# Traffic Jam Minigame

Greetings,

It has been a pleasure for me to partake in this test, and I am eager to showcase my best skills and abilities. Throughout the evaluation, I have strived to deliver nothing but my utmost effort and precision, aiming to minimize, if not eliminate, any errors in my work.

Thank you for the opportunity, and I look forward to moving on to the next step in the recruitment process.

Best regards,

## INTRODUCTION

The resolution of the test involved a clear understanding of the requirements and needs of the project, as well as the identification of optimal solutions.

## IMPLEMENTATION

Throughout the development process, I chose to implement the SOLID principles to avoid code issues such as Rigidity, Fragility, Immobility, Viscosity, and Needless Complexity. Simultaneously, I decided to incorporate as many Object-Oriented Programming (OOP) principles and design patterns as possible.

## MECHANICS:

## PLAYER CAR

The player car system consists of four classes:

1. **Moveable Class:**
    - **Responsibility:** Moves the object forward using physics.
    - **Interface Implemented:** IMoveable.
    - **Attributes:**
        - `stopCar` : Indicates whether the car movement should be stopped.
        - `carSpeed` : Represents the speed of the car.
    - **Methods:**
        - `MoveForward` : Moves the object forward using physics.
        - `MoveForwardToPoint` : Moves the object forward towards a specified point if the distance is greater than 1.5f.
2. **Rotatable Class:**
    - **Responsibility:** Controls the rotation of the object based on a specified position.
    - **Interface Implemented:** IRotatable.
    - **Attributes:**
        - `rotationSpeed` : Determines the rotation speed of the object.
        - `targetPoint` : Represents the target point towards which the object should rotate.
    - **Methods:**
        - `RotateTowards` : Rotates the object towards a specified position in world space. The rotation occurs only if the distance between the current object position and the mouse raycast point is greater than 0.9f.
3. **InputHandler Class:**
    - **Responsibility:** Handles user input for movement and rotation, check in FixedUpdate() for inputs.
    - **Attributes:**
        - `CanMove` : Indicates whether the object should move.
    - **Events:**
        - `OnMove` : Triggered when the object is supposed to move.
        - `OnRotate` : Triggered when the object is supposed to rotate.
    - **Methods:**
        - `FixedUpdate` : Checks for user input and mouse position, triggers move and rotate events accordingly.
4. **DetectBlueTrafficCollision Class:**
    - **Responsibility:** Detects hits on Blue car game objects.
    - **Attributes:**
        - `hitBill` : The amount of money subtracted from `gameManager.playerMoneyCount` when a collision occurs.
        - `hitEffect` : ParticleSystem for the hit NPC car effect.
    - **Methods:**
        - `OnCollisionEnter` : Handles collision events, deducts money, updates UI, plays hit effect, and deactivates the collided car.
        - `PlayHitEffect` : Instantiates and plays the hit effect at a given position.

These classes collectively contribute to the functionality of the player car system, enabling movement, rotation, user input handling, and collision detection with Blue cars. The logic behind the Player Car has been divided into multiple classes to avoid violating the Single Responsibility Principle (SRP). Interfaces have been implemented to adhere to the Open-Closed Principle, and the `ICar` interface, which includes the `IRotable` and `IMoveable` interfaces, implements the Interface Segregation Principle (ISP).

**Explanation:**

The Single Responsibility Principle (SRP) suggests that a class should have only one reason to change. By breaking down the logic of the Player Car into separate classes such as `Moveable`, `Rotatable`, `InputHandler`, and `DetectBlueTrafficCollision`, each class focuses on a specific responsibility, ensuring a more modular and maintainable codebase.

The Open-Closed Principle (OCP) encourages the design of classes to be open for extension but closed for modification. Implementing interfaces ( `IRotable` and `IMoveable` ) allows for the extension of behavior without modifying existing code. This facilitates the addition of new features or behaviors to the Player Car system without altering the existing classes, promoting a more scalable and adaptable design.

