Realistic Car Controller Pro - API Documentation

Thank you for purchasing and using Realistic Car Controller Pro. This documentation will guide you on how to use the RCCP API for spawning vehicles, managing player control, recording systems, and all runtime vehicle operations.

API Overview

The RCCP API provides a comprehensive set of static methods accessible through the RCCP class, designed for runtime vehicle management, player control, and scene coordination. The API is designed around simplicity - most complex operations can be performed with a single line of code.

Key Features

- **One Line Operations** - Complex vehicle operations simplified to single method calls

- **Automatic Management** - Handles RCCP_SceneManager integration automatically
- **Error Prevention** - Built-in validation and error handling
- **Performance Optimized** - Efficient operations suitable for runtime use
- **Event Integration** - Automatic event triggering for system coordination

Architecture Integration

The RCCP API works seamlessly with:

- **RCCP_SceneManager** - Automatic scene management and vehicle tracking
- **Event System** - Automatic event triggering for component communication
- **Input System** - Integration with player input and control systems
- **Physics System** - Proper physics state management during operations

Demo Scene Reference

Examine the **RCCP_Scene_Blank_API** demo scene to see practical API implementations. This scene demonstrates:

- Vehicle spawning and registration
- Control state management
- Engine state control
- Player vehicle switching
- Recording and replay operations

Note: This documentation covers all current API methods. For the most up-to-date method signatures, always reference the RCCP.cs script directly in the Scripts folder.

Getting Started with the API

Basic Setup Requirements

Before using the RCCP API, ensure your scene has the proper setup:

```
// Verify RCCP_SceneManager exists (created automatically)
if (RCCP_SceneManager.Instance == null) {
    Debug.LogError("RCCP_SceneManager not found in scene!");
    return;
}

// Verify you have vehicle prefabs to work with
if (vehiclePrefab == null) {
    Debug.LogError("Vehicle prefab not assigned!");
    return;
}
```

Essential Using Statements

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
// No additional using statements needed - RCCP API is globally accessible
```

Common Usage Patterns

Immediate Operations

Most RCCP API methods execute immediately:

```
// Spawn vehicle immediately
RCCP_CarController newVehicle = RCCP.SpawnRCC(vehiclePrefab, position, rotation, true, true, true);

// Change control state immediately
RCCP.SetControl(targetVehicle, false);
```

Validation Before Operations

Always validate objects before API operations:

```
// Check if vehicle exists before operations
if (targetVehicle != null) {
    RCCP.SetEngine(targetVehicle, true);
} else {
    Debug.LogWarning("Target vehicle is null!");
}
```

Vehicle Spawning System

Vehicle spawning is the foundation of dynamic vehicle management in RCCP, allowing runtime creation of vehicles with full configuration control.

Primary Spawning Method

SpawnRCC - Complete Vehicle Spawning

```
public static RCCP_CarController SpawnRCC(
    RCCP_CarController vehiclePrefab,
    Vector3 position,
    Quaternion rotation,
    bool registerAsPlayerVehicle,
    bool isControllable,
    bool isEngineRunning
)
```

Parameters:

- **vehiclePrefab** - RCCP_CarController prefab to instantiate

    - *Requirement:* Must be a properly configured RCCP vehicle prefab
    - *Validation:* Automatically checked for required components
- **position** - World position for vehicle spawning
    - *Type:* Vector3 world coordinates
    - *Recommendation:* Ensure position is above ground with clearance
- **rotation** - Initial vehicle rotation
    - *Type:* Quaternion rotation
    - *Common Usage:* Quaternion.identity for default orientation
- **registerAsPlayerVehicle** - Whether to register as player vehicle
    - *Effect:* Automatically registers with RCCP_SceneManager as controllable vehicle
    - *Limitation:* Only one player vehicle can be active at a time
- **isControllable** - Enable player input control
    - *Effect:* Vehicle responds to player input when true
    - *Use Case:* Set false for AI vehicles or cutscene vehicles
- **isEngineRunning** - Start vehicle with engine running
    - *Effect:* Engine starts immediately, ready for movement
    - *Performance:* Engine audio and effects activate immediately

Return Value:

- **RCCP_CarController** - Reference to the spawned vehicle instance
- **Null Check:* Always verify return value is not null before use

Spawning Examples

Basic Player Vehicle Spawning

// Spawn a player vehicle at a spawn point

public RCCP_CarController SpawnPlayerVehicle(Transform spawnPoint) {

