Pour répondre à ces exigences sur l'utilisation des shaders dans Unity, voici un guide détaillé sur chaque point :

1. Être capable d’utiliser les shaders

a. Modifier l’alpha dans un shader pour faire un shader transparent

Créer un Shader de Surface Transparent :

Dans Unity, crée un nouveau shader en allant dans Assets > Create > Shader > Unlit Shader.

Remplace le code par celui-ci pour un shader transparent :

shader

Copier le code

Shader "Custom/TransparentShader"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

\_Color ("Color", Color) = (1,1,1,1)

}

SubShader

{

Tags { "Queue"="Transparent" "RenderType"="Transparent" }

LOD 200

CGPROGRAM

#pragma surface surf Lambert alpha

sampler2D \_MainTex;

fixed4 \_Color;

struct Input

{

float2 uv\_MainTex;

};

void surf (Input IN, inout SurfaceOutput o)

{

fixed4 c = tex2D (\_MainTex, IN.uv\_MainTex) \* \_Color;

o.Albedo = c.rgb;

o.Alpha = c.a;

}

ENDCG

}

FallBack "Diffuse"

}

Ce shader utilise une texture (\_MainTex) et une couleur (\_Color) pour définir la transparence.

b. Utiliser plusieurs textures dans un shader

Créer un Shader avec Plusieurs Textures :

Modifie le shader pour accepter deux textures :

shader

Copier le code

Shader "Custom/MultipleTextures"

{

Properties

{

\_MainTex ("Main Texture", 2D) = "white" {}

\_SecondTex ("Second Texture", 2D) = "white" {}

}

SubShader

{

Tags { "RenderType"="Opaque" }

LOD 200

CGPROGRAM

#pragma surface surf Lambert

sampler2D \_MainTex;

sampler2D \_SecondTex;

struct Input

{

float2 uv\_MainTex;

float2 uv\_SecondTex;

};

void surf (Input IN, inout SurfaceOutput o)

{

fixed4 mainTex = tex2D (\_MainTex, IN.uv\_MainTex);

fixed4 secondTex = tex2D (\_SecondTex, IN.uv\_SecondTex);

o.Albedo = lerp(mainTex.rgb, secondTex.rgb, 0.5); // Combine textures

}

ENDCG

}

FallBack "Diffuse"

}

c. Faire un post-processing shader

Créer un Post-Processing Shader :

Crée un shader pour post-processing :

shader

Copier le code

Shader "Custom/PostProcessingShader"

{

Properties

{

\_MainTex ("Base (RGB)", 2D) = "white" {}

}

SubShader

{

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

sampler2D \_MainTex;

float4 \_MainTex\_TexelSize;

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

};

v2f vert (appdata\_base v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = v.texcoord;

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

fixed4 col = tex2D(\_MainTex, i.uv);

col.rgb = 1.0 - col.rgb; // Example: Invert colors

return col;

}

ENDCG

}

}

}

Applique ce shader via un script de post-processing attaché à la caméra.

d. Modifier les vertices à l’aide d’un shader

Créer un Shader pour Modifier les Sommets :

Utilise un shader de vertex displacement :

shader

Copier le code

Shader "Custom/VertexDisplacement"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

\_Displacement ("Displacement", Float) = 0.1

}

SubShader

{

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

uniform sampler2D \_MainTex;

uniform float \_Displacement;

struct appdata

{

float4 vertex : POSITION;

float2 uv : TEXCOORD0;

};

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

};

v2f vert (appdata v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = v.texcoord;

float4 displacement = tex2Dlod(\_MainTex, float4(v.uv, 0, 0));

o.pos.xyz += displacement.rgb \* \_Displacement;

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

return tex2D(\_MainTex, i.uv);

}

ENDCG

}

}

}

e. Modifier le facteur de Tesselation dans un shader

Créer un Shader avec Tesselation :

Utilise un shader de tessellation :

shader

Copier le code

Shader "Custom/TessellationShader"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

\_Tess ("Tessellation Factor", Range(1, 32)) = 1

}

SubShader

{

Tags { "RenderType"="Opaque" }

LOD 400

CGPROGRAM

#pragma surface surf Lambert tessellate:\_Tess

sampler2D \_MainTex;

struct Input

{

float2 uv\_MainTex;

};

void surf (Input IN, inout SurfaceOutput o)

{

fixed4 c = tex2D (\_MainTex, IN.uv\_MainTex);

o.Albedo = c.rgb;

}

ENDCG

}

FallBack "Diffuse"

}

Assure-toi que ton matériel supporte la tessellation (DX11 ou supérieur).

f. Comprendre la structure d’input d’un shader et utiliser ses propriétés

Structure d'Input :

L'input d'un shader définit les données que le shader reçoit :

shader

Copier le code

struct Input

{

float2 uv\_MainTex;

float3 worldPos;

};

Les propriétés peuvent être utilisées comme IN.uv\_MainTex pour accéder aux coordonnées UV ou IN.worldPos pour la position mondiale.

g. Écrire dans une texture à l’aide d’un shader

Créer un Shader pour Écrire dans une Texture :

Exemple d’un shader qui écrit dans une texture rendue :

shader

Copier le code

Shader "Custom/RenderToTexture"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

}

SubShader

{

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

sampler2D \_MainTex;

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

};

v2f vert (appdata\_base v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = v.texcoord;

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

fixed4 col = tex2D(\_MainTex, i.uv);

col.rgb = col.rgb \* 0.5; // Example: Darken color

return col;

}

ENDCG

}

}

}

Utilise ce shader dans une caméra avec un Render Texture pour obtenir l'effet.

Ressources pour Apprendre les Shaders

Documentation Unity sur les shaders : Unity Shader Reference

Tutoriels sur les shaders :

Catlike Coding Tutorials

Shader Tutorial by Ronja

1. NetworkManager

Fonction : Gère les connexions et la configuration du réseau.

Configuration :

Ajoute le composant NetworkManager à un GameObject dans la scène.

Configure les paramètres du NetworkManager tels que les adresses de connexion, le mode de démarrage (host, server, client) et les scènes à charger pour les joueurs.

Documentation : NetworkManager

2. NetworkObject

Fonction : Identifie les objets réseau, permettant leur synchronisation entre le serveur et les clients.

Configuration :

Ajoute le composant NetworkObject à chaque GameObject que vous souhaitez synchroniser sur le réseau.

Assurez-vous que chaque objet avec un NetworkObject a un identifiant unique et est enregistré dans le NetworkManager.

Documentation : NetworkObject

3. NetworkTransform

Fonction : Synchronise la position, la rotation et l'échelle des objets sur le réseau.

Configuration :

Ajoute le composant NetworkTransform aux objets ayant un NetworkObject.

Configure les paramètres de synchronisation pour déterminer la fréquence et la précision de la mise à jour des transformations.

Documentation : NetworkTransform

4. ClientNetworkTransform

Fonction : Permet la synchronisation autorisée par le propriétaire des transformations d'un objet.

