

2ème Soutenance

Menacing Duck Studios

13 Mars 2025



Table des matières

1	Organisation	2
1.1	Répartition des tâches	2
1.2	Réunions	2
2	Plan d'action	3
2.1	Compte rendu	3
2.2	Futur	29
3	Conclusion	30

Chapitre 1

Organisation

1.1 Répartition des tâches

Au sein de l'équipe, chacun avait une tâche principale attribuée. Cependant, nous restons disponibles si l'un d'entre nous avait besoin d'aide sur un domaine.

Nous nous sommes principalement occupés du design, du menu, d'un bouton du tutoriel, du multijoueur et des compétences qu'auront chaque personnages.

1.2 Réunions

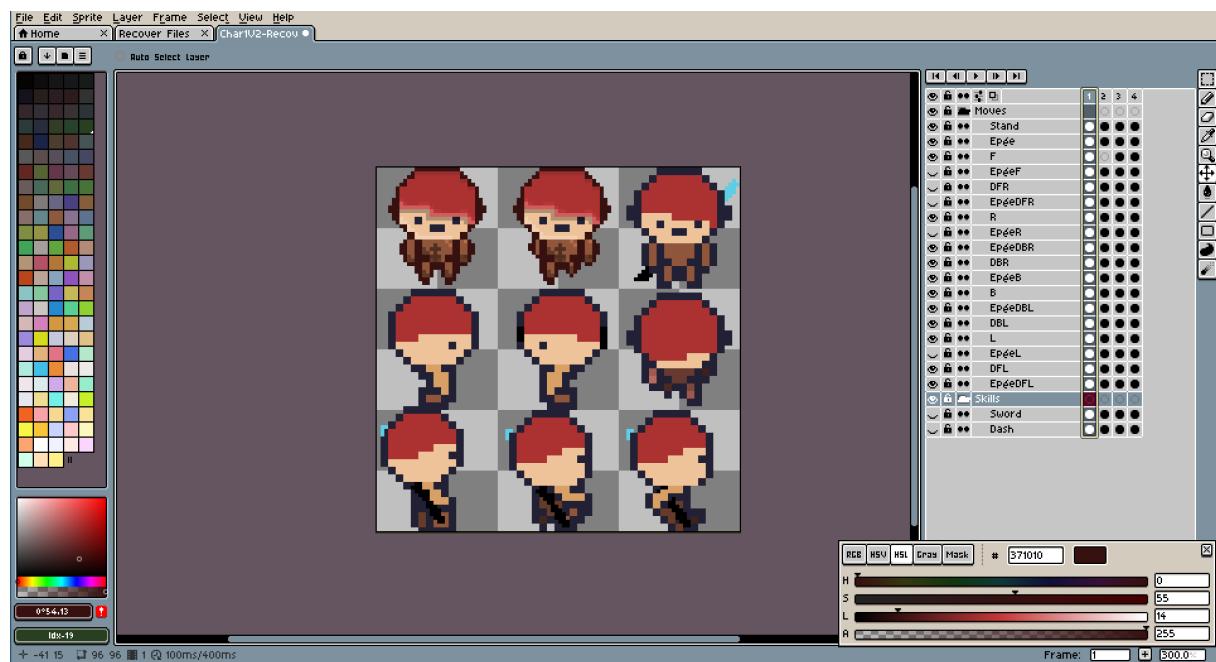
À l'aide de notre serveur discord, nous avons instauré des réunions tout au long de la journée pour s'organiser et discuter de nos progrès.

Chapitre 2

Plan d'action

2.1 Compte rendu

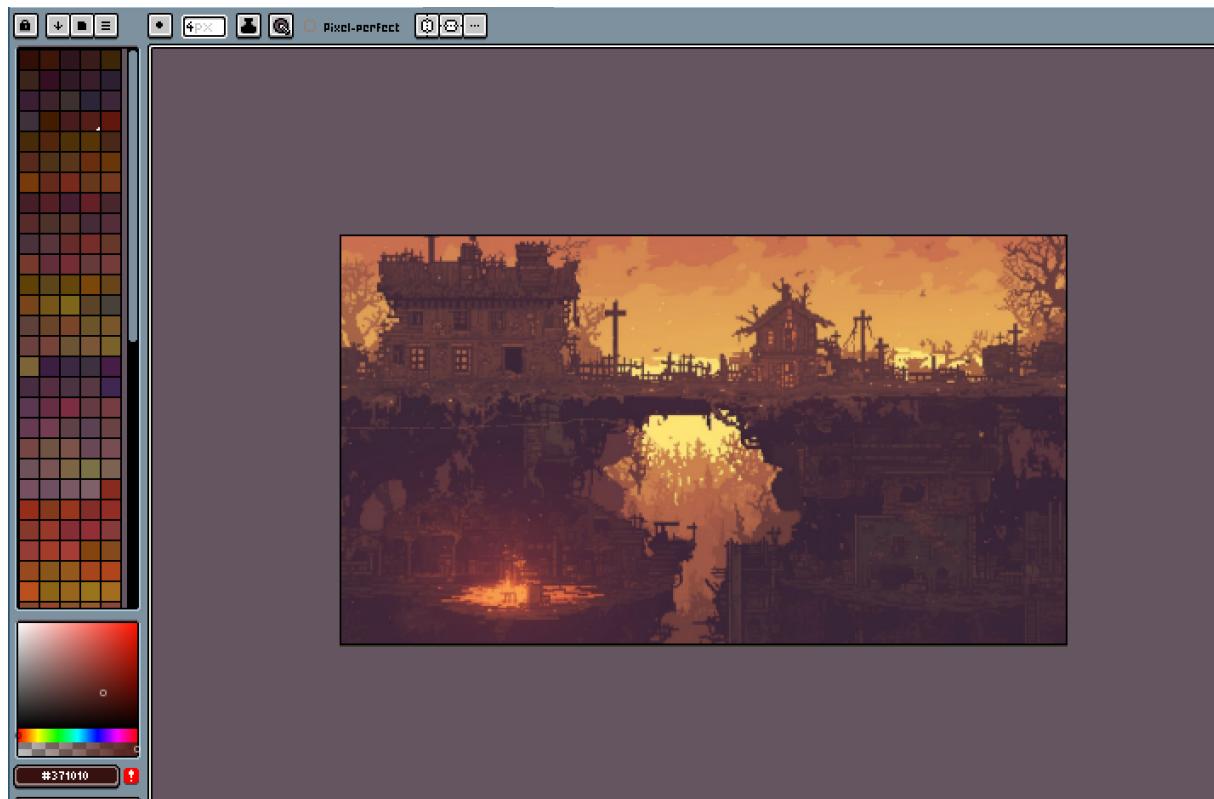
2.1.1 Designs



Tout d'abord, nous nous sommes concentrés sur le personnage, en créant 9 directions avec plusieurs frames de mouvement. Le personnage est en constante évolution, comme vous pouvez le voir ci-dessus.