The Interface Segregation Principle (ISP) suggests that a class should not be forced to implement interfaces it does not use. The `ICar` interface, which combines `IRotable` and `IMoveable` , adheres to ISP by ensuring that classes that implement `ICar` only need to provide the functionalities relevant to them. This avoids unnecessary dependencies and ensures that classes are not burdened with methods they do not require.

**Benefits of the Design:**

- **SRP Adherence:** Distinct classes for each logic uphold the Single Responsibility Principle, resulting in a more maintainable and understandable codebase.
- **ISP Consideration:** By creating specific interfaces for distinct behaviors, the design avoids ISP violations, ensuring that implementing classes are only obligated to support the methods relevant to their functionality.
- **Open-Closed Principle:** The use of interfaces and the consideration of the Open-Closed Principle facilitate a design that is open to extension but closed to modification, allowing for the incorporation of new features without altering existing code.

**Overall:** The design of the Player Car system reflects a commitment to SOLID principles, specifically SRP, OCP, and ISP, contributing to a more modular, extensible, and maintainable codebase.

# Blue Traffic Cars/NPCs

**Blue Traffic Cars/NPCs System Design Overview:**

The logic for Blue Traffic Cars (NPCs) has been meticulously structured to adhere to fundamental principles of software design, promoting modularity, extensibility, and readability.

**Single Responsibility Principle (SRP):**

- The SRP is honored by dividing the Blue Traffic Cars' logic into distinct classes, each with a specific responsibility.

1. **Rotable Class:**
    - **Responsibility:** Manages the rotation behavior of non-player controlled cars.
    - **Implementation:** Implements the `IRotatable` interface, allowing for the extension of rotation behavior without altering existing code.
2. **Moveable Class:**
    - **Responsibility:** Handles the forward movement of non-player controlled cars using physics.
    - **Implementation:** Implements the `IMoveable` interface, fostering code reuse and maintaining a consistent interface for movement.
3. **TrafficDetector Class:**
    - **Responsibility:** Detects obstacles in the environment using various sensors.
    - **Features:**
        - Utilizes line and sphere sensors for obstacle detection.
        - Provides settings for sensor ranges and minimum distances.
        - Visualizes sensor coverage in the editor for debugging purposes.
        - Determines if an obstacle is too close based on sensor readings.
        - Adaptable to different scenarios with configurable parameters.
4. **BlueTrafficCar Class:**
    - **Responsibility:** Represents a non-player controlled car with events for movement and rotation.
    - **Features:**
        - Subscribes to movement and rotation events triggered by the `IMoveable` and `IRotatable` interfaces.
        - Utilizes a `TrafficDetector` to determine if the car should stop based on proximity to obstacles.
        - Adaptable to different scenarios with a flag ( `mainStreetCar` ) indicating whether the car is on the main street.
        - Includes a target point for the car to navigate towards ( `dezactivatorPosition` ).

**Advantages of the Design:**

- **Modularity:** Each class has a clear responsibility, contributing to a modular and maintainable codebase.
- **Extensibility:** Interfaces and event-based communication allow for easy extension and addition of new features.
- **Readability:** The code structure and use of interfaces enhance code readability, making it easier to understand and maintain.

**Overall:** This system design for Blue Traffic Cars aligns with SOLID principles and best practices in object-oriented design. The thoughtful application of these principles contributes to a flexible, scalable, and maintainable codebase for handling non-player controlled cars in the game environment.

# Traffic Spawner

**Traffic Spawn System Design Overview:**

The Traffic Spawn system implements the Object Pool pattern to efficiently spawn traffic, enhancing performance by avoiding costly Instantiate and Destroy operations. The system comprises three classes: `ObjectPool` , `TrafficSpawner` , and `TrafficManager` , `BlueTrafficDezactivator` .