Configuration :

Utilise ClientNetworkTransform pour des objets dont les transformations sont contrôlées par les clients.

Configure les paramètres pour s'assurer que seuls les clients autorisés peuvent modifier les transformations.

Documentation : ClientNetworkTransform

5. NetworkRigidbody

Fonction : Synchronise les propriétés physiques des objets rigides sur le réseau.

Configuration :

Ajoute le composant NetworkRigidbody aux objets ayant un Rigidbody et un NetworkObject.

Configure les paramètres pour synchroniser les propriétés physiques telles que la vitesse et la position.

Documentation : NetworkRigidbody

6. NetworkVariable

Fonction : Permet de synchroniser des variables spécifiques entre le serveur et les clients.

Configuration :

Déclare une NetworkVariable dans votre script C# et attachez le script à un GameObject avec un NetworkObject.

Configure les paramètres de la NetworkVariable pour définir les conditions de mise à jour et de synchronisation.

Exemple :

csharp

Copier le code

public class Player : NetworkBehaviour

{

public NetworkVariable<int> health = new NetworkVariable<int>(100);

}

Documentation : NetworkVariable

7. NetworkPrefabsList

Fonction : Gère la liste des prefabs réseau utilisés par le NetworkManager.

Configuration :

Dans le NetworkManager, ajoutez vos prefabs réseau à la liste NetworkPrefabsList.

Assurez-vous que chaque prefab ajouté a un NetworkObject et est correctement configuré pour la synchronisation réseau.

Documentation : NetworkPrefabsList

Exemple de configuration d'un jeu multijoueur simple

Initialisation du NetworkManager :

Crée un GameObject nommé NetworkManager et ajoute-lui le composant NetworkManager.

Configure les scènes par défaut pour le client et le serveur dans les paramètres du NetworkManager.

Configuration des prefabs :

Crée des prefabs pour les objets joueurs et autres objets synchronisés.

Ajoute un NetworkObject et les composants nécessaires (NetworkTransform, NetworkRigidbody, etc.) à chaque prefab.

Ajoute ces prefabs à la liste NetworkPrefabsList du NetworkManager.

Scripts de synchronisation :

Crée des scripts pour gérer les comportements réseau et attache-les aux prefabs.

Utilise NetworkVariable pour synchroniser les données nécessaires.

Lancement du jeu :

Crée des boutons ou des commandes pour démarrer le jeu en tant qu'hôte, serveur ou client.

Utilise les méthodes du NetworkManager pour gérer les connexions :

csharp

Copier le code

public class NetworkController : MonoBehaviour

{

public NetworkManager networkManager;

public void StartHost()

{

networkManager.StartHost();

}

public void StartClient()

{

networkManager.StartClient();

}

public void StartServer()

{

networkManager.StartServer();

}

}

NetworkBehaviour

NetworkBehaviour est une classe de base pour les comportements de scripts réseau dans Unity. En l'utilisant, vous pouvez accéder aux fonctionnalités réseau de base telles que vérifier si l'objet est contrôlé par le client, le serveur, ou le propriétaire, et réagir aux événements réseau comme le spawn et le despawn.

Fonctions de base

1. IsClient

Description : Indique si l'instance de jeu actuelle est un client.

Utilisation :

csharp

Copier le code

public class MyNetworkBehaviour : NetworkBehaviour

{

void Update()

{

if (IsClient)

{

// Code spécifique pour le client

Debug.Log("This is running on the client.");

}

}

}

2. IsServer

Description : Indique si l'instance de jeu actuelle est le serveur.

Utilisation :

csharp

Copier le code

public class MyNetworkBehaviour : NetworkBehaviour

{

void Update()

{

if (IsServer)

{

// Code spécifique pour le serveur

Debug.Log("This is running on the server.");

}

}

}

3. IsOwner

Description : Indique si l'instance de jeu actuelle est le propriétaire de l'objet réseau.

Utilisation :

csharp

Copier le code

public class MyNetworkBehaviour : NetworkBehaviour

{

void Update()

{

if (IsOwner)

{

// Code spécifique pour le propriétaire de l'objet

Debug.Log("This object is owned by this client.");

}

}

}

4. OnNetworkSpawn

Description : Appelé lorsque l'objet réseau est spawné (créé) sur le réseau.

Utilisation :

csharp

Copier le code

public class MyNetworkBehaviour : NetworkBehaviour

{

public override void OnNetworkSpawn()

{

base.OnNetworkSpawn();

// Code à exécuter lorsque l'objet est spawné

Debug.Log("Network object spawned.");

}

}

5. OnNetworkDespawn

Description : Appelé lorsque l'objet réseau est despawné (supprimé) du réseau.

Utilisation :

csharp

Copier le code

public class MyNetworkBehaviour : NetworkBehaviour

{

public override void OnNetworkDespawn()

{

base.OnNetworkDespawn();

// Code à exécuter lorsque l'objet est despawné

Debug.Log("Network object despawned.");

}

}

Exemple pratique d'utilisation

Voici un exemple pratique intégrant toutes ces fonctions dans un script Unity pour un objet réseau :

csharp

Copier le code

using Unity.Netcode;

using UnityEngine;

public class PlayerController : NetworkBehaviour

{

public float moveSpeed = 5f;

private void Update()

{

if (IsClient && IsOwner)

{

HandleMovement();

}

}

private void HandleMovement()

{

float moveX = Input.GetAxis("Horizontal");

float moveY = Input.GetAxis("Vertical");

Vector3 move = new Vector3(moveX, 0, moveY) \* moveSpeed \* Time.deltaTime;

transform.position += move;

}

public override void OnNetworkSpawn()

{

base.OnNetworkSpawn();

if (IsServer)

{

Debug.Log("Player object spawned on the server.");

}

else if (IsClient)

{

Debug.Log("Player object spawned on the client.");

}

}

public override void OnNetworkDespawn()

{

base.OnNetworkDespawn();

Debug.Log("Player object despawned.");

}

}

Liens vers la documentation

Pour plus d'informations et des exemples détaillés, consulte la documentation officielle de Unity sur NetworkBehaviour :

NetworkBehaviour Documentation

1. Instancier des objets dans un contexte multijoueur

Pour instancier des objets dans un contexte multijoueur, vous devez utiliser le NetworkManager et des prefabs enregistrés. Voici comment le faire :

Étape 1 : Préparer le prefab

Créez un prefab de l'objet que vous souhaitez instancier.

Ajoutez un NetworkObject à ce prefab.

Étape 2 : Enregistrer le prefab dans le NetworkManager

Ouvrez le NetworkManager.

Dans la section NetworkPrefabs, ajoutez le prefab que vous avez créé.