Nous avons bien sûr dû nous occuper du design du menu avec les boutons et l'arrière-plan, qui a été en partie généré par IA, puis complété en pixel art.



2.1.2 Menu

Scène Menu



Pour créer les menus du jeu, nous avons développé une scène dédiée avec différents boutons représentant diverses fonctionnalités :

Play : Permet à l'utilisateur de choisir son mode de jeu.

Settings : Permet de paramétriser le jeu en fonction de ses préférences.

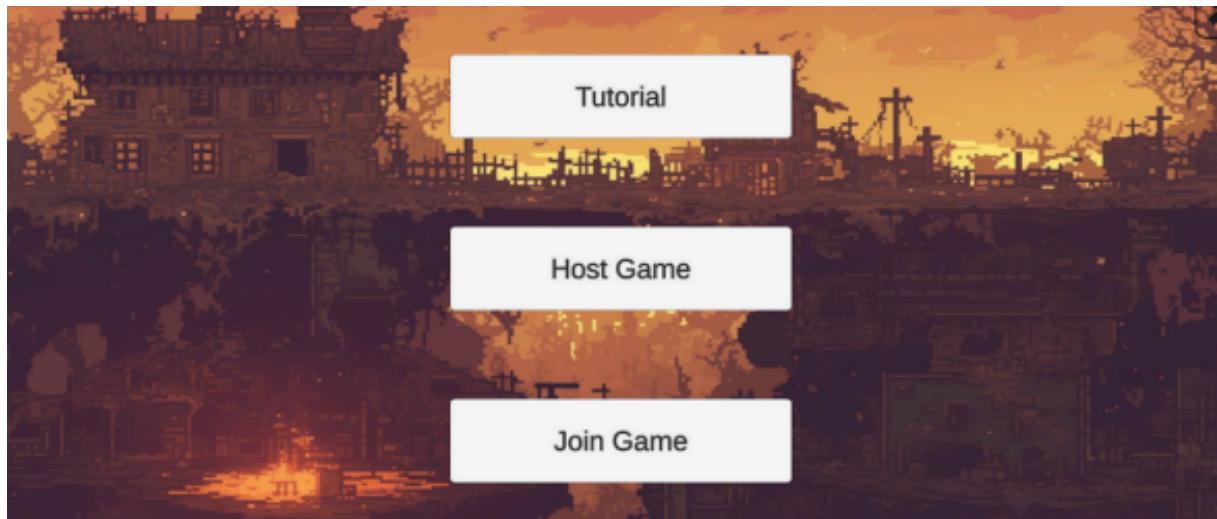
Exit : Permet de quitter l'application.

Nous avons conçu un script sur mesure pour gérer les boutons. Le voici :

```
Assets > C# MainMenu.cs
1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3
4  public class MainMenu : MonoBehaviour
5  {
6      public void playGame(){
7          SceneManager.LoadSceneAsync(1);
8      }
9      public void Settings(){
10         SceneManager.LoadSceneAsync(3);
11     }
12     public void QuitGame(){
13         Application.Quit();
14     }
15 }
16
```

Ainsi, lorsqu'on clique sur le bouton Play, cela redirige le joueur vers la scène 1. Si l'on clique sur Settings, cela mène à la scène 3. Enfin, si l'on clique sur Exit, l'application se ferme. Il suffit ensuite d'associer ces fonctions aux événements des boutons pour obtenir un premier menu fonctionnel.

Scène Play



Si l'on clique sur le bouton Play de la scène menu, on est redirigé vers cette scène. Elle contient plusieurs boutons :

Tutorial : Permet de lancer le tutoriel du jeu pour les débutants.

Host Game : Permet d'héberger une partie locale pour jouer à plusieurs.

Join Game : Permet de rejoindre une partie locale.

Back : Permet de revenir au menu principal.

Un script sur mesure gère les actions des boutons en utilisant LoadSceneAsync, comme dans la scène menu. Voici le code :

```
using UnityEngine.SceneManagement;
using Unity.Netcode;

3 references
public class MainPlay : MonoBehaviour
{
    3 references
    public static int IsHost=0;

    0 references
    public void back()
    {
        SceneManager.LoadSceneAsync(0);
    }

    0 references
    public void tutorial()
    {
        SceneManager.LoadSceneAsync(2);
    }

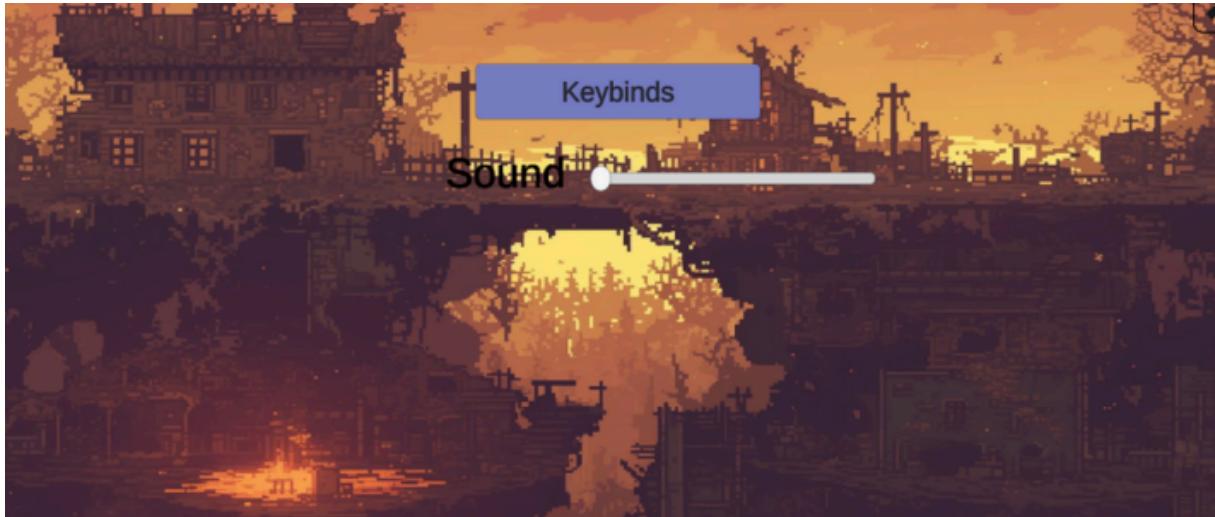
    0 references
    public void host()
    {
        IsHost = 1;
        SceneManager.LoadSceneAsync(5);
    }

    0 references
    public void join()
    {
        IsHost = 2;
        SceneManager.LoadSceneAsync(5);
    }
}
```

Les fonctions permettent de changer la scène actuelle. La seule spécificité réside dans la fonction host, qui définit une valeur à 1, tandis que la fonction join définit cette valeur à 2. Ainsi, on peut identifier si le joueur souhaite héberger ou rejoindre une partie. Après avoir relié ces fonctions aux boutons correspondants, la scène Play est terminée.