**Object Pool Pattern Explanation:**

The Object Pool pattern optimizes resource usage by reusing pre-created objects instead of creating and destroying them dynamically. This leads to improved performance and responsiveness.

**Key Components of the Object Pool Pattern:**

1. **Object Initialization:**
    - During initialization, a predefined number of traffic objects are created and configured, forming the object pool.
2. **Pooling Mechanism:**

    - Rather than destroying traffic objects, they are returned to the object pool when no longer needed. This promotes efficient reuse and minimizes resource overhead.

3. **Performance Benefits:**
    - Object Pooling avoids the performance costs associated with dynamic object creation and destruction, resulting in a more responsive traffic spawning mechanism.

**Advantages of Object Pooling in Traffic Spawn System:**

1. **Reduced Resource Overhead:**
    - Minimizes strain on system resources by avoiding frequent instantiation and destruction of traffic objects.
2. **Optimized Memory Usage:**
    - Keeps a fixed number of objects in the pool, optimizing memory usage and preventing unnecessary allocation and deallocation.
3. **Smoother Performance:**
    - Ensures a smoother traffic spawning process by reusing existing objects, reducing the likelihood of performance spikes.
4. **Scalability:**
    - Provides a scalable solution that adapts to varying levels of traffic, accommodating changing demands without sacrificing performance.

**TrafficSpawner Class:**

- **Responsibility:**
    - Spawns cars at regular intervals using an object pool.
- **Features:**
    - Utilizes a coroutine to spawn cars at specified intervals.
    - Checks if spawning is allowed based on the TrafficManager's maximum active NPC limit.
    - Sets target points and speed for spawned cars.

**ObjectPool Class:**

- **Responsibility:**
    - Manages an object pool for efficient instantiation and deactivation of objects.
- **Features:**
    - Initializes the object pool during start by creating and deactivating a predefined number of objects.
    - Retrieves an inactive object from the pool or creates a new one if none are available.

**TrafficManager Class:**

- **Responsibility:**
    - Manages the spawning of non-player controlled characters (NPCs) in the game.
- **Features:**
    - Utilizes the Object Pool pattern to check if it is allowed to spawn a new NPC based on the maximum active NPC limit set by the GameManager.
      **BlueTrafficDezactivator Class:**
- **Responsibility:**
    - Deactivates non-player controlled characters (NPCs) upon entering the trigger zone, contributing to efficient resource management.
- **Features:**
    - Implements a time delay before deactivating the NPC after entering the trigger zone.
    - Checks if the entering collider is an NPC and initiates the deactivation process.
    - Uses a coroutine to delay the deactivation process, allowing for better control and responsiveness.

**Overall:** The `BlueTrafficDezactivator` class complements the Traffic Spawn system by providing a mechanism for controlled NPC deactivation, aligning with best practices for efficient resource usage and contributing to a well-organized and responsive game environment.

**Overall:** The use of the Object Pool pattern in the Traffic Spawn system contributes to a more efficient and responsive traffic spawning mechanism, aligning with best practices for performance optimization.

## Money System Design Overview:

The Money System consists of three classes: `MoneySpawner`, `MoneyValue`, and `MoneyRotable`. The system is designed to handle the spawning, collection, and rotation of money objects within the game.

**MoneySpawner Class:**

- **Responsibility:**
    - Handles the spawning of money objects within a specified radius.
- **Features:**
    - Uses an object pool pattern to efficiently manage money objects.
    - Spawns money objects at regular intervals based on a timer.
    - Checks if the maximum number of active money objects is not exceeded.
    - Visualizes the spawn radius in the Scene view.
    - Utilizes a coroutine for spawning money.

**MoneyValue Class:**

- **Responsibility:**
    - Represents a money pickup in the game.
- **Features:**

- Stores the value of the money pickup.
- Plays a particle system effect when the money is collected.
- Updates the player's total money count and UI text when collected.
- Deactivates the money pickup object after a specified time.
- Uses tags and triggers to detect collisions with the player.