```
    RCCP_CarController newVehicle = RCCP.SpawnRCC(
        playerVehiclePrefab,       // Player's selected vehicle
        spawnPoint.position,       // Spawn point position
        spawnPoint.rotation,       // Spawn point rotation
        true,                      // Register as player vehicle
        true,                      // Make controllable
        true                       // Start engine
    );

    if (newVehicle != null) {
        Debug.Log("Player vehicle spawned successfully: " + newVehicle.name);
    } else {
        Debug.LogError("Failed to spawn player vehicle!");
    }

    return newVehicle;
}
```

AI Vehicle Spawning

```
// Spawn an AI vehicle for traffic or opponents
public RCCP_CarController SpawnAIVehicle(Vector3 position, Quaternion rotation) {
    RCCP_CarController aiVehicle = RCCP.SpawnRCC(
        aiVehiclePrefab,           // AI vehicle prefab
        position,                  // AI spawn position
        rotation,                  // AI spawn rotation
        false,                     // Don't register as player vehicle
```

```
        false,                  // Not directly controllable by player
        true                    // Start with engine running
    );

    // Add AI component after spawning
    if (aiVehicle != null) {
        RCCP_AI aiComponent = aiVehicle.GetComponent<RCCP_AI>();
        if (aiComponent != null) {
            aiComponent.enabled = true;
        }
    }

    return aiVehicle;
}
```

Parked Vehicle Spawning

```
// Spawn a parked vehicle (engine off, not controllable)
public RCCP_CarController SpawnParkedVehicle(Vector3 position, Quaternion rotation) {
    return RCCP.SpawnRCC(
        parkedVehiclePrefab,      // Parked vehicle prefab
        position,                 // Parking position
        rotation,                 // Parking rotation
        false,                    // Not a player vehicle
        false,                    // Not controllable
        false                     // Engine off
    );
```

```
}
```

Advanced Spawning Techniques

Conditional Spawning with Validation

```
public RCCP_CarController SafeSpawnVehicle(RCCP_CarController prefab, Vector3 position, Quaternion rotation) {
    // Validate spawn point
    if (IsValidSpawnPoint(position)) {
        return RCCP.SpawnRCC(prefab, position, rotation, true, true, true);
    } else {
        Debug.LogWarning("Invalid spawn point: " + position);
        return null;
    }
}

private bool IsValidSpawnPoint(Vector3 position) {
    // Check for overlapping colliders
    Collider[] overlapping = Physics.OverlapSphere(position, 2f);
    return overlapping.Length == 0;
}
```

Random Spawn Point Selection

```
public RCCP_CarController SpawnAtRandomLocation(RCCP_CarController prefab, Transform[] spawnPoints) {
    if (spawnPoints.Length == 0) {
        Debug.LogError("No spawn points available!");
        return null;
```

```
    }

    Transform randomSpawn = spawnPoints[Random.Range(0, spawnPoints.Length)];
    return RCCP.SpawnRCC(prefab, randomSpawn.position, randomSpawn.rotation, true, true, true);
}
```

Vehicle Registration and Management

Vehicle registration manages which vehicle is considered the "player vehicle" and how vehicles are tracked by the scene management system.

Player Vehicle Registration

RegisterPlayerVehicle - Make Vehicle Player Controllable

```
// Basic registration
public static void RegisterPlayerVehicle(RCCP_CarController vehicle)

// Registration with controllable state
public static void RegisterPlayerVehicle(RCCP_CarController vehicle, bool isControllable)

// Full registration with control and engine state
public static void RegisterPlayerVehicle(RCCP_CarController vehicle, bool isControllable, bool engineState)
```

Parameters:

- **vehicle** - Vehicle to register as player vehicle
    - *Requirement:* Must be a valid RCCP_CarController instance
    - *Effect:* Becomes the active player vehicle in RCCP_SceneManager

- **isControllable** - Whether vehicle responds to player input
    - *Default:* true (when parameter omitted)
    - *Use Case:* Set false for cinematic control or temporary disable
- **engineState** - Whether engine should be running
    - *Default:* true (when parameter omitted)
    - *Effect:* Starts/stops engine immediately

DeRegisterPlayerVehicle - Remove Player Vehicle

public static void DeRegisterPlayerVehicle()

Effects:

- **Removes Player Control** - Current player vehicle loses player control
- **Updates Scene Manager** - RCCP_SceneManager.activePlayerVehicle becomes null
- **UI Updates** - Dashboard and UI elements disconnect from vehicle
- **Camera Behavior** - Camera may switch to free mode or follow last position

Registration Examples

Player Vehicle Switching

```
// Switch between vehicles seamlessly
public void SwitchToVehicle(RCCP_CarController newVehicle) {
    // First deregister current player vehicle
    RCCP.DeRegisterPlayerVehicle();

    // Register new vehicle as player vehicle
    RCCP.RegisterPlayerVehicle(newVehicle, true, true);

