assez de portraits uniques sans manquer de noms, sinon certains portraits auraient des noms répétés.

**8. Que doit faire Clone() pour éviter des bugs ?**

Elle doit copier toutes les données du portrait, pas seulement l’adresse mémoire, pour que les deux portraits soient complètement indépendants.

**9. Comment gérer la génération de milliers de portraits ?**

Il faudrait optimiser le code, éviter des répétitions inutiles, peut-être générer par blocs, et garder en mémoire les portraits déjà créés pour éviter de refaire les mêmes.

**10. Pourquoi la rareté “legendary” a une faible probabilité ?**

Pour que ces portraits soient spéciaux et rares, ce qui rend le jeu plus intéressant et valorise leur découverte.

**11. Quels risques liés aux fichiers dans GetRandomAsset ?**

Le dossier peut être vide, ou les fichiers peuvent manquer ou être mal nommés, ce qui provoquerait une erreur. Il faut vérifier leur présence et gérer les erreurs.

**12. Peut-on améliorer la sélection des noms pour plus de genres ?**

Oui, en ajoutant plus de fichiers de noms pour différents genres, et en adaptant le code pour choisir dans le bon fichier selon le genre.

**13. Comment tester l’unicité des portraits générés ?**

On peut écrire un test qui compare tous les portraits entre eux pour vérifier qu’aucun ne dépasse la limite de similarité fixée.

**14. Que se passe-t-il si le fichier de noms utilise un autre séparateur que la virgule ?**

Le code ne reconnaîtra pas bien les noms, ce qui peut créer des erreurs ou des noms mal lus. Il faudrait adapter le séparateur en fonction du fichier.

**15. Pourquoi gérer les exceptions dans LoadNames ?**

Pour éviter que le programme plante si le fichier est manquant ou illisible, et pouvoir informer l’utilisateur du problème sans bloquer tout le programme.

Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Ce code trie d’abord les catégories valides, ajuste leurs probabilités pour qu’elles totalisent 100%, puis réalise un tirage au sort respectant ces probabilités, et enfin choisit un fichier dans la catégorie tirée.

Détail :  
Filtrer les catégories disponibles :

On commence par ne retenir que les catégories de fichiers qui contiennent au moins un élément. Par exemple, s’il n’y a pas de fichiers “legendary”, cette catégorie est exclue du tirage. Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Calculer le poids total :

On additionne les poids (probabilités) définis pour chaque catégorie disponible, afin de pouvoir normaliser ces poids par la suite. Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Normaliser les poids :

Chaque poids est divisé par la somme totale pour que la somme des probabilités devienne égale à 1. Cela garantit que la sélection soit juste et cohérente, même si certaines catégories manquent. Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Tirage aléatoire pondéré :

On génère un nombre aléatoire entre 0 et 1 (roll).

On parcourt les catégories en ajoutant leurs probabilités accumulées (cumulative).

Dès que roll est inférieur à cette somme cumulative, on choisit la catégorie correspondante (selectedRarity) et on arrête la boucle. Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Sélection finale dans la catégorie choisie :

Dans la catégorie sélectionnée, on choisit au hasard un fichier parmi ceux disponibles, et on le retourne. Une image contenant texte, capture d’écran, logiciel

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.

Ce code extrait, nettoie, filtre et déduplique une liste de noms à partir d’une chaîne texte où les noms sont séparés par des virgules.

Détail :

**.Split(',')**

Divise la chaîne content en plusieurs morceaux, en utilisant la virgule comme séparateur. Cela crée un tableau de chaînes (ex : une liste de noms séparés par des virgules).

**.Select(n => n.Trim())**

Pour chaque élément n du tableau, enlève les espaces inutiles au début et à la fin (les "blancs") afin de nettoyer les noms.

**.Where(n => !string.IsNullOrWhiteSpace(n))**

Garde seulement les éléments qui ne sont pas vides ni composés uniquement d’espaces. Cela élimine les chaînes vides ou contenant uniquement des espaces.