**MoneyRotable Class:**

- **Responsibility:**
  - Rotates the money object around its up axis.
- **Features:**
  - Rotates the money object at a specified rotation speed.
  - Updates rotation every frame.

**Key Concepts and Patterns Used:**

- **Object Pool Pattern:**
  - Efficiently manages the instantiation and deactivation of money objects to avoid the overhead of constant instantiation and destruction.

**Overall:** The Money System is well-structured, providing an engaging and visually appealing experience for players collecting money objects. The use of the Object Pool pattern contributes to the system's efficiency and performance.

# Utility System Overview:

The Utility System consists of two utility classes, `RaycastUtility` and `StringUtility`, providing common functionality for raycasting operations and string formatting, respectively.

## RaycastUtility Class:

- **Responsibility:**
  - Handles common raycasting operations, particularly obtaining the world space position where the ray hits based on the cursor's position on the screen.
- **Key Methods:**
  1. **GetMouseRaycastPoint():**
     - Obtains the raycast hit point from the position of the cursor on the screen.
     - Uses the main camera to cast a ray from the screen position and returns the world space position where the ray hits.

## StringUtility Class:

- **Responsibility:**
  - Provides utility functions for formatting string values, specifically for displaying money values in a user-friendly manner.
- **Key Methods:**
  1. **FormatMoney(int moneyValue):**
     - Formats the money value for display.
     - Converts values greater than or equal to 1000 to "k" format (e.g., 1000 becomes "1k").
     - Returns a string representation of the formatted money value.

**Advantages of the Utility System Design:**

1. **Reusability:**
   - Both utility classes provide reusable functions that can be used across different parts of the game without duplicating code.
2. **Readability:**
   - The utility classes encapsulate specific functionalities, making the code more readable and modular.
3. **Consistent Formatting:**
   - The `StringUtility` class ensures consistent and user-friendly formatting of money values throughout the game.
4. **Ease of Maintenance:**
   - Centralizing common functionalities in utility classes simplifies maintenance and updates.

**Overall:** The Utility System demonstrates good software design practices by encapsulating related functionality into utility classes. These classes can be easily reused and maintained, contributing to a more modular and readable codebase.

# SceneTimer Class Overview:

The `SceneTimer` class is responsible for managing and displaying the countdown timer for a game scene. It is designed to update the timer each frame, initialize and start the timer, and invoke events when the timer reaches zero. The timer's initial time is retrieved from the `GameManager` instance, and it utilizes a `TextMeshProUGUI` component to display the remaining time.

## Class Structure:

1. **Fields:**

- `private float currentTime` : Stores the current time remaining on the timer.
- `public TextMeshProUGUI timeText` : References the TextMeshProUGUI component used to display the time remaining.
- `public UnityEvent invokeAfterTime` : Unity Event invoked after the timer reaches zero.

2. **Initialization (Start Method):**
    - In the `Start` method, the timer is initialized and started using the `StartTimer` method.
3. **Update Method:**
    - In the `Update` method, the timer is updated each frame using the `UpdateTimer` and `UpdateTimeText` methods.
4. **StartTimer Method:**
    - The `StartTimer` method initializes the timer with the initial time obtained from the `GameManager`.
5. **UpdateTimer Method:**
    - The `UpdateTimer` method decrements the timer each frame using `Time.deltaTime`.
    - When the timer reaches zero, it displays the final score, invokes events, and resets the timer.
6. **UpdateTimeText Method:**
    - The `UpdateTimeText` method updates the `TextMeshProUGUI` component with the current seconds value.

### Unity Events and Final Score Display:

- When the timer reaches zero, the `UpdateTimer` method:
    - Retrieves the `GameManager` instance.
    - Displays the final score in the `finalScoreText`.
    - Invokes events using `invokeAfterTime.Invoke()`.