    Debug.Log("Switched to vehicle: " + newVehicle.name);
```

```
}
```

Conditional Vehicle Registration

```
// Register vehicle only if it meets conditions
public void RegisterVehicleIfValid(RCCP_CarController vehicle) {
    if (vehicle != null && vehicle.gameObject.activeInHierarchy) {
        RCCP.RegisterPlayerVehicle(vehicle, true, false); // Controllable but engine off
    } else {
        Debug.LogWarning("Cannot register invalid vehicle");
    }
}
```

Temporary Player Vehicle Handoff

```
// Temporarily give control to another vehicle (e.g., cutscenes)
public IEnumerator TemporaryVehicleSwitch(RCCP_CarController tempVehicle, float duration) {
    // Store current player vehicle
    RCCP_CarController originalVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;

    // Switch to temporary vehicle
    RCCP.RegisterPlayerVehicle(tempVehicle, true, true);

    // Wait for duration
    yield return new WaitForSeconds(duration);

    // Switch back to original vehicle
    if (originalVehicle != null) {
        RCCP.RegisterPlayerVehicle(originalVehicle, true, true);
```

```
    }
}
```

Vehicle Control and State Management

Control and state management methods provide fine-grained control over vehicle behavior and player interaction.

Control State Management

SetControl - Enable/Disable Player Control

public static void SetControl(RCCP_CarController vehicle, bool controlState)

Parameters:

- **vehicle** - Target vehicle to modify
- **controlState** - Whether vehicle responds to player input
    - *true:* Vehicle responds to player input
    - *false:* Vehicle ignores player input (useful for AI control, cutscenes)

Use Cases:

- **Cutscene Control** - Disable control during cinematics
- **AI Takeover** - Allow AI to control while player watches
- **Menu Systems** - Disable control when UI menus are open
- **Temporary Disable** - Prevent control during loading or transitions

SetEngine - Engine State Control

public static void SetEngine(RCCP_CarController vehicle, bool engineState)

Parameters:

- **vehicle** - Target vehicle to modify

- **engineState** - Whether engine should be running
    - *true:* Engine starts/continues running
    - *false:* Engine stops

Effects of Engine State:

- **Performance** - Engine off prevents movement
- **Audio** - Engine sounds start/stop accordingly
- **Visual Effects** - Exhaust effects and engine heat effects
- **Fuel Consumption** - Engine off stops fuel consumption

Advanced Control Examples

Smart Control Management

```
public class VehicleControlManager : MonoBehaviour {
  public bool allowPlayerControl = true;
  private RCCP_CarController playerVehicle;

  void Update() {
    playerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;

    if (playerVehicle != null) {
      // Enable control only if allowed and vehicle is active
      bool shouldControl = allowPlayerControl && playerVehicle.gameObject.activeInHierarchy;
      RCCP.SetControl(playerVehicle, shouldControl);
    }
  }
```

```
    public void DisableControlTemporarily(float duration) {
        StartCoroutine(DisableControlCoroutine(duration));
    }

    IEnumerator DisableControlCoroutine(float duration) {
        allowPlayerControl = false;
        yield return new WaitForSeconds(duration);
        allowPlayerControl = true;
    }
}
```

Engine Management System

```
public class EngineController : MonoBehaviour {
    [Header("Engine Settings")]
    public bool autoStartEngine = true;
    public float engineWarmupTime = 2f;

    public void StartEngine(RCCP_CarController vehicle) {
        if (vehicle != null) {
            StartCoroutine(EngineStartupSequence(vehicle));
        }
    }

    IEnumerator EngineStartupSequence(RCCP_CarController vehicle) {
        // Start engine
        RCCP.SetEngine(vehicle, true);
```

```
        // Disable control during warmup
        RCCP.SetControl(vehicle, false);

        Debug.Log("Engine starting... warming up for " + engineWarmupTime + " seconds");

        // Wait for warmup
        yield return new WaitForSeconds(engineWarmupTime);

        // Enable control after warmup
        RCCP.SetControl(vehicle, true);

        Debug.Log("Engine ready!");
    }

    public void StopEngine(RCCP_CarController vehicle) {
        if (vehicle != null) {
            RCCP.SetEngine(vehicle, false);
            RCCP.SetControl(vehicle, false);
            Debug.Log("Engine stopped");
        }
    }
}
```

Transmission Control

SetAutomaticGear - Transmission Mode Control

public static void SetAutomaticGear(RCCP_CarController vehicle, bool isAutomatic)

Parameters:

- **vehicle** - Target vehicle to modify
- **isAutomatic** - Transmission mode
    - *true:* Automatic transmission (gear changes handled automatically)
    - *false:* Manual transmission (player controls gear changes)

Implementation Example:

```
public class TransmissionManager : MonoBehaviour {
  public void SetTransmissionMode(RCCP_CarController vehicle, bool automatic) {
    if (vehicle != null) {
      RCCP.SetAutomaticGear(vehicle, automatic);

      string mode = automatic ? "Automatic" : "Manual";
      Debug.Log($"Transmission set to {mode} for {vehicle.name}");

      // Update UI to reflect transmission mode
      UpdateTransmissionUI(automatic);
    }
  }

  private void UpdateTransmissionUI(bool isAutomatic) {
    // Update transmission indicator UI
```

```
    // Show/hide manual gear shift buttons
  }
}
```

Camera and Scene Management

Camera and scene management methods provide control over player perspective and scene coordination.

Camera Control

ChangeCamera - Cycle Camera Modes

public static void ChangeCamera()

Effects:

- **Cycles Camera Modes** - Switches to next available camera mode
- **Available Modes** - Third Person, First Person, Hood Camera, Wheel Camera
- **Automatic Transition** - Smooth transition between camera modes
- **Player Preference** - Remembers last used camera mode

Camera Mode Examples:

```
public class CameraController : MonoBehaviour {
  [Header("Camera Settings")]
  public KeyCode cameraToggleKey = KeyCode.C;

  void Update() {
    // Toggle camera mode when key is pressed
    if (Input.GetKeyDown(cameraToggleKey)) {
      RCCP.ChangeCamera();
```

```
    }
  }

  // Programmatically cycle through camera modes
  public void CycleCameraMode() {
    RCCP.ChangeCamera();

    // Get current camera mode for UI updates
    if (RCCP_SceneManager.Instance.activePlayerCamera != null) {
      var camera = RCCP_SceneManager.Instance.activePlayerCamera;
      Debug.Log("Camera switched to: " + camera.cameraMode);
    }
  }
}
```

Vehicle Transportation

Transport - Instant Vehicle Positioning

```
// Transport player vehicle
public static void Transport(Vector3 position, Quaternion rotation)

// Transport specific vehicle
public static void Transport(RCCP_CarController vehicle, Vector3 position, Quaternion rotation)
```

Parameters:

- **vehicle** - Target vehicle to transport (uses player vehicle if omitted)
- **position** - Destination world position

- **rotation** - Destination rotation

Effects:

- **Instant Movement** - Vehicle is immediately moved to new position
- **Velocity Reset** - Rigidbody velocity is frozen to prevent momentum
- **Physics Stabilization** - Ensures vehicle settles properly at new position
- **Camera Update** - Camera follows to new position automatically

Transportation Examples:

```
public class VehicleTransporter : MonoBehaviour {
  [Header("Transport Points")]
  public Transform[] checkpoints;
  public Transform garage;
  public Transform startLine;

  // Transport to specific checkpoint
  public void TransportToCheckpoint(int checkpointIndex) {
    if (checkpointIndex >= 0 && checkpointIndex < checkpoints.Length) {
      Transform checkpoint = checkpoints[checkpointIndex];
      RCCP.Transport(checkpoint.position, checkpoint.rotation);
      Debug.Log($"Transported to checkpoint {checkpointIndex}");
    }
  }

  // Transport to garage
  public void TransportToGarage() {
    RCCP.Transport(garage.position, garage.rotation);
```

```
        // Additional garage functionality
        var playerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;
        if (playerVehicle != null) {
            RCCP.SetEngine(playerVehicle, false); // Turn off engine in garage
        }
    }

    // Transport specific vehicle with validation
    public void SafeTransport(RCCP_CarController vehicle, Vector3 position, Quaternion rotation) {
        if (vehicle != null && IsValidTransportPosition(position)) {
            RCCP.Transport(vehicle, position, rotation);
        } else {
            Debug.LogWarning("Invalid transport parameters");
        }
    }