Étape 3 : Instancier l'objet sur le serveur

csharp

Copier le code

using Unity.Netcode;

using UnityEngine;

public class GameManager : NetworkBehaviour

{

public GameObject playerPrefab;

public override void OnNetworkSpawn()

{

if (IsServer)

{

// Instancier un objet réseau

GameObject playerInstance = Instantiate(playerPrefab);

playerInstance.GetComponent<NetworkObject>().Spawn();

}

}

}

2. Faire bouger les objets de manière uniforme entre les divers clients

Pour synchroniser les mouvements des objets entre les clients, utilisez NetworkTransform. Voici un exemple pour un joueur :

Étape 1 : Ajouter un NetworkTransform au prefab

Ajoutez un NetworkTransform au prefab de votre joueur.

Étape 2 : Déplacer l'objet uniquement sur le client propriétaire

csharp

Copier le code

using Unity.Netcode;

using UnityEngine;

public class PlayerController : NetworkBehaviour

{

public float moveSpeed = 5f;

private void Update()

{

if (IsOwner)

{

HandleMovement();

}

}

private void HandleMovement()

{

float moveX = Input.GetAxis("Horizontal");

float moveY = Input.GetAxis("Vertical");

Vector3 move = new Vector3(moveX, 0, moveY) \* moveSpeed \* Time.deltaTime;

transform.position += move;

}

}

3. Exécuter une commande sur le serveur seulement

Pour exécuter une commande sur le serveur depuis un client, utilisez ServerRpc. Voici un exemple :

csharp

Copier le code

using Unity.Netcode;

using UnityEngine;

public class PlayerController : NetworkBehaviour

{

private void Update()

{

if (IsOwner && Input.GetKeyDown(KeyCode.Space))

{

JumpServerRpc();

}

}

[ServerRpc]

private void JumpServerRpc(ServerRpcParams rpcParams = default)

{

// Cette méthode s'exécute uniquement sur le serveur

Debug.Log("Jump command received on the server.");

}

}

4. Exécuter une commande sur les clients

Pour exécuter une commande sur tous les clients depuis le serveur, utilisez ClientRpc. Voici un exemple :

csharp

Copier le code

using Unity.Netcode;

using UnityEngine;

public class PlayerController : NetworkBehaviour

{

private void Update()

{

if (IsOwner && Input.GetKeyDown(KeyCode.Space))

{

JumpServerRpc();

}

}

[ServerRpc]

private void JumpServerRpc(ServerRpcParams rpcParams = default)

{

// Cette méthode s'exécute uniquement sur le serveur

Debug.Log("Jump command received on the server.");

NotifyClientsJumpClientRpc();

}

[ClientRpc]

private void NotifyClientsJumpClientRpc(ClientRpcParams rpcParams = default)

{

// Cette méthode s'exécute sur tous les clients

Debug.Log("Jump command executed on all clients.");

// Ajoutez ici la logique pour animer ou faire bouger le joueur sur tous les clients

}

}

Liens vers la documentation

Pour plus de détails et des exemples supplémentaires, consultez la documentation officielle de Unity sur ces sujets :

ServerRpc

ClientRpc

Perlin Noise

Connaissance du Perlin Noise

Perlin Noise est une technique de génération de bruit cohérent développé par Ken Perlin. Il est souvent utilisé dans les jeux vidéo et les applications graphiques pour créer des textures naturelles, des terrains, et d'autres effets visuels. Contrairement au bruit aléatoire, le Perlin Noise produit des variations douces et continues, ce qui le rend idéal pour simuler des phénomènes naturels.

Utilisation du Perlin Noise

En Unity, vous pouvez utiliser Mathf.PerlinNoise pour générer du bruit de Perlin. Voici un exemple pour générer un terrain simple :

csharp

Copier le code

using UnityEngine;

public class PerlinNoiseTerrain : MonoBehaviour

{

public int width = 256;

public int height = 256;

public float scale = 20f;

void Start()

{

Terrain terrain = GetComponent<Terrain>();

terrain.terrainData = GenerateTerrain(terrain.terrainData);

}

TerrainData GenerateTerrain(TerrainData terrainData)

{

terrainData.heightmapResolution = width + 1;

terrainData.size = new Vector3(width, 50, height);

terrainData.SetHeights(0, 0, GenerateHeights());

return terrainData;

}

float[,] GenerateHeights()

{

float[,] heights = new float[width, height];

for (int x = 0; x < width; x++)

{

for (int y = 0; y < height; y++)

{

heights[x, y] = CalculateHeight(x, y);

}

}

return heights;

}

float CalculateHeight(int x, int y)

{

float xCoord = (float)x / width \* scale;

float yCoord = (float)y / height \* scale;

return Mathf.PerlinNoise(xCoord, yCoord);

}

}

Dans cet exemple :

width et height définissent la taille du terrain.

scale contrôle la fréquence du bruit.

Mathf.PerlinNoise est utilisé pour générer des valeurs de hauteur pour chaque point du terrain.

Sauvegarde et Chargement des Données

Comprendre les Fonctions de Sauvegarde et de Chargement

Pour sauvegarder et charger des données dans Unity, vous pouvez utiliser la sérialisation en C#. Voici un aperçu des étapes nécessaires :

Créer une classe de données : Définir les données à sauvegarder.

Sérialiser les données : Convertir les données en un format qui peut être enregistré (comme JSON).

Enregistrer les données : Écrire les données sérialisées dans un fichier.

Charger les données : Lire le fichier et désérialiser les données pour les utiliser dans le jeu.

Exemple de Sauvegarde et de Chargement

Créer une classe de données :

csharp

Copier le code

[System.Serializable]

public class PlayerData

{

public int level;

public float health;

public Vector3 position;

}

Sérialiser et Enregistrer les données :

csharp

Copier le code

using System.IO;

using UnityEngine;

public class SaveSystem : MonoBehaviour

{

public PlayerData playerData;

public void SavePlayer()

{

string json = JsonUtility.ToJson(playerData);

File.WriteAllText(Application.persistentDataPath + "/playerdata.json", json);

}

}

Charger et Désérialiser les données :

csharp

Copier le code

using System.IO;

using UnityEngine;

public class SaveSystem : MonoBehaviour

{

public PlayerData playerData;

public void LoadPlayer()

{

string path = Application.persistentDataPath + "/playerdata.json";

if (File.Exists(path))

{

string json = File.ReadAllText(path);

playerData = JsonUtility.FromJson<PlayerData>(json);

}

}

}

Structure de Sauvegarde Cohérente

Pour une structure de sauvegarde cohérente, il est essentiel de bien organiser les données et de gérer les versions des sauvegardes si nécessaire. Voici quelques conseils :

Classe de Sauvegarde : Utilisez des classes pour structurer vos données de manière logique.

Versioning : Ajoutez un champ de version pour gérer les changements dans la structure des données.

Modularité : Séparez les données en modules si votre jeu est complexe (par exemple, PlayerData, GameSettings, etc.).