Scène Settings

La scène Settings est toujours en développement. Par exemple, pour gérer le son du jeu, il nous faut d'abord implémenter des sons, ce qui viendra plus tard. Voici la scène actuelle :



À terme, cette scène permettra de régler le son et d'autres paramètres. Pour l'instant, deux boutons sont fonctionnels :

Back : Permet de revenir au menu principal.

Keybinds : Permet d'accéder à une scène où l'on peut personnaliser ses touches.

Voici le script associé :

```
1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3  using UnityEngine.Audio;
4
5  public class MainSettings : MonoBehaviour
6  {
7      public void back(){
8          SceneManager.LoadSceneAsync(0);
9      }
10     public void Keybinds(){
11         SceneManager.LoadSceneAsync(4);
12     }
13     public void Sound()
14     {
15     }
16 }
17
18 }
```

Le principe de changement de scène est identique aux autres menus. La fonction sound n'est pas encore implémentée et est donc vide. Une fois les fonctions reliées aux boutons, la scène Settings devient fonctionnelle.

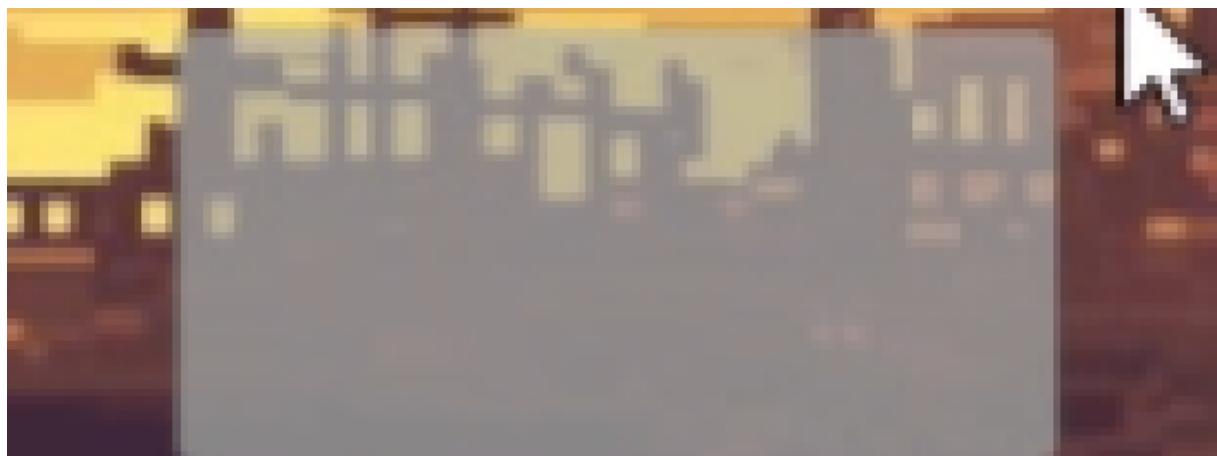
Scène KeyBinds

La scène KeyBinds diffère des autres car elle ne sert pas à rediriger le joueur, mais à lui permettre de personnaliser ses touches selon ses préférences. Voici la scène :



Elle comporte 13 boutons :

Move UP : Affiche la touche actuelle pour se déplacer vers le haut. Si sélectionné, le bouton change de couleur (devient gris), comme ceci :



Dans cet état, le joueur peut entrer une nouvelle touche. Si elle est valide, elle est enregistrée et s'affiche dans le bouton :



Les autres boutons suivent le même principe :

Move Down : Pour se déplacer vers le bas.

Move Left : Pour se déplacer vers la gauche.

Move Right : Pour se déplacer vers la droite.

Interact : Pour interagir avec l'environnement.

Inventory : Pour ouvrir l'inventaire (fonctionnalité à venir).

Utility 1, 2, 3 : Pour activer les différentes capacités du personnage.

Ultimate : Pour déclencher la capacité ultime.

Pause : Pour ouvrir le menu des paramètres (fonctionnalité à venir).

Back : Retour à la scène Settings.

Save : Sauvegarde les touches configurées.

Voici le code derrière cette scène :

```
void Start(){
    Keys.Add("up", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("up", "W")));
    Keys.Add("down", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("down", "S")));
    Keys.Add("left", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("left", "Q")));
    Keys.Add("right", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("right", "D")));
    Keys.Add("interact", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("interact", "F")));
    Keys.Add("inventory", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("inventory", "E")));
    Keys.Add("capa1", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("capa1", "F")));
    Keys.Add("capa2", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("capa2", "G")));
    Keys.Add("capa3", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("capa3", "C")));
    Keys.Add("ultimate", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("ultimate", "X")));
    Keys.Add("pause", (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("pause", "escape")));
    up.text = Keys["up"].ToString();
    down.text = Keys["down"].ToString();
    left.text = Keys["left"].ToString();
    right.text = Keys["right"].ToString();
    interact.text = Keys["interact"].ToString();
    inventory.text = Keys["inventory"].ToString();
    capa1.text = Keys["capa1"].ToString();
    capa2.text = Keys["capa2"].ToString();
    capa3.text = Keys["capa3"].ToString();
    ultimate.text = Keys["ultimate"].ToString();
    pause.text = Keys["pause"].ToString();
    SaveKeys();
}
```

Ce script initialise les touches. Si le joueur a déjà configuré des touches, elles s'affichent. Sinon, des touches par défaut sont utilisées.

```

void OnGUI()
{
    if(CurrentKey != null){
        Event e = Event.current;
        if(e.isKey){
            Keys[CurrentKey.name] = e.keyCode;
            CurrentKey.transform.GetChild(1).GetComponent<TextMeshProUGUI>().text = e.keyCode.ToString();
            CurrentKey.GetComponent<Image>().color = normal;
            CurrentKey = null;
        }
    }
}
2 references
public void SaveKeys(){
    foreach(var Key in Keys){
        PlayerPrefs.SetString(Key.Key, Key.Value.ToString());
    }
    PlayerPrefs.Save();
}
0 references
public void ChangeKey(GameObject clicked){
    if(CurrentKey != null){
        CurrentKey.GetComponent<Image>().color = normal;
    }
    CurrentKey = clicked;
    CurrentKey.GetComponent<Image>().color = Clicked;
}
0 references
void OnApplicationQuit()
{
    SaveKeys();
}

```

Cette partie met à jour les touches. Lorsqu'un bouton est sélectionné, il change de couleur pour signaler qu'il attend un input. Si un autre bouton est sélectionné, le précédent revient à sa couleur normale. Une fois la touche modifiée, le texte du bouton est mis à jour.

La fonction SaveKeys enregistre les touches personnalisées dans les PlayerPrefs de Unity. Cet outil conserve les informations du joueur même après la fermeture du jeu ou un redémarrage du PC. Ainsi, les touches restent sauvegardées même en cas de crash ou d'arrêt brutal de l'application.

