TP4 : Abstraction - Système de Véhicules

Objectif

Comprendre et appliquer le concept d'abstraction en créant un système de véhicules pour un jeu de course.

Problème

Vous développez un jeu de course avec différents types de véhicules (voiture, moto, avion, bateau). Chaque véhicule a des caractéristiques communes comme la vitesse, l'accélération et le freinage, mais leur comportement et leur mécanisme de mouvement diffèrent selon le type.

Actuellement, le code est structuré de manière inefficace, avec une logique spécifique à chaque véhicule dispersée dans différentes parties du code et beaucoup de duplication.

Code Problématique

Examinons le fichier `VehicleController.cs` actuel :

```
// VehicleController.cs - Problème
using UnityEngine;

public class VehicleController : MonoBehaviour
{
    public enum VehicleType { Car, Motorcycle, Airplane, Boat }

    public VehicleType vehicleType;
    public float speed;
    public float maxSpeed;
    public float acceleration;
    public float handling;
    public float brakeForce;

    // Variables spécifiques aux véhicules
    public float carTraction;
    public float motorcycleLeanAngle;
    public float airplaneLift;
    public float boatBuoyancy;

    void Update()
    {
        float moveInput = Input.GetAxis("Vertical");
        float turnInput = Input.GetAxis("Horizontal");

        // Gérer l'accélération et le freinage
        if (moveInput > 0)
        {
            // Accélération
            if (vehicleType == VehicleType.Car)
            {
```

```
                speed += acceleration * moveInput * Time.deltaTime;
                // Logique spécifique à la voiture
                ApplyCarTraction();
            }
            else if (vehicleType == VehicleType.Motorcycle)
            {
                speed += acceleration * 1.2f * moveInput * Time.deltaTime; // Les
motos accélèrent plus vite
                // Logique spécifique à la moto
                ApplyMotorcycleLean(turnInput);
            }
            else if (vehicleType == VehicleType.Airplane)
            {
                speed += acceleration * 0.8f * moveInput * Time.deltaTime;
                // Logique spécifique à l'avion
                ApplyAirplaneLift();
            }
            else if (vehicleType == VehicleType.Boat)
            {
                speed += acceleration * 0.7f * moveInput * Time.deltaTime;
                // Logique spécifique au bateau
                ApplyBoatBuoyancy();
            }
        }
        else if (moveInput < 0)
        {
            // Freinage
            if (vehicleType == VehicleType.Car)
            {
                speed -= brakeForce * Mathf.Abs(moveInput) * Time.deltaTime;
            }
            else if (vehicleType == VehicleType.Motorcycle)
            {
                speed -= brakeForce * 0.8f * Mathf.Abs(moveInput) *
Time.deltaTime;
            }
            else if (vehicleType == VehicleType.Airplane)
            {
                speed -= brakeForce * 0.4f * Mathf.Abs(moveInput) *
Time.deltaTime;
            }
            else if (vehicleType == VehicleType.Boat)
            {
                speed -= brakeForce * 0.6f * Mathf.Abs(moveInput) *
Time.deltaTime;
            }
        }

        // Limiter la vitesse maximale
        speed = Mathf.Clamp(speed, 0, maxSpeed);

        // Gérer la direction
        if (vehicleType == VehicleType.Car)
        {
```

```
            transform.Rotate(0, turnInput * handling * speed * 0.1f *
Time.deltaTime, 0);
        }
        else if (vehicleType == VehicleType.Motorcycle)
        {
            transform.Rotate(0, turnInput * handling * speed * 0.15f *
Time.deltaTime, 0);
        }
        else if (vehicleType == VehicleType.Airplane)
        {
            transform.Rotate(turnInput * handling * 0.5f * Time.deltaTime,
                             moveInput * handling * 0.3f * Time.deltaTime,
                             -turnInput * handling * Time.deltaTime);
        }
        else if (vehicleType == VehicleType.Boat)
        {
            transform.Rotate(0, turnInput * handling * speed * 0.05f *
Time.deltaTime, 0);
        }

        // Déplacement
        transform.Translate(Vector3.forward * speed * Time.deltaTime);
    }

    void ApplyCarTraction()
    {
        // Simuler la traction d'une voiture
        if (Physics.Raycast(transform.position, -transform.up, out RaycastHit hit,
1.0f))
        {
            float surfaceFactor = 1.0f;
            if (hit.collider.CompareTag("Dirt")) surfaceFactor = 0.7f;
            if (hit.collider.CompareTag("Ice")) surfaceFactor = 0.3f;
            speed *= (1.0f - (1.0f - carTraction) * (1.0f - surfaceFactor));
        }
    }

    void ApplyMotorcycleLean(float turnInput)
    {
        // Simuler l'inclinaison d'une moto dans les virages
        float targetLean = -turnInput * motorcycleLeanAngle;
        Vector3 currentRotation = transform.localEulerAngles;
        currentRotation.z = Mathf.LerpAngle(currentRotation.z, targetLean,
Time.deltaTime * 2.0f);
        transform.localEulerAngles = currentRotation;
    }

    void ApplyAirplaneLift()
    {
        // Simuler la portance d'un avion
        if (speed > maxSpeed * 0.3f)
        {
            float liftForce = airplaneLift * (speed / maxSpeed);
            transform.Translate(Vector3.up * liftForce * Time.deltaTime,
```

```
Space.World);
        }
    }

    void ApplyBoatBuoyancy()
    {
        // Simuler la flottabilité d'un bateau
        if (Physics.Raycast(transform.position, -transform.up, out RaycastHit hit,
2.0f))
        {
            if (hit.collider.CompareTag("Water"))
            {
                float desiredHeight = hit.point.y + boatBuoyancy;
                Vector3 pos = transform.position;
                pos.y = Mathf.Lerp(pos.y, desiredHeight, Time.deltaTime * 2.0f);
                transform.position = pos;
            }
        }
    }
}
```

Problèmes avec ce code

1. Ce code enfreint plusieurs principes de conception orientée objet :
    - Il mélange des fonctionnalités pour différents types de véhicules dans une seule classe
    - Il utilise des conditions if/else basées sur le type pour déterminer le comportement
    - Il contient de nombreuses variables qui ne sont pertinentes que pour certains types de véhicules
    - Il est difficile d'ajouter un nouveau type de véhicule sans modifier directement cette classe
2. L'abstraction est absente :
    - Il n'y a pas de séparation claire entre les caractéristiques communes des véhicules et leurs comportements spécifiques
    - La logique de chaque type de véhicule est dispersée à travers la même classe

Exercice

Refactorisez ce code en utilisant l'abstraction :

1. Créez une classe abstraite `Vehicle` qui contient les propriétés et comportements communs
2. Définissez des méthodes abstraites pour les comportements qui varient entre les types de véhicules
3. Créez des classes concrètes pour chaque type de véhicule qui héritent de la classe abstraite
4. Implémentez les méthodes abstraites dans chaque classe concrète selon les spécificités du véhicule

Avantage de l'abstraction

- Organisation plus claire du code
- Élimination des blocs conditionnels basés sur le type
- Facilité d'extension (ajout de nouveaux types de véhicules)
- Meilleure encapsulation des comportements spécifiques

- Code plus maintenable et testable

Conseil

Pensez à quelles fonctionnalités devraient être communes à tous les véhicules (dans la classe abstraite) et lesquelles devraient être spécifiques à chaque type (méthodes abstraites à implémenter).