Exemple Avancé de Structure de Sauvegarde

Classe de Sauvegarde avec Versioning :

csharp

Copier le code

[System.Serializable]

public class SaveData

{

public int version = 1;

public PlayerData playerData;

public GameSettings gameSettings;

}

GameSettings Class :

csharp

Copier le code

[System.Serializable]

public class GameSettings

{

public float volume;

public string difficulty;

}

Sauvegarder et Charger avec Versioning :

csharp

Copier le code

using System.IO;

using UnityEngine;

public class SaveSystem : MonoBehaviour

{

public SaveData saveData;

public void SaveGame()

{

saveData.version = 1; // Versioning

string json = JsonUtility.ToJson(saveData);

File.WriteAllText(Application.persistentDataPath + "/savegame.json", json);

}

public void LoadGame()

{

string path = Application.persistentDataPath + "/savegame.json";

if (File.Exists(path))

{

string json = File.ReadAllText(path);

saveData = JsonUtility.FromJson<SaveData>(json);

// Gestion de la version si nécessaire

if (saveData.version == 1)

{

// Migrer ou ajuster les données si nécessaire

}

}

}

}

Modélisation de base dans Blender 3.4

Pour effectuer de simples modélisations dans Blender 3.4, il est essentiel de connaître quelques opérations de base. Voici un guide pour chaque opération :

Ajout de meshes (maillages)

Ajouter un mesh :

Appuyez sur Shift + A pour ouvrir le menu d'ajout.

Sélectionnez le type de mesh que vous souhaitez ajouter (par exemple, Cube, Sphere).

Modification de vertices (sommets)

Sélectionner les sommets :

Entrez en mode Edit en appuyant sur Tab.

Sélectionnez les sommets en utilisant l'outil de sélection (B pour la boîte de sélection, C pour le pinceau de sélection).

Déplacer (G), Redimensionner (S), et Faire pivoter (R) :

G pour déplacer les sommets.

S pour redimensionner.

R pour faire pivoter.

Extrusion (E)

Extruder :

Sélectionnez les sommets, arêtes, ou faces.

Appuyez sur E et déplacez la souris pour extruder.

Incrustation (I)

Incruster :

Sélectionnez une face.

Appuyez sur I et déplacez la souris pour incruster.

Coupe en anneau (CTRL+R)

Couper en anneau :

Appuyez sur Ctrl + R.

Déplacez la souris pour prévisualiser la coupe.

Cliquez pour confirmer, puis déplacez la coupe et cliquez à nouveau pour placer.

Créer une face (ALT+F)

Créer une face :

Sélectionnez les sommets qui forment le périmètre d'une face.

Appuyez sur Alt + F pour créer une face.

Appliquer un matériel

Ajouter un matériel :

En mode Object, sélectionnez votre objet.

Dans le panneau Properties, allez dans l'onglet Material.

Cliquez sur New pour ajouter un nouveau matériau et configurez-le comme souhaité.

Animation simple

Créer une animation :

En mode Object, allez dans l'onglet Animation (en haut de l'interface).

Utilisez la timeline pour définir des clés d'animation (appuyez sur I pour insérer une clé).

Exportation en FBX

Exporter en FBX :

Allez dans File > Export > FBX.

Configurez les options d'exportation et cliquez sur Export FBX.

Importation dans Unity

Importer un FBX dans Unity :

Glissez le fichier FBX dans le dossier Assets de votre projet Unity.

Unity importera automatiquement le modèle.

Autres notions dans Unity

Particle System

Configurer un Particle System :

Allez dans GameObject > Effects > Particle System.

Utilisez l'inspecteur pour configurer les paramètres du système de particules (émission, forme, vitesse, etc.).

Jouer des sons

Ajouter un son :

Importez un fichier audio dans le dossier Assets.

Ajoutez un composant AudioSource à un GameObject.

Assignez le fichier audio et utilisez Play() pour jouer le son.

Créer un singleton

Créer un singleton en C# :

csharp

Copier le code

public class SingletonExample : MonoBehaviour

{

public static SingletonExample Instance { get; private set; }

private void Awake()

{

if (Instance == null)

{

Instance = this;

DontDestroyOnLoad(gameObject);

}

else

{

Destroy(gameObject);

}

}

}

DontDestroyOnLoad

Utiliser DontDestroyOnLoad :

csharp

Copier le code

void Awake()

{

DontDestroyOnLoad(gameObject);

}

Changement de scène

Changer de scène :

csharp

Copier le code

using UnityEngine.SceneManagement;

public void ChangeScene(string sceneName)

{

SceneManager.LoadScene(sceneName);

}

Événements de collision

Détecter les collisions :

csharp

Copier le code

private void OnCollisionEnter(Collision collision)

{

// Logique lors de la collision

}

Matrice de collision

Configurer la matrice de collision :

Allez dans Edit > Project Settings > Physics.

Configurez la matrice de collision dans la section Layer Collision Matrix.

Rigidbody et Physique

Utiliser un Rigidbody :

csharp

Copier le code

private Rigidbody rb;

void Start()

{

rb = GetComponent<Rigidbody>();

}

void Update()

{

rb.AddForce(Vector3.forward \* force);

}

Raycast

Effectuer un Raycast :

csharp

Copier le code

void Update()

{

if (Physics.Raycast(transform.position, transform.forward, out RaycastHit hit))

{

// Logique lorsque le raycast touche un objet

}

}

Lumière

Configurer la lumière :

Ajoutez une source de lumière via GameObject > Light.

Configurez les paramètres dans l'inspecteur (intensité, couleur, type de lumière, etc.).

Utilisation de Time.deltaTime

Utiliser Time.deltaTime :

csharp

Copier le code

void Update()

{

transform.Translate(Vector3.forward \* speed \* Time.deltaTime);

}

Boutons et Actions

Ajouter un bouton et une action :

Créez un bouton via UI > Button.

Ajoutez un script au bouton et assignez la méthode à appeler dans l'inspecteur.

csharp

Copier le code

public void OnButtonClick()

{

// Logique lors du clic sur le bouton

}

Mettre à jour le texte de l'interface

Mettre à jour le texte :

csharp

Copier le code

using UnityEngine.UI;

public Text uiText;

void UpdateText(string newText)

{

uiText.text = newText;

}

Tilemaps et Tilesets

Configurer Tilemaps et Tilesets :

Ajoutez un Tilemap via GameObject > 2D Object > Tilemap.

Utilisez le Tile Palette pour dessiner sur le Tilemap.

Ajoutez des colliders via Tilemap Collider 2D.

Étape 1 : Créer un Plan dans la Scène

Ouvrez Unity et créez une nouvelle scène.

Ajoutez un plan :

Allez dans le menu GameObject > 3D Object > Plane.

Positionnez le plan selon vos besoins dans la scène.

Étape 2 : Créer un Unlit Shader nommé LavaShader

Créez un nouveau shader :

Dans le dossier Assets, faites un clic droit et sélectionnez Create > Shader > Unlit Shader.

Nommez-le LavaShader.

Étape 3 : Retirer tout le code concernant le fog

Ouvrez le shader et supprimez toutes les lignes concernant le brouillard.