**.Distinct()**

Supprime les doublons, pour ne garder qu’une seule occurrence de chaque nom.

**.ToList()**

Transforme le résultat en une liste (objet List<string>) prête à être utilisée dans le programme.

**PortraitRenderer**

**1. Pourquoi utilise-t-on un dictionnaire (Dictionary<string, Texture2D>) pour stocker les textures ?**

Pour éviter de charger plusieurs fois la même texture, ce qui économise la mémoire GPU et améliore la performance.

**2. Que se passe-t-il si une texture n’est pas trouvée sur le disque lors du chargement ?**

Le chargement est ignoré et un message d’erreur est affiché, évitant ainsi une exception ou un crash.

**3. Pourquoi la méthode LoadValidTexture vérifie-t-elle si la texture est déjà chargée avant de la recharger ?**

Pour empêcher les doublons dans la mémoire, garantissant que chaque texture est chargée une seule fois.

**4. Expliquez la fonction et l’intérêt de GenerateAndAssignPortraitTexture. Pourquoi combiner plusieurs textures en une seule ?**

Cette méthode crée une texture composite en superposant tous les calques du portrait. Cela est utile pour l’export ou pour cacher les calculs lors du rendu, améliorant les performances lors d’affichages répétitifs.

**5. Comment la méthode DrawPortrait gère-t-elle le cas où un portrait est éliminé ?**

Elle applique un filtre couleur noir opaque (new Color(0, 0, 0, 255)) pour rendre le portrait visible mais visuellement distinct (assombri), indiquant l’élimination.

**6. Quelles sont les conséquences d’appeler UnloadAll et dans quels cas doit-on le faire ?**

Cela libère toute la mémoire GPU utilisée par les textures, ce qui est nécessaire lors de la fermeture du jeu ou du changement de scène pour éviter les fuites mémoire.

**7. Pourquoi utiliser SetTextureFilter(texture, TextureFilter.Bilinear) après le chargement d’une texture ?**

Pour appliquer un filtrage lissé qui améliore le rendu visuel des textures, évitant un aspect pixelisé quand elles sont redimensionnées.

**8. Que se passe-t-il si LoadValidTexture reçoit un chemin vide ou nul ?**

La fonction affiche un message d’erreur et ne tente pas de charger la texture, évitant un plantage.

**9. Pourquoi LoadValidTexture utilise-t-elle File.Exists(path) ? Que se passe-t-il sans cette vérification ?**

Pour s’assurer que le fichier existe réellement avant de charger, sinon le chargement échouerait ou générerait une exception.

**10. Dans GenerateAndAssignPortraitTexture, pourquoi utilise-t-on RenderTexture2D et BeginTextureMode / EndTextureMode ?**

Cela permet de dessiner hors écran sur une texture cible, afin de créer une image composite sans l’afficher directement.

**11. Quelle est la différence entre dessiner directement dans DrawPortrait et dessiner dans GenerateAndAssignPortraitTexture ?**

DrawPortrait dessine en temps réel à l’écran, tandis que GenerateAndAssignPortraitTexture précompose la texture pour un usage futur, évitant de superposer les calques à chaque rendu.

**12. Pourquoi vérifier si portrait.portraitTexture.Id != 0 avant de décharger une texture existante ?**

Pour s’assurer que la texture est bien chargée (valide) avant de tenter de la libérer, évitant ainsi des erreurs.

**13. Comment la méthode DrawPortrait gère-t-elle les calques ? Est-ce que l’ordre d’affichage est important ?**

Oui, l’ordre dans le tableau layers définit la superposition des textures : la peau en premier, puis les vêtements, accessoires, etc., pour un rendu cohérent.



Cela améliore la qualité visuelle de la texture lorsqu’elle est agrandie ou réduite en lissant les pixels. Au lieu d’avoir un rendu pixellisé ou grossier, le filtrage bilinéaire crée une transition plus douce entre les couleurs, rendant l’image plus nette et agréable à l’œil.

**Bonus : Qu’est-ce qu’un binding ? (Raylib-cs, C#, C, etc…)**