2.1.3 Tutoriel

Map

Voici la map de la scène Tutorial :



Nous avons la zone de spawn :



La zone de début comprends un petit parc avec des collisions sur les bords, si l'on prends la carte avec les collisions, ça ressemble à cela:



Mouvements de la caméra

Le personnage peut se déplacer et la caméra suit le personnage, on a fait un script sur mesure pour avoir une caméra plus fluide :

```
using UnityEngine;
0 references
public class CameraFol : MonoBehaviour
{
    3 references
    public Transform target;
    2 references
    public Vector2 offset;
    1 reference
    public float smoothSpeed = 0.125f;

    0 references
    void LateUpdate()
    {
        if (target != null)
        {
            Vector3 desiredPosition = new Vector3(target.position.x + offset.x, target.position.y + offset.y, transform.position.z);
            transform.position = Vector3.Lerp(transform.position, desiredPosition, smoothSpeed);
        }
    }
}
```

Déplacement du personnage

Le code de déplacement du personnage est le suivant :

```
0 references
public class PlayerMov : MonoBehaviour
{
    1 reference
    public float speed = 10f;
    2 references
    private Rigidbody2D rb;
    2 references
    private Vector2 movement;
    4 references
    private float moveX = 0f;
    4 references
    private float moveY = 0f;

    0 references
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }
}
```

Au début, quand on initialise le joueur, on initialise aussi le rigidbody du joueur, lui permettant entre autres de gérer les collisions, mais aussi de lui donner de la vitesse, c'est pour cela que l'on fait appel au rigidbody du joueur dans la fonction

```

0 références
void FixedUpdate()
{
    if (Input.GetKey(KeyCode.Escape)){
        SceneManager.LoadSceneAsync(0);
    }

    moveX = 0f;
    moveY = 0f;

    if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("left", "A")))){
        moveX = -1f;
    }
    if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("right", "D")))){
        moveX = 1f;
    }
    if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("up", "W")))){
        moveY = 1f;
    }
    if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("down", "S")))){
        moveY = -1f;
    }

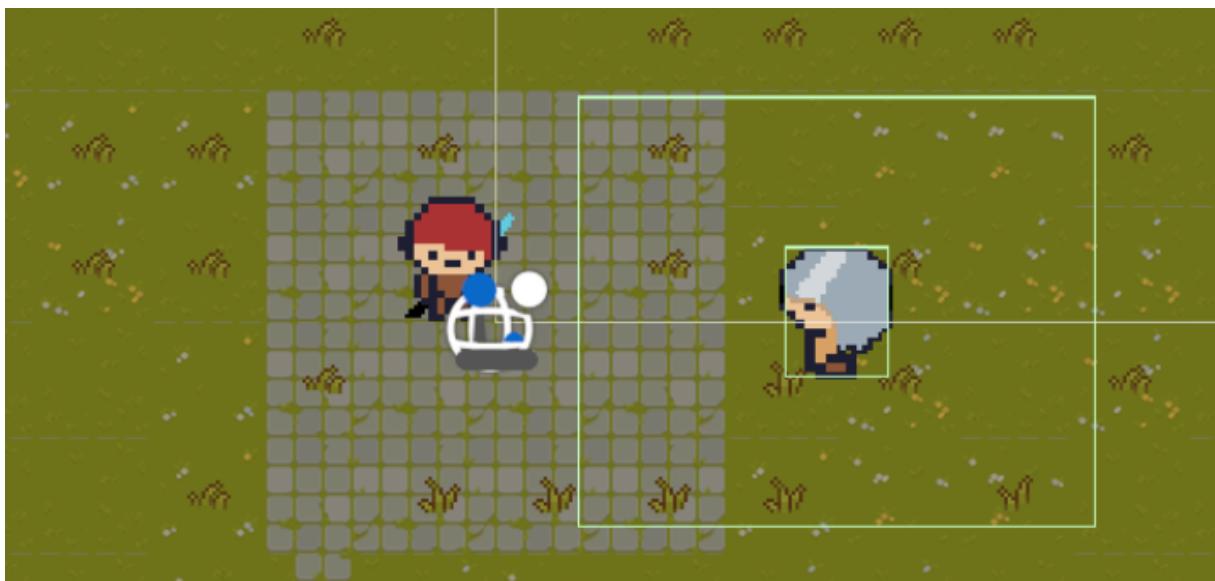
    movement = new Vector2(moveX, moveY).normalized;

    rb.linearVelocity = movement * speed;
}

```

Ici, la fonction gère le vectoring du personnage, il gère sa vitesse dans différentes directions, si la touche enregistrée dans les playerprefs d'une action est utilisée, alors, on lui donne une vitesse dans un certain axe tant que la touche est utilisée, pour cela, on lui donne une valeur dans une variable remise à 0 à chaque tour de boucle, ainsi, si aucune touche n'est utilisée, le joueur n'a pas de vitesse et donc ne se déplace pas, et si une ou plusieurs touches sont sélectionnées, les variables sont modifiées, les variables passent enfin par une dernière étape, la normalisation du vecteur déplacement, ça évite ainsi de se déplacer plus vite en diagonale que en temps normal, enfin, on applique le vecteur calculé au rigidbody du joueur, et l'on laisse Unity faire la magie. Enfin, si l'on utilise la touche escape, on revient au menu principal.