### Advantages:

1. **Modularity:**
    - The class is focused on a specific responsibility – managing and displaying the countdown timer.
2. **Unity Events:**
    - Utilizes Unity Events for extensibility, allowing additional actions to be triggered when the timer reaches zero.
3. **Readability:**
    - Clear method names and comments enhance code readability.
4. **Initialization Handling:**
    - The timer is properly initialized in the `Start` method.

**Overall:** The `SceneTimer` class effectively manages the countdown timer, updating it each frame, and invoking events for load MainMenu when the timer reaches zero. Its modular structure and use of Unity Events contribute to a clean and organized implementation.

## GameManager Class Overview:

The `GameManager` class serves as a central hub for managing various aspects of the game, including player money, player car, NPCs, UI elements, and level time. It follows the singleton pattern to ensure that there is only one instance of the `GameManager` throughout the game.

### Responsibilities:

1. **Player Money Management:**
    - Tracks the current amount of player money ( `playerMoneyCount` ).
    - Sets the maximum allowed active money objects ( `maxActiveMoneyObjects` ).
2. **Player Car Management:**
    - References the player's car GameObject ( `playerCar` ).
    - Sets the speed of the player's car ( `playerCarSpeed` ).
3. **NPC (Non-Player Character) Management:**
    - Defines the maximum allowed active NPC cars ( `maxActiveNpcCar` ).
    - Sets the speed of NPC cars ( `npcSpeed` ).
4. **UI Element References:**
    - Stores references to TextMeshProUGUI elements for displaying player money count and final score ( `moneyCountText` and `finalScoreText` ).
5. **Level Time Management:**
    - Sets the time limit for the level ( `levelTime` ).
6. **Singleton Implementation:**

- Implements the singleton pattern to ensure a single instance of the `GameManager`.

7. **Initialization:**
   - In the `Awake` method, sets the singleton instance to the current instance.
   - In the `Start` method, assigns the player car speed and sets the time scale to normal.
8. **StopGame Method:**
   - Provides a method (`StopGame`) to pause the game by setting the time scale to 0.

### Advantages:

1. **Centralized Management:**
   - Centralizes key game parameters and elements in a single class for easy access and management.
2. **Singleton Pattern:**
   - Ensures that there is only one instance of the `GameManager` throughout the game.
3. **Readability:**
   - Uses header sections and comments to clearly organize and document different aspects of the game management.
4. **Flexibility:**
   - Allows easy adjustment of various game parameters without modifying multiple parts of the codebase.
5. **Initialization Handling:**
   - Manages initialization tasks, such as assigning the player car speed, in the `Start` method.

**Overall:** The `GameManager` class effectively serves as a central controller for managing game-related parameters and elements. Its design supports readability, flexibility, and centralized control, contributing to a well-organized game architecture.

## Important Notes

Important Notes

The naming of NPCs in BlueCraTraffic was influenced by the fact that, in testing, this term was more commonly used than "NPCs."

For the naming convention, camel case has been employed for both private and public fields.

The mechanics have been implemented as understood; players can move their cars anywhere on the map, while NPCs can only move in the direction of the deactivator.

Additionally, In checking the distance between the player and `RaycastUtility.GetMouseRaycastPoint()`, some magic numbers were used, but I deemed it unnecessary to introduce variables containing these numbers, as they only need adjustment in the code. Of course, this can be modified if a more intricate algorithmic requirement arises.

## Feedback

The test was quite interesting, showcasing the programmer's ability to handle dynamic situations, such as avoiding NPC blocking, memory overloads, or farm drop issues.

Additionally, on the architectural side, it's important to note that the implementation of Object-Oriented Programming (OOP) principles is a crucial aspect that should not be overlooked. This is essential for maintaining a robust codebase from all perspectives.

I completed the test in approximately 3 days, not working on it full-time, but I made an effort to give my best.

Thank you also to the Marmalade Game Studio team for this opportunity. I hope and look forward to seeing you at the next interview.