    private bool IsValidTransportPosition(Vector3 position) {
        // Check if position is valid (not inside colliders, etc.)
        return !Physics.CheckSphere(position, 1f);
    }
}
```

Recording and Replay System

The recording and replay system allows capturing and replaying vehicle movements and inputs for various gameplay purposes.

Recording Control

StartStopRecord - Recording Management

public static void StartStopRecord(RCCP_CarController vehicle)

Functionality:

- **Toggle Behavior** - Starts recording if not recording, stops if currently recording
- **Input Capture** - Records all player inputs (steering, throttle, brake, etc.)
- **Physics Data** - Captures vehicle position, rotation, and velocity
- **Performance Optimized** - Efficient data recording suitable for runtime use

StartStopReplay - Replay Management

// Replay last recording

public static void StartStopReplay(RCCP_CarController vehicle)

// Replay specific recording clip

public static void StartStopReplay(RCCP_CarController vehicle, RCCP_Recorder.Recorded recordClip)

Parameters:

- **vehicle** - Target vehicle for replay
- **recordClip** - Specific recording to replay (uses last recording if omitted)

Replay Behavior:

- **Input Override** - Player inputs are overridden with recorded inputs
- **Physics Accuracy** - Vehicle follows recorded physics data precisely
- **Visual Fidelity** - Maintains visual accuracy of original recording

StopRecordReplay - Complete Stop

public static void StopRecordReplay(RCCP_CarController vehicle)

Effects:

- **Stops All Recording** - Halts any active recording
- **Stops All Replay** - Halts any active replay
- **Restores Control** - Returns normal input control to player

Recording System Examples

Basic Recording Controller

```
public class RecordingManager : MonoBehaviour {
    [Header("Recording Settings")]
    public KeyCode recordKey = KeyCode.R;
    public KeyCode playKey = KeyCode.P;
    public KeyCode stopKey = KeyCode.O;

    private RCCP_CarController playerVehicle;

    void Update() {
        playerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;

        if (playerVehicle == null) return;

        // Recording controls
        if (Input.GetKeyDown(recordKey)) {
            RCCP.StartStopRecord(playerVehicle);
```

```
            Debug.Log("Recording toggled");
        }

        if (Input.GetKeyDown(playKey)) {
            RCCP.StartStopReplay(playerVehicle);
            Debug.Log("Replay toggled");
        }

        if (Input.GetKeyDown(stopKey)) {
            RCCP.StopRecordReplay(playerVehicle);
            Debug.Log("Recording/Replay stopped");
        }
    }
}
```

Advanced Recording System

```
public class AdvancedRecordingSystem : MonoBehaviour {
    [Header("Recording Management")]
    public float maxRecordingTime = 300f; // 5 minutes
    public List<RCCP_Recorder.Recorded> savedRecordings = new
List<RCCP_Recorder.Recorded>();

    private bool isRecording = false;
    private bool isReplaying = false;
    private float recordingStartTime;

    public void StartRecording(RCCP_CarController vehicle) {
```

```
    if (!isRecording && vehicle != null) {
        RCCP.StartStopRecord(vehicle);
        isRecording = true;
        recordingStartTime = Time.time;
        Debug.Log("Recording started");

        // Start recording duration check
        StartCoroutine(MonitorRecordingDuration(vehicle));
    }
}

public void StopRecording(RCCP_CarController vehicle) {
    if (isRecording && vehicle != null) {
        RCCP.StartStopRecord(vehicle); // Toggle to stop
        isRecording = false;
        Debug.Log("Recording stopped");

        // Save recording
        SaveCurrentRecording(vehicle);
    }
}

IEnumerator MonitorRecordingDuration(RCCP_CarController vehicle) {
    while (isRecording) {
        if (Time.time - recordingStartTime >= maxRecordingTime) {
            StopRecording(vehicle);
```

```
            Debug.Log("Recording stopped - maximum duration reached");
            break;
        }
        yield return new WaitForSeconds(1f);
    }
  }

  private void SaveCurrentRecording(RCCP_CarController vehicle) {
    var recorder = vehicle.GetComponent<RCCP_Recorder>();
    if (recorder != null && recorder.recorded != null) {
        savedRecordings.Add(recorder.recorded);
        Debug.Log("Recording saved. Total recordings: " + savedRecordings.Count);
    }
  }

  public void PlayRecording(RCCP_CarController vehicle, int recordingIndex) {
    if (recordingIndex >= 0 && recordingIndex < savedRecordings.Count) {
        RCCP.StartStopReplay(vehicle, savedRecordings[recordingIndex]);
        isReplaying = true;
        Debug.Log($"Playing recording {recordingIndex}");
    }
  }
}
```

Behavior and Configuration Management

Behavior and configuration management allows runtime modification of vehicle handling characteristics and global settings.

Behavior Control

SetBehavior - Vehicle Handling Presets

public static void SetBehavior(int behaviorIndex)

Parameters:

- **behaviorIndex** - Index of behavior preset to apply
    - *Range:* 0 to available behavior count - 1
    - *Effect:* Changes global vehicle behavior settings

Behavior Types (Typical Setup):

- **0** - Simulation (Realistic physics)
- **1** - Arcade (Responsive, game-like physics)
- **2** - Fun (Exaggerated physics for entertainment)
- **3** - Semi-Simulation (Balanced realism and fun)
- **4** - Custom (User-defined settings)

SetController - Input Controller Type

public static void SetController(int controllerIndex)

Parameters:

- **controllerIndex** - Type of input controller
    - *0:* Keyboard and Mouse
    - *1:* Mobile Touch Controls
    - *2:* Gamepad/Controller

- *3:* Custom Input System

Behavior Management Examples

Dynamic Behavior Switching