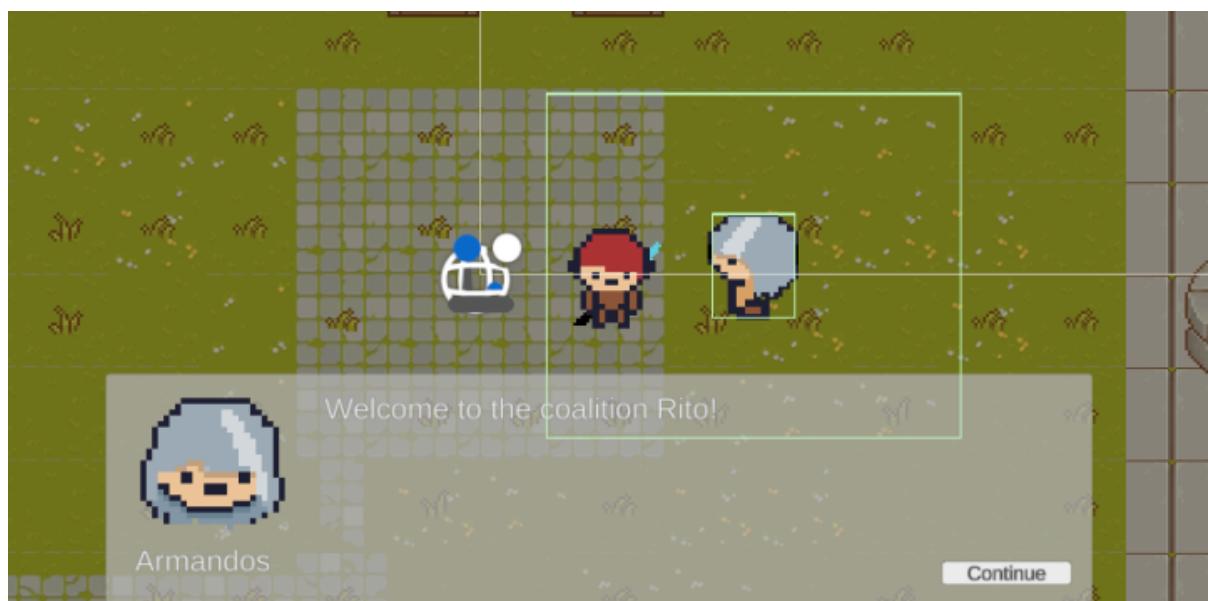
Personnage non jouable du début



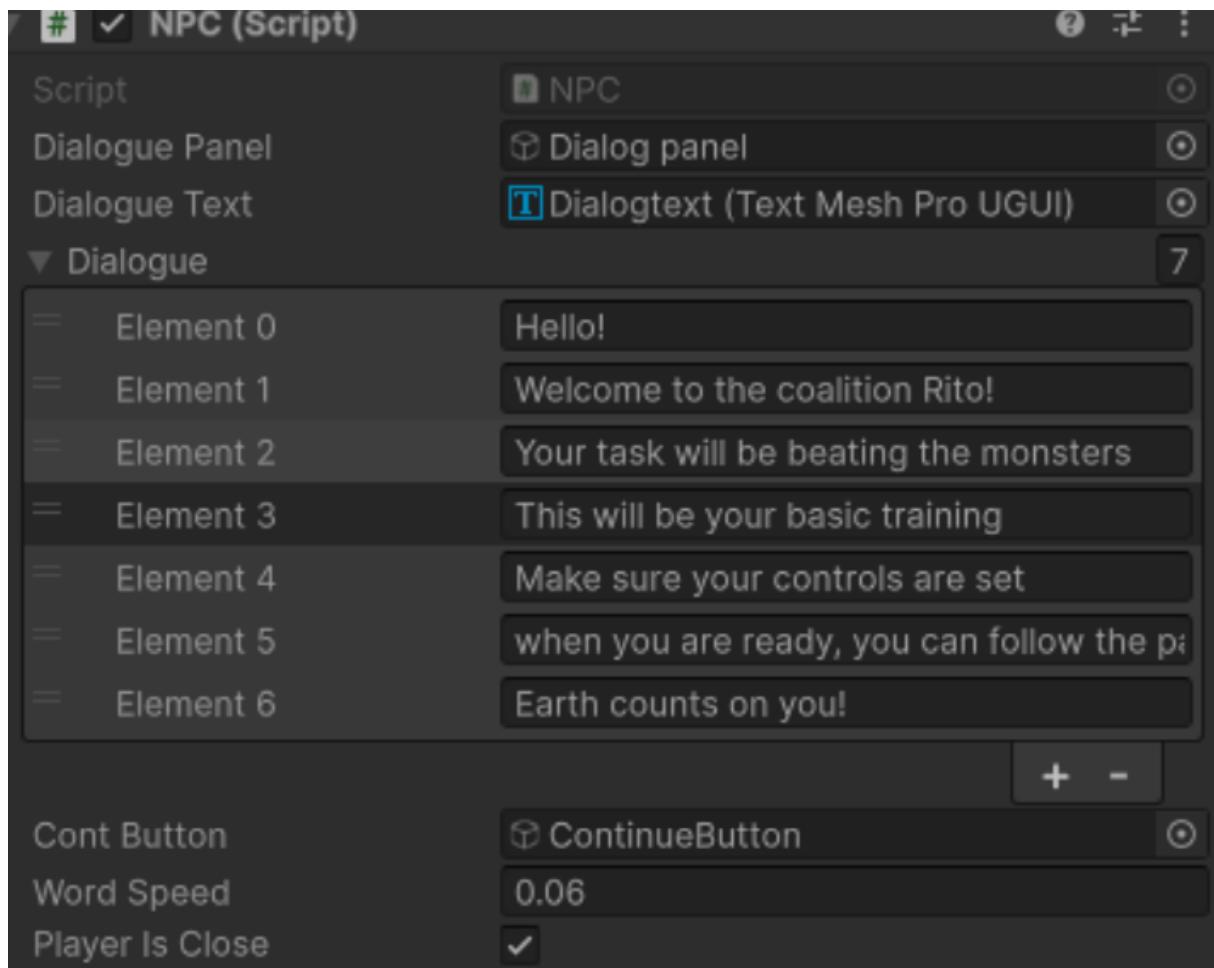
Voici les 2 zones de collision du personnage non jouable, la plus petite est une zone de collision empêchant le personnage de se déplacer dans la zone, la 2ème zone de collision sert à savoir si le joueur se trouve à distance suffisamment proche pour parler au personnage non jouable, si l'on appuie sur sa touche interactif, l'icône suivante apparaît sur l'écran :



Le personnage non jouable va donc parler au joueur, par le biais de l'icone, on peut clicker sur continuer pour avoir la suite du dialogue :



On peut paramétrer plusieurs lignes de dialogue directement dans Unity grâce au script qui sera présenté plus loin :



Enn, si le dialogue est terminé ou bien que le personnage sort de la zone de dialogue, l'icone disparaît de l'écran. Le code pour le personnage non jouable du début est le suivant :

```
0 references
void Update()
{
    KeyCode interactKey = (KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("interact", "F"));

    if (Input.GetKeyDown(interactKey) && PlayerIsClose)
    {
        if (DialoguePanel.activeInHierarchy)
        {
            ZeroText();
        }
        else
        {
            DialoguePanel.SetActive(true);
            StartCoroutine(Typing());
        }
    }

    if (DialogueText.text == Dialogue[Index])
    {
        ContButton.SetActive(true);
    }
}
```

Si l'on est dans la zone de dialogue de Begin, notre personnage non jouable, et que l'on appuie sur la touche interact, le panel de dialogue deviens visible et la première ligne de dialogue est chargée.

```

3 references
public void ZeroText(){
    DialogueText.text = "";
    Index = 0;
    DialoguePanel.SetActive(false);
}

2 references
IEnumerator Typing(){
    foreach(char letter in Dialogue[Index].ToCharArray()){
        DialogueText.text += letter;
        yield return new WaitForSeconds(WordSpeed);
    }
}

0 references
public void NextLine(){
    ContButton.SetActive(false);

    if(Index < Dialogue.Length - 1){
        Index++;
        DialogueText.text = "";
        StartCoroutine(Typing());
    }
    else{
        ZeroText();
    }
}