Étape 4 : Modifier le Shader

Voici comment vous pouvez modifier votre shader pour répondre aux exigences.

csharp

Copier le code

Shader "Custom/LavaShader"

{

Properties

{

\_LavaTexture ("Texture de lave", 2D) = "white" {}

\_RockTexture ("Texture de roche", 2D) = "white" {}

\_DispFactor ("Facteur de deplacement", Range(0, 10)) = 1

\_DispTexture ("Texture de deplacement", 2D) = "white" {}

}

SubShader

{

Tags { "RenderType"="Opaque" }

LOD 100

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

struct appdata\_t

{

float4 vertex : POSITION;

float2 uv : TEXCOORD0;

float3 normal : NORMAL;

};

struct v2f

{

float2 uv : TEXCOORD0;

float4 vertex : SV\_POSITION;

};

sampler2D \_LavaTexture;

sampler2D \_RockTexture;

sampler2D \_DispTexture;

float \_DispFactor;

float4 \_LavaTexture\_ST;

float4 \_RockTexture\_ST;

v2f vert (appdata\_t v)

{

v2f o;

// Apply displacement

float4 disp = tex2D(\_DispTexture, TRANSFORM\_TEX(v.uv, \_LavaTexture\_ST));

float3 displacement = v.normal \* disp.r \* \_DispFactor;

v.vertex.xyz += displacement;

o.vertex = UnityObjectToClipPos(v.vertex);

o.uv = v.uv;

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

// Scroll the lava texture

float2 uvLava = i.uv + \_Time.y \* float2(0.1, 0.1); // Scroll in X and Y

fixed4 colLava = tex2D(\_LavaTexture, uvLava);

fixed4 colRock = tex2D(\_RockTexture, i.uv);

return lerp(colRock, colLava, colLava.a); // Combine rock and lava textures

}

ENDCG

}

}

FallBack "Diffuse"

}

Explications des Modifications

Propriétés :

Ajout des propriétés \_LavaTexture, \_RockTexture, \_DispFactor, et \_DispTexture avec les noms affichés appropriés.

Vert Shader :

Calcul de la texture de déplacement et application du déplacement le long de la normale du sommet.

Frag Shader :

Scroll de la texture de lave.

Combinaison des textures de lave et de roche en utilisant la valeur alpha de la texture de lave pour la combinaison.

Étape 5 : Créer un Matériel Utilisant le Shader

Créez un nouveau matériel :

Faites un clic droit dans le dossier Assets et sélectionnez Create > Material.

Nommez-le LavaMaterial.

Assignez le shader :

Dans l'inspecteur du matériau, changez le shader à `

Custom/LavaShader.

Étape 6 : Appliquer le Matériel à votre Plan

Assignez le matériau au plan :

Glissez le LavaMaterial que vous avez créé sur le plan dans la scène.

Compléter les Détails du Shader

Pour s'assurer que toutes les fonctionnalités demandées fonctionnent correctement, voici le code complet avec des commentaires pour chaque étape :

csharp

Copier le code

Shader "Custom/LavaShader"

{

Properties

{

\_LavaTexture ("Texture de lave", 2D) = "white" {}

\_RockTexture ("Texture de roche", 2D) = "white" {}

\_DispFactor ("Facteur de deplacement", Range(0, 10)) = 1

\_DispTexture ("Texture de deplacement", 2D) = "white" {}

}

SubShader

{

Tags { "RenderType"="Opaque" }

LOD 100

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

struct appdata\_t

{

float4 vertex : POSITION;

float2 uv : TEXCOORD0;

float3 normal : NORMAL;

};

struct v2f

{

float2 uv : TEXCOORD0;

float4 vertex : SV\_POSITION;

};

sampler2D \_LavaTexture;

sampler2D \_RockTexture;

sampler2D \_DispTexture;

float \_DispFactor;

float4 \_LavaTexture\_ST;

float4 \_RockTexture\_ST;

v2f vert (appdata\_t v)

{

v2f o;

// Apply displacement

float4 disp = tex2D(\_DispTexture, TRANSFORM\_TEX(v.uv, \_LavaTexture\_ST));

float3 displacement = v.normal \* disp.r \* \_DispFactor;

v.vertex.xyz += displacement;

o.vertex = UnityObjectToClipPos(v.vertex);

o.uv = v.uv;

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

// Scroll the lava texture

float2 uvLava = i.uv;

uvLava.x += \_Time.y \* 0.1; // Scroll in X

uvLava.y += \_Time.y \* 0.1; // Scroll in Y

fixed4 colLava = tex2D(\_LavaTexture, uvLava);

fixed4 colRock = tex2D(\_RockTexture, i.uv);

// Blend the two textures based on the alpha of the lava texture

return lerp(colRock, colLava, colLava.a);

}

ENDCG

}

}

FallBack "Diffuse"

}

Explications

Propriétés du Shader :

\_LavaTexture et \_RockTexture sont définies avec des noms lisibles pour l'inspecteur.

\_DispFactor est un facteur de déplacement avec une plage de 0 à 10.

\_DispTexture est la texture de déplacement.

Vertex Shader :

Utilise tex2D pour échantillonner la texture de déplacement.

Multiplie la valeur de déplacement par la normale du sommet et par \_DispFactor pour déplacer les sommets.

Fragment Shader :

Scrolle les coordonnées UV de la texture de lave en fonction du temps.

Mélange les textures de lave et de roche en utilisant l'alpha de la texture de lave.

Conclusion

En suivant ces étapes, vous avez créé un shader personnalisé nommé LavaShader qui :

Utilise des textures pour la lave et la roche.

Applique un défilement aux textures de lave.

Déplace les sommets du plan en utilisant une texture de déplacement et un facteur de déplacement.

Mélange les textures de manière à obtenir un effet de lave et de roche combiné.

Étape 1 : Créer un Shader Image Effect Shader nommé ShaderAim

Créez un nouveau shader :

Dans le dossier Assets, faites un clic droit et sélectionnez Create > Shader > Image Effect Shader.

Nommez-le ShaderAim.

Ouvrez le shader et modifiez-le pour répondre aux exigences :

Voici un exemple de code pour le shader :

csharp

Copier le code

Shader "Custom/ShaderAim"

{

Properties

{

\_MainTex ("Base (RGB)", 2D) = "white" {}

\_AimTexture ("Aim Texture", 2D) = "white" {}

}

SubShader

{

Tags { "RenderType"="Opaque" }

LOD 100

Pass

{

ZTest Always Cull Off ZWrite Off

Fog { Mode Off }

CGPROGRAM

#pragma vertex vert\_img

#pragma fragment frag

#include "UnityCG.cginc"

sampler2D \_MainTex;

sampler2D \_AimTexture;

float4 \_MainTex\_ST;

struct v2f

{

float4 pos : POSITION;

float2 uv : TEXCOORD0;

};

v2f vert\_img (float4 pos : POSITION, float2 uv : TEXCOORD0)

{

v2f o;

o.pos = UnityObjectToClipPos(pos);

o.uv = TRANSFORM\_TEX(uv, \_MainTex);

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

fixed4 col = tex2D(\_MainTex, i.uv);

fixed4 aim = tex2D(\_AimTexture, i.uv);

// Combine the aim texture with the main texture

fixed4 finalCol = col \* aim;

return finalCol;

}

ENDCG

}

}

FallBack "Diffuse"

}

Étape 2 : Créer un script nommé AimPostProcessing

Créez un script C# dans Unity :

Dans le dossier Assets, faites un clic droit et sélectionnez Create > C# Script.