```
public class BehaviorController : MonoBehaviour {
    [Header("Behavior Settings")]
    public string[] behaviorNames = {"Simulation", "Arcade", "Fun", "Semi-Sim"};
    public KeyCode behaviorToggleKey = KeyCode.B;

    private int currentBehavior = 0;

    void Update() {
        if (Input.GetKeyDown(behaviorToggleKey)) {
            CycleBehavior();
        }
    }

    public void CycleBehavior() {
        currentBehavior = (currentBehavior + 1) % behaviorNames.Length;
        RCCP.SetBehavior(currentBehavior);

        Debug.Log($"Behavior changed to: {behaviorNames[currentBehavior]}");

        // Update UI
        UpdateBehaviorUI();
    }
```

```
    public void SetSpecificBehavior(int behaviorIndex) {
        if (behaviorIndex >= 0 && behaviorIndex < behaviorNames.Length) {
            currentBehavior = behaviorIndex;
            RCCP.SetBehavior(currentBehavior);
            Debug.Log($"Behavior set to: {behaviorNames[currentBehavior]}");
        }
    }

    private void UpdateBehaviorUI() {
        // Update UI elements to show current behavior
    }
}
```

Platform-Specific Controller Setup

```
public class PlatformControllerManager : MonoBehaviour {
    void Start() {
        SetControllerForPlatform();
    }

    void SetControllerForPlatform() {
        #if UNITY_STANDALONE
            RCCP.SetController(0); // Keyboard/Mouse for PC
        #elif UNITY_ANDROID || UNITY_IOS
            RCCP.SetController(1); // Touch controls for mobile
        #elif UNITY_GAMEPAD
```

```
        RCCP.SetController(2); // Gamepad for console
    #else
        RCCP.SetController(0); // Default to keyboard
    #endif

    Debug.Log("Controller set for current platform");
  }
}
```

Mobile Controller Management

Mobile controller management provides specific functionality for touch-based input systems on mobile platforms.

Mobile Controller Control

SetMobileController - Mobile Input Type

public static void SetMobileController(RCCP_Settings.MobileController mobileController)

Parameters:

- **mobileController** - Type of mobile input interface
    - **TouchScreen** - Traditional touch buttons
    - **Tilt** - Gyroscope/accelerometer steering
    - **SteeringWheel** - Virtual steering wheel interface
    - **Joystick** - Virtual joystick controls

Mobile Controller Examples

Adaptive Mobile Interface

```
public class MobileControllerManager : MonoBehaviour {
  [Header("Mobile Settings")]
  public RCCP_Settings.MobileController defaultController =
RCCP_Settings.MobileController.TouchScreen;
  public bool allowControllerSwitching = true;

  void Start() {
    // Only setup mobile controls on mobile platforms
    #if UNITY_ANDROID || UNITY_IOS
      SetupMobileController();
    #endif
  }

  void SetupMobileController() {
    // Set default mobile controller
    RCCP.SetMobileController(defaultController);
    Debug.Log($"Mobile controller set to: {defaultController}");

    // Enable controller switching if allowed
    if (allowControllerSwitching) {
      SetupControllerSwitching();
    }
  }
```

```
    void SetupControllerSwitching() {
        // Create UI for switching between mobile control types
        // This would typically involve UI buttons for each control type
    }

    public void SwitchMobileController(int controllerType) {
        if (controllerType >= 0 && controllerType < System.Enum.GetValues(typeof(RCCP_Settings.MobileController)).Length) {
            RCCP_Settings.MobileController newController = (RCCP_Settings.MobileController)controllerType;
            RCCP.SetMobileController(newController);
            Debug.Log($"Mobile controller switched to: {newController}");
        }
    }
}
```

Utility and Maintenance Functions

Utility functions provide maintenance and cleanup operations for optimal performance and visual quality.

Repair and Maintenance

Repair - Vehicle Repair System

// Repair player vehicle

public static void Repair()

// Repair specific vehicle

public static void Repair(RCCP_CarController vehicle)

Effects:

- **Damage Reset** - Resets all damage components to pristine condition
- **Deformation Reset** - Restores original mesh deformation
- **Component Repair** - Repairs all damaged vehicle components
- **Visual Restoration** - Restores original materials and textures

Visual Cleanup

CleanSkidmarks - Skidmark Management

// Clean all skidmarks in scene

public static void CleanSkidmarks()

// Clean specific skidmark index

public static void CleanSkidmarks(int index)

Purpose:

- **Performance Optimization** - Removes accumulated skidmarks that impact performance
- **Visual Cleanup** - Clears visual clutter from extended play sessions
- **Memory Management** - Frees memory used by skidmark textures

Utility Examples

Automated Maintenance System