```

La fonction ZeroText permet de faire disparaître le panel de dialogue et réinitialise au conditions initiales. La fonction Typing permet de faire apparaître le texte de la ligne peut à peu pour rendre les discussions plus réalistes, le personnage mettra du temps à parler. La fonction NextLine permet de charger la prochaine ligne de dialogue, si il n'y a plus de dialogues à dire, alors il fait appel à ZeroText pour faire disparaître le panel de dialogue et réinitialiser les conditions initiales.

```

private void OnTriggerEnter2D(Collider2D other)
{
    Debug.Log("OnTriggerEnter2D called with: " + other.gameObject.name);

    if (other.CompareTag("Player"))
    {
        Debug.Log("Player entered the trigger!");
        PlayerIsClose = true;
    }
}

0 references
private void OnTriggerExit2D(Collider2D other)
{
    Debug.Log("OnTriggerExit2D called with: " + other.gameObject.name);

    if (other.CompareTag("Player"))
    {
        Debug.Log("Player exited the trigger!");
        PlayerIsClose = false;
        ZeroText();
    }
}

```

La fonction OnTriggerEnter2D permet de savoir si le joueur est rentré dans la zone de dialogue de Begin, et inversement, la fonction OnTriggerExit2D permet de savoir si le joueur à quitté la zone de dialogue, si le joueur quitte la zone de dialogue, la fonction fait appel à ZeroText pour faire disparaître le panel de dialogue ainsi que remettre les conditions de départ du dialogue.

2.1.4 Multijoueur

Mode

Après avoir sélectionné une option dans la scène *Play*, on récupère la variable `isHost` et on décide de lancer le joueur en tant que client ou serveur. Plus tard, un menu sera ajouté pour entrer un code de 8 caractères (l'IP en hexadécimal, plus abordable pour l'utilisateur). Exemple : `c0a80401`. Ce menu permettra de rejoindre une partie en tant

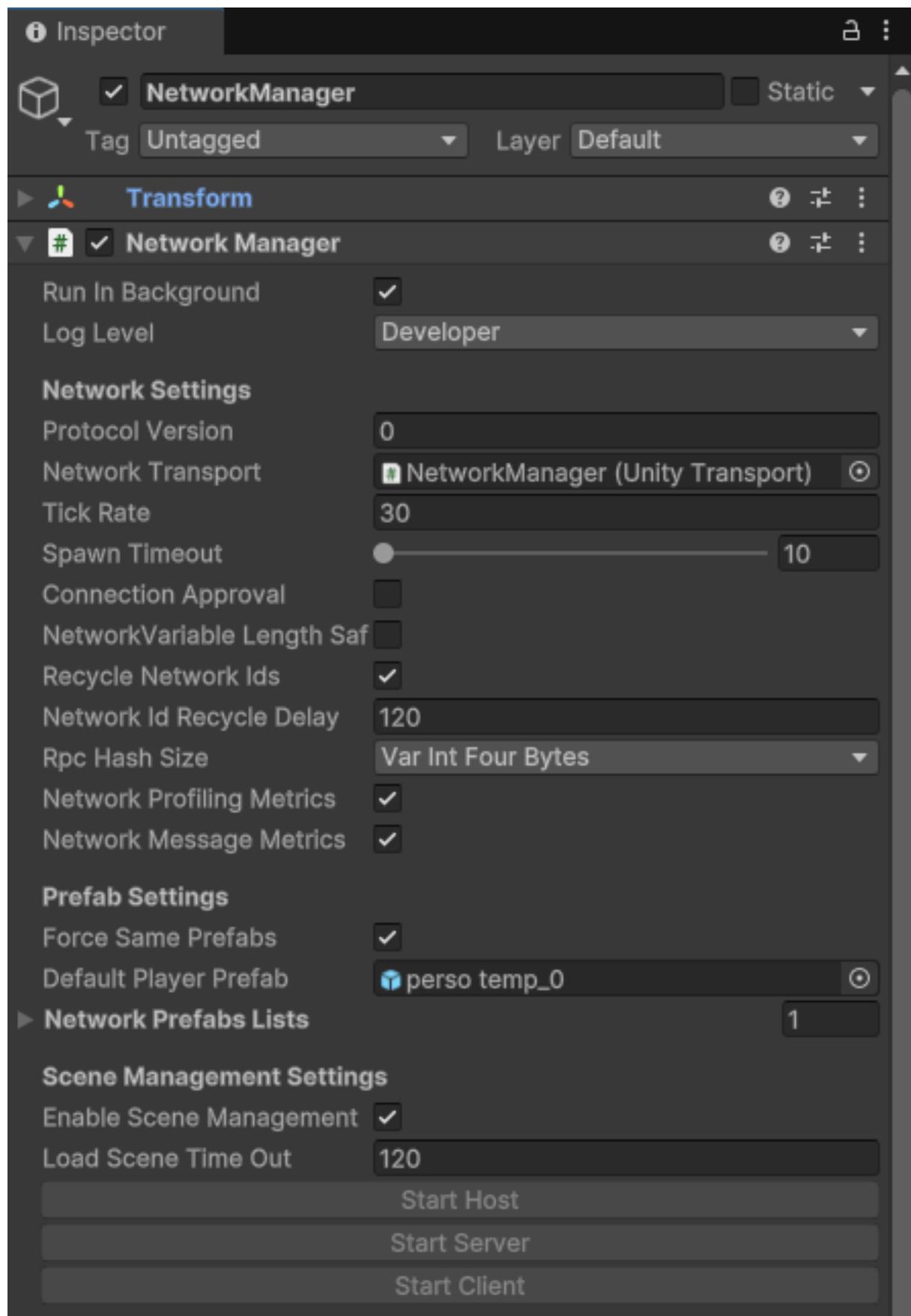
que client.

```
if (MainPlay.IsHost==1)
{
    Debug.Log("Starting as Host...");
    NetworkManager.Singleton.StartHost();
}
else if(MainPlay.IsHost==2)
{
    Debug.Log("Starting as Client...");
    NetworkManager.Singleton.StartClient();
}
```