Nommez-le AimPostProcessing.

Modifiez le script pour qu'il applique le shader à l'image de la caméra :

csharp

Copier le code

using UnityEngine;

[ExecuteInEditMode]

[RequireComponent(typeof(Camera))]

public class AimPostProcessing : MonoBehaviour

{

public Shader shader;

private Material material;

void Start()

{

if (shader == null)

{

Debug.LogError("Shader not assigned!");

enabled = false;

return;

}

if (!shader.isSupported)

{

Debug.LogError("Shader not supported on this platform!");

enabled = false;

return;

}

material = new Material(shader);

}

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material != null)

{

Graphics.Blit(src, dest, material);

}

else

{

Graphics.Blit(src, dest);

}

}

}

Étape 3 : Ajouter le script à la caméra

Ajoutez le script à la caméra :

Sélectionnez la caméra principale dans votre scène.

Faites glisser le script AimPostProcessing sur la caméra.

Assignez le shader au script :

Dans l'inspecteur de la caméra, vous verrez le script AimPostProcessing.

Assignez le ShaderAim au champ Shader.

Test et Vérification

Assurez-vous que le shader et le script fonctionnent sans être en mode Play :

Le script AimPostProcessing utilise l'attribut [ExecuteInEditMode], ce qui garantit qu'il fonctionne même lorsque le mode de jeu n'est pas activé.

Vérifiez le résultat :

Appliquez des textures à \_MainTex et \_AimTexture dans les propriétés du matériau ShaderAim.

Observez les modifications de l'image de la caméra en temps réel.

Étape 1 : Modélisation de l'Avion dans Blender

Ouvrez Blender.

Ajoutez les pièces de l'avion :

Utilisez des primitives telles que des cubes, des cylindres, et des sphères pour créer les différentes parties de l'avion (fuselage, ailes, queue, hélice).

Alignez et positionnez les pièces pour former l'avion.

Appliquez les couleurs :

Sélectionnez chaque pièce et appliquez un matériau avec la couleur appropriée.

Par exemple, utilisez un matériau bleu pour le fuselage, rouge pour les ailes, et jaune pour l'hélice.

Organisez les pièces :

Joignez toutes les pièces pour former un seul objet.

Assurez-vous que les pièces sont correctement parentées pour rester ensemble.

Exporter en FBX :

Sélectionnez l'avion complet.

Allez dans File > Export > FBX.

Configurez les options d'exportation et enregistrez le fichier.

Étape 2 : Importer le Modèle dans Unity

Ouvrez Unity et importez le modèle :

Faites glisser le fichier FBX exporté dans le dossier Assets de votre projet Unity.

Sélectionnez le modèle importé et configurez les paramètres d'importation si nécessaire.

Créer un Préfab :

Faites glisser le modèle de l'avion dans la hiérarchie.

Positionnez-le et appliquez les ajustements nécessaires (échelle, rotation).

Faites glisser l'objet de la hiérarchie vers le dossier Assets pour créer un préfab.

Étape 3 : Configuration de l'Avion

Ajoutez un Rigidbody pour affecter la gravité :

Sélectionnez l'avion dans la hiérarchie.

Allez dans Component > Physics > Rigidbody.

Ajouter une hélice :

Ajoutez un nouvel objet (cylindre) en tant qu'enfant de l'avion pour représenter l'hélice.

Positionnez et redimensionnez l'hélice.

Ajoutez un script pour faire tourner l'hélice proportionnellement à l'input de saut.

Étape 4 : Script pour Contrôler l'Avion

Créer un script C# pour l'avion :

Créez un script nommé AirplaneController et attachez-le à l'avion.

csharp

Copier le code

using UnityEngine;

public class AirplaneController : MonoBehaviour

{

public float speed = 10f;

public float rotationSpeed = 50f;

public Transform propeller;

public ParticleSystem smokeParticles;

private Rigidbody rb;

private bool isJumping;

void Start()

{

rb = GetComponent<Rigidbody>();

smokeParticles.Stop();

}

void Update()

{

// Control vertical and horizontal axis

float vertical = Input.GetAxis("Vertical");

float horizontal = Input.GetAxis("Horizontal");

transform.Rotate(Vector3.right \* vertical \* rotationSpeed \* Time.deltaTime);

transform.Rotate(Vector3.forward \* -horizontal \* rotationSpeed \* Time.deltaTime);

// Control propeller rotation and forward movement

if (Input.GetButton("Jump"))

{

isJumping = true;

propeller.Rotate(Vector3.forward \* speed \* 10 \* Time.deltaTime);

rb.AddForce(transform.forward \* speed);

if (!smokeParticles.isPlaying)

{

smokeParticles.Play();

}

}

else

{

isJumping = false;

if (smokeParticles.isPlaying)

{

smokeParticles.Stop();

}

}

}

}

Étape 5 : Particules de Fumée

Créer un système de particules :

Allez dans GameObject > Effects > Particle System.

Configurez les paramètres pour ressembler à de la fumée (couleur grise, transition vers le blanc, transparence).

Désactivez Play On Awake.

Ajouter les particules à l'avion :

Faites glisser le système de particules en tant qu'enfant de l'avion.

Positionnez-le derrière l'avion.

Assurez-vous que smokeParticles est assigné dans le script.

Étape 6 : Finalisation et Préfab

Assurez-vous que tout fonctionne correctement :

Testez les contrôles de l'avion, le mouvement de l'hélice, et les particules de fumée.

Créer le préfab final :

Faites glisser l'avion de la hiérarchie vers le dossier Assets pour créer un préfab.

Étape 1 : Configuration de la Scène

Créez un nouveau projet Unity ou utilisez le projet existant.

Créez les objets de la scène :

Joueur : Créez un objet 3D (par exemple, un cube) et nommez-le Player.

Balle : Créez un autre objet 3D (par exemple, une sphère) et nommez-le Bullet.

Ennemi : Créez un objet 3D (par exemple, un cube) et nommez-le Enemy.

Particules : Créez un système de particules pour les explosions d'ennemis.

Étape 2 : Configuration des Objets

Joueur :

Ajoutez un composant Rigidbody et cochez Is Kinematic.

Ajoutez un script PlayerController.

Balle :

Ajoutez un composant Rigidbody et cochez Is Kinematic.

Ajoutez un script BulletController.