```
public class VehicleMaintenanceSystem : MonoBehaviour {
  [Header("Maintenance Settings")]
  public float autoRepairInterval = 60f; // Repair every minute
```

```
    public float skidmarkCleanupInterval = 30f; // Clean skidmarks every 30 seconds
    public bool enableAutoMaintenance = true;

    void Start() {
        if (enableAutoMaintenance) {
            InvokeRepeating(nameof(PerformAutoRepair), autoRepairInterval, autoRepairInterval);
            InvokeRepeating(nameof(CleanupSkidmarks), skidmarkCleanupInterval,
skidmarkCleanupInterval);
        }
    }

    void PerformAutoRepair() {
        var playerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;
        if (playerVehicle != null) {
            RCCP.Repair(playerVehicle);
            Debug.Log("Auto-repair performed on player vehicle");
        }
    }

    void CleanupSkidmarks() {
        RCCP.CleanSkidmarks();
        Debug.Log("Skidmarks cleaned up");
    }

    public void ManualRepair() {
        RCCP.Repair();
```

```
        Debug.Log("Manual repair performed");
    }

    public void RepairAllVehicles() {
        var allVehicles = RCCP_SceneManager.Instance.allVehicles;
        foreach (var vehicle in allVehicles) {
            if (vehicle != null) {
                RCCP.Repair(vehicle);
            }
        }
        Debug.Log($"Repaired {allVehicles.Count} vehicles");
    }
}
```

Scene Manager Integration

Understanding RCCP_SceneManager integration is crucial for effective API usage, as it provides the context and state management for all RCCP operations.

Scene Manager Properties

Key Properties Access

```
// Get current player vehicle
RCCP_CarController playerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;

// Get current camera
RCCP_Camera activeCamera = RCCP_SceneManager.Instance.activePlayerCamera;
```

```
// Get current UI canvas
RCCP_UI_Canvas activeUI = RCCP_SceneManager.Instance.activePlayerCanvas;

// Get all vehicles in scene
List<RCCP_CarController> allVehicles = RCCP_SceneManager.Instance.allVehicles;
```

Integration Examples

Scene State Monitor

```
public class SceneStateMonitor : MonoBehaviour {
    [Header("Monitoring")]
    public float updateInterval = 1f;

    void Start() {
        InvokeRepeating(nameof(MonitorSceneState), 0f, updateInterval);
    }

    void MonitorSceneState() {
        var sceneManager = RCCP_SceneManager.Instance;

        if (sceneManager != null) {
            Debug.Log($"Player Vehicle: {(sceneManager.activePlayerVehicle ?
sceneManager.activePlayerVehicle.name : "None")}");
            Debug.Log($"Total Vehicles: {sceneManager.allVehicles.Count}");
            Debug.Log($"Active Camera: {(sceneManager.activePlayerCamera ?
sceneManager.activePlayerCamera.name : "None")}");
        }
```

```
    }
}
```

Code Examples and Best Practices

Complete Vehicle Management System

```
public class VehicleManager : MonoBehaviour {
    [Header("Vehicle Prefabs")]
    public RCCP_CarController[] availableVehicles;

    [Header("Spawn Settings")]
    public Transform[] spawnPoints;
    public bool autoRegisterAsPlayer = true;

    [Header("Management Settings")]
    public bool enableAutoRepair = false;
    public float repairInterval = 60f;

    private RCCP_CarController currentPlayerVehicle;
    private int currentVehicleIndex = 0;

    void Start() {
        // Spawn initial vehicle
        SpawnInitialVehicle();

        // Setup auto-repair if enabled
```

```
    if (enableAutoRepair) {
        InvokeRepeating(nameof(AutoRepairPlayerVehicle), repairInterval, repairInterval);
    }
}

void Update() {
    // Vehicle switching
    if (Input.GetKeyDown(KeyCode.V)) {
        SwitchToNextVehicle();
    }

    // Quick repair
    if (Input.GetKeyDown(KeyCode.F)) {
        RepairPlayerVehicle();
    }

    // Engine toggle
    if (Input.GetKeyDown(KeyCode.E)) {
        ToggleEngine();
    }
}

void SpawnInitialVehicle() {
    if (availableVehicles.Length > 0 && spawnPoints.Length > 0) {
        Transform spawnPoint = spawnPoints[0];
        RCCP_CarController vehicle = RCCP.SpawnRCC(
```