NetworkManager

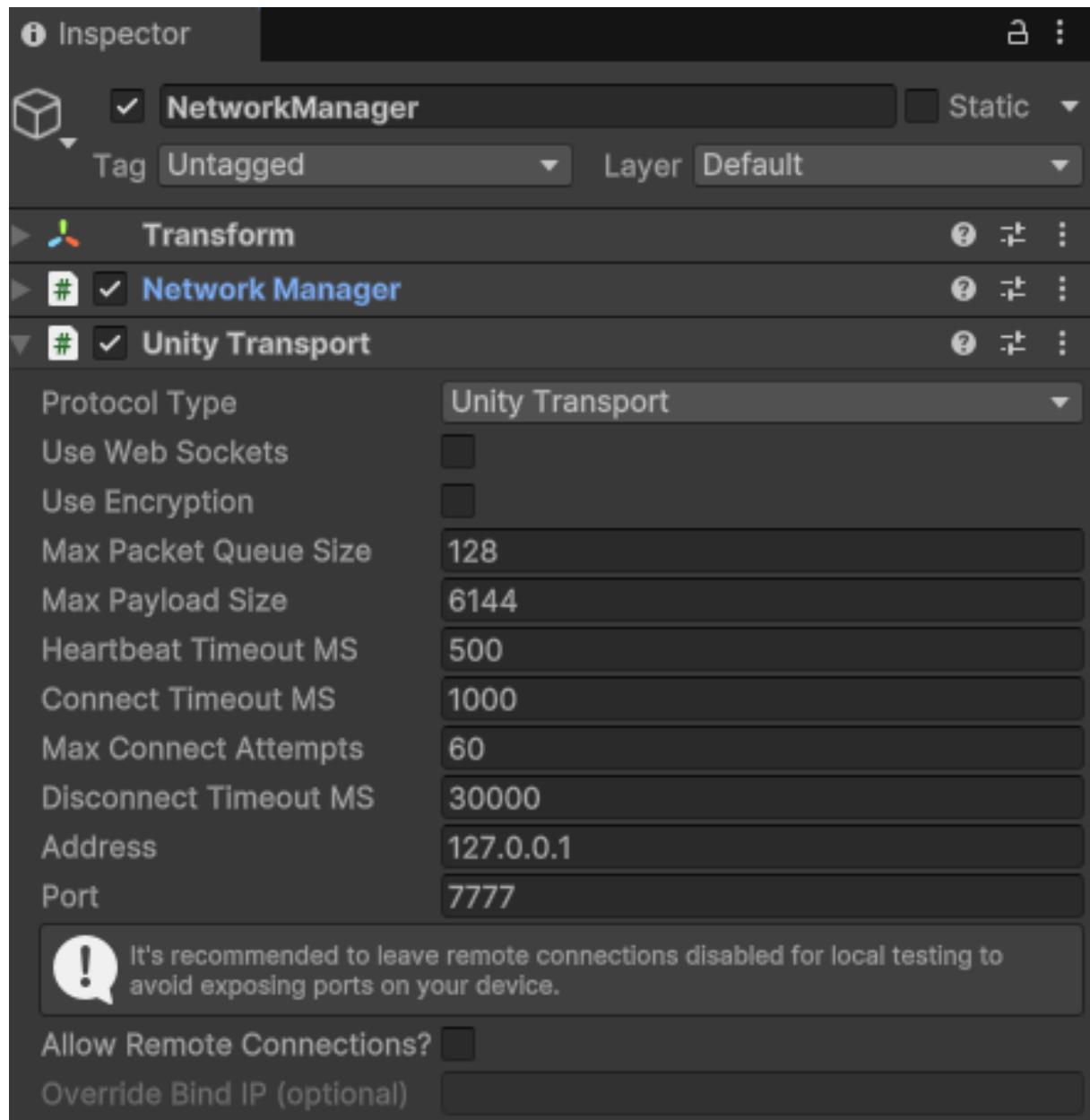
Pour le multijoueur, on utilise **NGO** (*Netcode for GameObjects*), la solution de Unity pour le multijoueur. C'est un paquet disponible dans le *Package Manager* de Unity. Ensuite, il suffit de créer un objet vide (*Empty GameObject*) et d'y attacher le composant **NetworkManager**.





Transport

Le **transport** gère la passerelle de communication, l'adresse IP, le port et le *netcode* (les fonctions réseau). On utilise celui de Unity, mais on pourrait par exemple utiliser celui de Steam si notre jeu est publié sur Steam (ce qui éviterait d'avoir à ouvrir des ports manuellement). Pour l'instant, on utilise les paramètres par défaut de Unity avec le port 7777 en mode `localhost`.



La console

Pour faciliter le débogage, notamment en multijoueur, nous avons ajouté une console permettant d'exécuter des commandes. Elle s'active avec la touche ².





Nous avons donc la classe DebugCommandBase, qui définit les attributs d'une commande :

```
public class DebugCommandBase
{
    2 references
    private string _commandId;
    2 references
    private string _commandDescription;
    2 references
    private string _commandFormat;

    0 references
    public string commandId {get {return _commandId;}}
    0 references
    public string commandDescription {get {return _commandDescription;}}
    0 references
    public string commandFormat {get {return _commandFormat;}}

    1 reference
    public DebugCommandBase(string id, string description, string format){
        this._commandId=id;
        this._commandDescription=description;
        this._commandFormat=format;
    }
}
```

Elle contient un nom, une description et le format de la commande.

Puis une deuxième classe, DebugCommand, qui hérite de DebugCommandBase :

```

1 reference
public class DebugCommand : DebugCommandBase{
    2 references
    private Action command;
    0 references
    public DebugCommand(string id, string desc, string form, Action command) : base (id, desc, form){
        this.command=command;
    }
    0 references
    public void Invoke(){
        command.Invoke();
    }
}

```

On définit l'ID, la description et le format de la commande, mais on ajoute aussi la variable `commande` de type `Action`. C'est elle qui stocke la fonction que la méthode `Invoke` va appeler.

Après avoir défini le type `DebugCommand`, nous créons la console elle-même. Tout d'abord, nous définissons nos commandes :

```

0 references
public class Console : MonoBehaviour
{
    4 references
    bool showConsole = false;
    4 references
    string input = "";

    2 references
    public static DebugCommand HOST;
    2 references
    public static DebugCommand JOIN;
    // public static DebugCommand SET;

    4 references
    public List<object> commandList;

    0 references
    private void Awake()
    {

        HOST = new DebugCommand("host", "hosts a server", "host", () =>
        {
            if (NetworkManager.Singleton.IsServer || NetworkManager.Singleton.IsClient)
                return;

            Debug.Log("Starting as Host...");
            NetworkManager.Singleton.StartHost();
        });

        JOIN = new DebugCommand("join", "joins a server", "join", () =>
        {
            if (NetworkManager.Singleton.IsServer || NetworkManager.Singleton.IsClient)
                return;

            Debug.Log("Starting as Client...");
            NetworkManager.Singleton.StartClient();
        });
    }
}

```

```
    commandList = new List<object>
    {
        HOST,
        JOIN
        // SET
    };
}
```

Puis, on surveille l'appui sur la touche d'ouverture de la console et on l'affiche :

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.BackQuote))
    {
        showConsole = !showConsole;
    }
}
```

OnGUI est une fonction de base de Unity. Elle est appelée plusieurs fois par frame et, ici, elle est utilisée pour dessiner la console à l'écran lorsqu'on appuie sur la touche ².