Ennemi :

Ajoutez un composant Rigidbody et cochez Is Kinematic.

Ajoutez un script EnemyController.

Étape 3 : Scripts

PlayerController.cs

csharp

Copier le code

using UnityEngine;

public class PlayerController : MonoBehaviour

{

public float speed = 10f;

public GameObject bulletPrefab;

public Transform bulletSpawn;

private float minY, maxY;

void Start()

{

// Calculate screen boundaries

Camera camera = Camera.main;

float screenHeight = camera.orthographicSize \* 2f;

float screenWidth = screenHeight \* camera.aspect;

minY = -screenHeight / 2;

maxY = screenHeight / 2;

}

void Update()

{

// Move player up and down

float vertical = Input.GetAxis("Vertical");

Vector3 newPosition = transform.position + Vector3.up \* vertical \* speed \* Time.deltaTime;

newPosition.y = Mathf.Clamp(newPosition.y, minY, maxY);

transform.position = newPosition;

// Fire bullet

if (Input.GetButtonDown("Fire1"))

{

Fire();

}

}

void Fire()

{

GameObject bullet = Instantiate(bulletPrefab, bulletSpawn.position, bulletSpawn.rotation);

bullet.GetComponent<Rigidbody>().velocity = Vector3.right \* speed;

}

}

BulletController.cs

csharp

Copier le code

using UnityEngine;

public class BulletController : MonoBehaviour

{

public float speed = 20f;

public float lifeTime = 2f;

void Start()

{

Destroy(gameObject, lifeTime);

}

void Update()

{

transform.Translate(Vector3.right \* speed \* Time.deltaTime);

}

void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Enemy"))

{

// Instantiate particle system and destroy enemy and bullet

Instantiate(other.GetComponent<EnemyController>().explosion, other.transform.position, other.transform.rotation);

Destroy(other.gameObject);

Destroy(gameObject);

}

}

}

EnemyController.cs

csharp

Copier le code

using UnityEngine;

public class EnemyController : MonoBehaviour

{

public float speed = 5f;

public float lifeTime = 5f;

public GameObject explosion;

void Start()

{

Destroy(gameObject, lifeTime);

}

void Update()

{

transform.Translate(Vector3.left \* speed \* Time.deltaTime);

}

void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Bullet") || other.CompareTag("Player"))

{

// Instantiate particle system and destroy enemy

Instantiate(explosion, transform.position, transform.rotation);

Destroy(other.gameObject);

Destroy(gameObject);

}

}

}

Étape 4 : Particules et Préfab

Créer les Préfab :

Balle : Faites glisser l'objet Bullet dans le dossier Assets pour créer un préfab.

Ennemi : Faites glisser l'objet Enemy dans le dossier Assets pour créer un préfab.

Particules : Faites glisser le système de particules dans le dossier Assets pour créer un préfab.

Configurer les Particules :

Configurez les particules pour ressembler à de la fumée.

Assurez-vous que les particules ne suivent pas l'avion (désactivez le parentage).

Étape 5 : Spawner d'Ennemis

Créez un script pour faire apparaître les ennemis de manière aléatoire.

EnemySpawner.cs

csharp

Copier le code

using UnityEngine;

public class EnemySpawner : MonoBehaviour

{

public GameObject enemyPrefab;

public float spawnRate = 3f;

public float minY = -4f;

public float maxY = 4f;

void Start()

{

InvokeRepeating("SpawnEnemy", 2f, spawnRate);

}

void SpawnEnemy()

{

Vector3 spawnPosition = new Vector3(Camera.main.orthographicSize \* Camera.main.aspect + 1f, Random.Range(minY, maxY), 0f);

Instantiate(enemyPrefab, spawnPosition, Quaternion.identity);

}

}

Étape 6 : Configuration des Collisions

Configurer les Layers :

Créez des layers pour Player, Bullet, et Enemy.

Assignez les layers appropriés à chaque objet.

Configurer la Matrice de Collisions :

Allez dans Edit > Project Settings > Physics.

Configurez la matrice de collisions pour permettre les interactions souhaitées (par exemple, Bullet ne collise qu'avec Enemy).

Étape 7 : Finalisation et Test

Ajoutez tous les scripts aux objets appropriés (Joueur, Balle, Ennemi, Spawner).

Testez le jeu pour vérifier que tout fonctionne comme prévu :

Le joueur peut se déplacer.

Le joueur peut tirer des balles.

Les balles détruisent les ennemis.

Les ennemis apparaissent et se déplacent correctement.

Les systèmes de particules fonctionnent lors des collisions.

Étape 1 : Préparation de l'Environnement Unity

Importez le Package :

Importez Question6\_start.unitypackage dans votre projet Unity.

Étape 2 : Génération de Poissons avec Perlin Noise

FishGenerator.cs

Complétez la méthode GenerateFishes :

csharp

Copier le code

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class FishGenerator : MonoBehaviour

{

public GameObject fishPrefab;

public int numberOfFishes = 100;

public float noiseScale = 0.1f;

public float spawnRadius = 50f;

private List<GameObject> fishes = new List<GameObject>();

public void GenerateFishes()

{

DeleteFishes();

for (int i = 0; i < numberOfFishes; i++)

{

float angle = i \* Mathf.PI \* 2 / numberOfFishes;

float x = Mathf.Cos(angle) \* spawnRadius;

float y = Mathf.Sin(angle) \* spawnRadius;

Vector3 spawnPosition = new Vector3(x, y, 0) + transform.position;

GameObject fish = Instantiate(fishPrefab, spawnPosition, Quaternion.identity);

fish.transform.parent = transform;

fishes.Add(fish);

}

}

public void DeleteFishes()

{

foreach (GameObject fish in fishes)

{

Destroy(fish);

}

fishes.Clear();

}

}

Assurez-vous que DeleteFishes fonctionne :

La méthode DeleteFishes est déjà incluse ci-dessus.

Étape 3 : Génération de Mouvement avec Perlin Noise

MovementGenerator.cs

Complétez la méthode Update :

csharp

Copier le code

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class MovementGenerator : MonoBehaviour

{

public float speed = 5f;

public float noiseScale = 0.1f;

private float timer = 0f;

private float directionChangeInterval = 5f;

private Vector2 movementDirection;

void Start()

{

ChangeDirection();

}

void Update()

{

timer += Time.deltaTime;

if (timer >= directionChangeInterval)

{

ChangeDirection();

timer = 0f;

}

foreach (Transform fish in transform)

{

Rigidbody2D rb = fish.GetComponent<Rigidbody2D>();

if (rb != null)

{

float noiseX = Mathf.PerlinNoise(fish.position.x \* noiseScale, Time.time \* noiseScale);

float noiseY = Mathf.PerlinNoise(fish.position.y \* noiseScale, Time.time \* noiseScale);

Vector2 noiseMovement = new Vector2(noiseX, noiseY) \* speed;

rb.AddForce(noiseMovement);

}

}

}

void ChangeDirection()

{

float angle = Random.Range(0f, Mathf.PI \* 2);

movementDirection = new Vector2(Mathf.Cos(angle), Mathf.Sin(angle));

}

}

Étape 4 : Paramétrage dans l'Editeur Unity

Assurez-vous que les scripts sont attachés aux objets appropriés :

FishGenerator doit être attaché à un objet vide dans la scène.