```

private void OnGUI()
{
    if (!showConsole)
        return;

    float y = 0f;
    GUI.Box(new Rect(0, y, Screen.width, 30), "");

    GUI.SetNextControlName("ConsoleInput");
    input = GUI.TextField(new Rect(10f, y + 5f, Screen.width - 20f, 20f), input);

    if (Event.current.type == EventType.Repaint && GUI.GetNameOfFocusedControl() != "ConsoleInput")
    {
        GUI.FocusControl("ConsoleInput");
    }

    Event e = Event.current;
    if (e.type == EventType.KeyDown &&
        (e.keyCode == KeyCode.Return || e.keyCode == KeyCode.KeypadEnter || e.character == '\n'))
    {
        HandleInput();

        input = "";
        showConsole = false;

        GUI.FocusControl(null);

        Input.ResetInputAxes();

        e.Use();
    }
}

```

On détecte si la touche Entrée (Return) est pressée, puis on valide la commande et on appelle la fonction HandleInput. HandleInput va parcourir la liste des commandes et vérifier si celle tapée par l'utilisateur est valide. Si c'est le cas, elle est exécutée.

```
1 reference
private void HandleInput()
{
    for (int i = 0; i < commandList.Count; i++)
    {
        DebugCommandBase commandBase = commandList[i] as DebugCommandBase;
        if (commandBase != null && input.Contains(commandBase.commandId))
        {
            DebugCommand command = commandList[i] as DebugCommand;
            if (command != null)
            {
                command.Invoke();
            }
        }
    }
}
```

2.1.5 Mouvement & Skills

Mouvements

Le code de mouvement en réseau est quasiment identique au code de mouvement en solo. Les enjeux principaux sont : contrôler son propre personnage et non celui des autres, et synchroniser la position du joueur. Au lieu de faire hériter notre classe de `MonoBehaviour` (le comportement par défaut dans Unity), on la fait hériter de `NetworkBehaviour`, ce qui permet de gérer le *networking* du joueur.

```

using System;
using UnityEngine;
using Unity.Netcode;
0 references
public class PlayerMovement : NetworkBehaviour
{
    1 reference
    public float speed = 10f;
    2 references
    private Rigidbody2D rb;
    2 references
    private Vector2 movement;
    4 references
    private float moveX = 0f;
    4 references
    private float moveY = 0f;

    0 references
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    0 references
    void FixedUpdate()
    {
        if (!IsOwner) return;
        moveX = 0f;
        moveY = 0f;

        if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("left", "A"))))
            moveX = -1f; // Gauche
        if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("right", "D"))))
            moveX = 1f; // Droite
        if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("up", "W"))))
            moveY = 1f; // Haut
        if (Input.GetKey((KeyCode)System.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("down", "S"))))
            moveY = -1f; // Bas

        movement = new Vector2(moveX, moveY).normalized;
        rb.linearVelocity = movement * speed;
    }
}

```

La classe Ability

La classe **Ability** est la classe de base de toutes les capacités. Elle gère les *cooldowns* et définit la structure que chaque capacité doit suivre.

```

public abstract class Ability : MonoBehaviour
{
    public string AbilityName;
    public float cooldown = 1f;

    protected bool isOnCooldown = false;

    2 references
    public abstract void Activate();

    // Gère le cooldown automatiquement
    1 reference
    protected IEnumerator CooldownRoutine()
    {
        isOnCooldown = true;
        yield return new WaitForSeconds(cooldown);
        isOnCooldown = false;
    }
}

```

AbilityManager

AbilityManager est responsable de la gestion des capacités du joueur.

```

public class AbilityManager : MonoBehaviour
{
    private List<Ability> Abilities = new List<Ability>();
    private Ability activeAbility;
    private int currentAbilityIndex = 0;

    Unity Message | 0 references
    void Start()
    {
        // Trouve toutes les capacités attachées au même GameObject
        Abilities.AddRange(GetComponents<Ability>());
        if (Abilities.Count > 0)
        {
            SetActiveAbility(0); // Sélectionne la 1ere capacité au début
        }
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Q)) // Changer de capacité
        {
            CycleAbility();
        }

        if (Input.GetKeyDown(KeyCodeSystem.Enum.Parse(typeof(KeyCode), PlayerPrefs.GetString("interact", "F"))))
        {
            Debug.Log(Abilities[currentAbilityIndex].AbilityName);
            activeAbility?.Activate();
        }
    }
}

```

```

1 reference
void CycleAbility()
{
    currentAbilityIndex = (currentAbilityIndex + 1) % Abilities.Count;
    SetActiveAbility(currentAbilityIndex);
}

2 references
void SetActiveAbility(int index)
{
    activeAbility = Abilities[index];
}

```

DashAbility

Une capacité de type **Dash** qui permet au joueur de se déplacer rapidement dans une direction spécifique.

```

void Start()
{
    rb = GetComponent<Rigidbody2D>();
    AbilityName = "Dash";
    cooldown = 0.2f;
}

2 references
public override void Activate()
{
    if (!isOnCooldown && !isDashing)
    {
        StartCoroutine(DashRoutine());
        StartCoroutine(CooldownRoutine());
    }
}

```

```

private IEnumerator DashRoutine()
{
    isDashing = true;

    Vector2 dashDirection = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical")).normalized;
    if (dashDirection != Vector2.zero)
        lastMoveDirection = dashDirection;
    rb.linearVelocity = lastMoveDirection * dashSpeed;
    PlayerMovement movement = GetComponent<PlayerMovement>();
    if (movement != null)
        movement.enabled = false;

    yield return new WaitForSeconds(dashDuration);

    if (movement != null)
        movement.enabled = true;

    rb.linearVelocity = Vector2.zero;
    isDashing = false;
}

```

2.2 Futur

Il reste encore beaucoup à faire. Même si nous avons établi les bases, nous devons encore implémenter l'intégralité de l'IA des monstres, les différentes mécaniques comme les armes et la magie, ainsi que les trois phases de notre jeu :

- la phase d'équipement,
- la phase de défense,
- la phase d'attaque.

Notre jeu reposant entièrement sur la coopération en équipe face à une horde de monstres, nous souhaitons offrir une certaine diversité dans ces derniers. Ainsi, il y aura plusieurs types de monstres, plus ou moins puissants, chacun doté d'une IA et de mécaniques propres. Certains privilégieront le corps à corps, tandis que d'autres attaqueront à distance ou utiliseront des compétences spéciales conçues pour compliquer la tâche des joueurs.

En parallèle, nous devons également concevoir tout le système économique, incluant l'achat d'armes et d'armures indispensables pour survivre.

Chapitre 3

Conclusion

Cette deuxième phase de développement a permis à **Menacing Duck Studios** de poursuivre sur une base solide qu'il ne faut pas négliger. Nous allons ainsi pouvoir introduire les neuf personnages restants avec leurs compétences propres, ainsi que les ennemis.

Il nous reste encore quelques ajustements à faire, mais dans l'ensemble, nous pouvons être satisfaits de notre avancée. Cependant, nous sommes conscients qu'il va falloir être bien plus efficaces dans les prochains temps.

Nous remercions chacun des membres de l'équipe pour leur implication et leur travail acharné.