MovementGenerator doit être attaché à ce même objet ou à un autre selon votre besoin.

Configurer les paramètres dans l'éditeur :

Configurez numberOfFishes, noiseScale, et spawnRadius dans FishGenerator.

Configurez speed et noiseScale dans MovementGenerator.

Étape 5 : Vérifications et Tests

Vérifiez que les poissons sont générés correctement :

Exécutez la méthode GenerateFishes depuis l'éditeur ou en mode jeu.

Vérifiez le mouvement des poissons :

Assurez-vous que les poissons se déplacent de manière fluide et changent de direction toutes les 5 secondes.

Assurez-vous que les poissons sont détruits correctement :

Exécutez la méthode DeleteFishes et vérifiez que tous les poissons sont supprimés.

Étape 1 : Configuration de Base pour le Multijoueur

Importez le Package :

Importez Question7\_start.unitypackage dans votre projet Unity.

Installer le package Netcode for GameObjects :

Dans Unity, allez dans Window -> Package Manager, cherchez Netcode for GameObjects et installez-le.

Étape 2 : Créer le Menu de Connexion

Créer le Menu :

Ajoutez un Canvas à la scène.

Ajoutez deux boutons : Host et Join.

Ajoutez un champ texte pour entrer l'adresse IP (si nécessaire).

Ajoutez un script NetworkManagerUI pour gérer les interactions du menu.

csharp

Copier le code

using UnityEngine;

using UnityEngine.UI;

using Unity.Netcode;

using Unity.Netcode.Transports.UTP;

public class NetworkManagerUI : MonoBehaviour

{

public Button hostButton;

public Button joinButton;

public InputField ipInputField;

void Start()

{

hostButton.onClick.AddListener(() =>

{

NetworkManager.Singleton.StartHost();

});

joinButton.onClick.AddListener(() =>

{

string ipAddress = ipInputField.text;

NetworkManager.Singleton.GetComponent<UnityTransport>().ConnectionData.Address = ipAddress;

NetworkManager.Singleton.StartClient();

});

}

}

Étape 3 : Configurer les Joueurs

Ajouter le Composant NetworkObject :

Ajoutez le composant NetworkObject aux avatars des joueurs (ex: Player prefab).

Configurer le Script de Contrôle du Joueur :

Ajoutez le script NetworkPlayer aux avatars des joueurs.

csharp

Copier le code

using UnityEngine;

using Unity.Netcode;

public class NetworkPlayer : NetworkBehaviour

{

public Color localPlayerColor = Color.blue;

public Color remotePlayerColor = Color.red;

private SpriteRenderer spriteRenderer;

void Start()

{

spriteRenderer = GetComponent<SpriteRenderer>();

if (IsOwner)

{

spriteRenderer.color = localPlayerColor;

}

else

{

spriteRenderer.color = remotePlayerColor;

}

}

void Update()

{

if (IsOwner)

{

float move = Input.GetAxis("Vertical") \* Time.deltaTime \* 5f;

transform.Translate(Vector3.up \* move);

}

}

}

Étape 4 : Configurer le Serveur

Spawning des Ennemis et des Balles par le Serveur :

Modifiez les scripts pour que le serveur soit responsable de la création et du mouvement des ennemis et des balles.

EnemySpawner.cs

csharp

Copier le code

using UnityEngine;

using Unity.Netcode;

public class EnemySpawner : NetworkBehaviour

{

public GameObject enemyPrefab;

public float spawnRate = 3f;

public float minY = -4f;

public float maxY = 4f;

private void Start()

{

if (IsServer)

{

InvokeRepeating(nameof(SpawnEnemy), 2f, spawnRate);

}

}

[ServerRpc]

private void SpawnEnemy()

{

Vector3 spawnPosition = new Vector3(Camera.main.orthographicSize \* Camera.main.aspect + 1f, Random.Range(minY, maxY), 0f);

GameObject enemy = Instantiate(enemyPrefab, spawnPosition, Quaternion.identity);

enemy.GetComponent<NetworkObject>().Spawn();

}

}

BulletController.cs

csharp

Copier le code

using UnityEngine;

using Unity.Netcode;

public class BulletController : NetworkBehaviour

{

public float speed = 20f;

public float lifeTime = 2f;

void Start()

{

if (IsServer)

{

Invoke(nameof(DestroySelf), lifeTime);

}

}

void Update()

{

if (IsServer)

{

transform.Translate(Vector3.right \* speed \* Time.deltaTime);

}

}

void OnTriggerEnter(Collider other)

{

if (IsServer && other.CompareTag("Enemy"))

{

// Instantiate particle system and destroy enemy and bullet

Instantiate(other.GetComponent<EnemyController>().explosion, other.transform.position, other.transform.rotation);

other.GetComponent<NetworkObject>().Despawn();

GetComponent<NetworkObject>().Despawn();

}

}

[ServerRpc]

private void DestroySelf()

{

GetComponent<NetworkObject>().Despawn();

}

}

Étape 5 : Gestion des Collisions

Configurer les collisions sur le serveur :

Utilisez le script suivant pour gérer les collisions sur le serveur.

csharp

Copier le code

using UnityEngine;

using Unity.Netcode;

public class NetworkCollisionHandler : NetworkBehaviour

{

void OnTriggerEnter(Collider other)

{

if (IsServer)

{

if (other.CompareTag("Bullet") || other.CompareTag("Player"))

{

// Instantiate particle system and destroy enemy

Instantiate(explosion, transform.position, transform.rotation);

Destroy(other.gameObject);

Destroy(gameObject);

}

}

}

}

Étape 6 : Configurer les Prefabs pour le Réseau

Ajoutez les prefabs dans la liste des prefabs du réseau :

Allez dans NetworkManager -> NetworkPrefabsList et ajoutez les prefabs Player, Enemy, et Bullet.

Étape 7 : Tests et Vérifications

Tester le Jeu :

Lancer le jeu en mode Host et Client sur deux instances différentes de Unity pour vérifier que tout fonctionne correctement.

Assurez-vous que les mouvements et les interactions sont synchronisés :

Vérifiez que les joueurs voient leurs avatars en bleu, peuvent se déplacer, et voient les autres joueurs se déplacer.

Vérifiez que les ennemis et les balles sont créés et déplacés par le serveur.

Conclusion

Avec ces étapes, vous aurez configuré votre jeu pour être jouable en multijoueur. Les joueurs peuvent créer et rejoindre des parties, déplacer leurs avatars, tirer des balles, et voir les ennemis et autres joueurs se déplacer. Le serveur est responsable de la gestion des ennemis et des balles, ainsi que des collisions.