```
            availableVehicles[currentVehicleIndex],
            spawnPoint.position,
            spawnPoint.rotation,
            autoRegisterAsPlayer,
            true,
            true
        );

        if (autoRegisterAsPlayer) {
            currentPlayerVehicle = vehicle;
        }
    }
}

public void SwitchToNextVehicle() {
    // Destroy current vehicle if exists
    if (currentPlayerVehicle != null) {
        RCCP.DeRegisterPlayerVehicle();
        Destroy(currentPlayerVehicle.gameObject);
    }

    // Move to next vehicle
    currentVehicleIndex = (currentVehicleIndex + 1) % availableVehicles.Length;

    // Spawn new vehicle
    Transform spawnPoint = spawnPoints[Random.Range(0, spawnPoints.Length)];
```

```
    currentPlayerVehicle = RCCP.SpawnRCC(
        availableVehicles[currentVehicleIndex],
        spawnPoint.position,
        spawnPoint.rotation,
        true,
        true,
        true
    );

    Debug.Log($"Switched to: {currentPlayerVehicle.name}");
}

public void RepairPlayerVehicle() {
    if (currentPlayerVehicle != null) {
        RCCP.Repair(currentPlayerVehicle);
        Debug.Log("Player vehicle repaired");
    }
}

public void ToggleEngine() {
    if (currentPlayerVehicle != null) {
        var engine = currentPlayerVehicle.GetComponent<RCCP_Engine>();
        if (engine != null) {
            RCCP.SetEngine(currentPlayerVehicle, !engine.engineRunning);
            Debug.Log($"Engine {(engine.engineRunning ? "started" : "stopped")}");
        }
```

```
    }
  }

  void AutoRepairPlayerVehicle() {
    if (currentPlayerVehicle != null) {
      RCCP.Repair(currentPlayerVehicle);
    }
  }
}
```

Best Practices Summary

Always Validate Objects

```
// Good practice
if (vehicle != null && vehicle.gameObject.activeInHierarchy) {
  RCCP.SetControl(vehicle, true);
}

// Avoid this
RCCP.SetControl(vehicle, true); // Could cause null reference
```

Use Scene Manager Properties

```
// Good practice
var playerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;
if (playerVehicle != null) {
  // Operate on player vehicle
}
```

```
// Avoid finding objects manually
var playerVehicle = FindObjectOfType<RCCP_CarController>(); // Inefficient and unreliable
```

Handle State Changes Properly

```
// Good practice - deregister before destroying
RCCP.DeRegisterPlayerVehicle();
Destroy(vehicleToDestroy.gameObject);

// Avoid destroying registered vehicles without deregistering
Destroy(playerVehicle.gameObject); // Can cause scene manager issues
```

Error Handling and Troubleshooting

Common Issues and Solutions

Vehicle Not Responding to Input

```
public void DiagnoseVehicleInput(RCCP_CarController vehicle) {
    if (vehicle == null) {
        Debug.LogError("Vehicle is null!");
        return;
    }

    // Check if vehicle is registered as player vehicle
    if (RCCP_SceneManager.Instance.activePlayerVehicle != vehicle) {
        Debug.LogWarning("Vehicle is not registered as player vehicle");
        RCCP.RegisterPlayerVehicle(vehicle);
```

```
    }

    // Check control state
    if (!vehicle.canControl) {
        Debug.LogWarning("Vehicle control is disabled");
        RCCP.SetControl(vehicle, true);
    }

    // Check input component
    var inputComponent = vehicle.GetComponent<RCCP_Input>();
    if (inputComponent == null) {
        Debug.LogError("Vehicle missing RCCP_Input component!");
    }
}
```

Performance Issues

```
public void OptimizePerformance() {
    // Clean up skidmarks regularly
    RCCP.CleanSkidmarks();

    // Limit number of active vehicles
    var allVehicles = RCCP_SceneManager.Instance.allVehicles;
    if (allVehicles.Count > 10) {
        Debug.LogWarning($"High vehicle count: {allVehicles.Count}. Consider optimizing.");
    }
```

```
    // Check for memory leaks
    System.GC.Collect();
}
```

Error Prevention

Safe API Usage Pattern

```
public class SafeAPIUsage : MonoBehaviour {
    public bool SafeSpawnVehicle(RCCP_CarController prefab, Vector3 position, Quaternion rotation) {
        try {
            // Validate inputs
            if (prefab == null) {
                Debug.LogError("Vehicle prefab is null");
                return false;
            }

            if (RCCP_SceneManager.Instance == null) {
                Debug.LogError("RCCP_SceneManager not found in scene");
                return false;
            }

            // Check spawn position
            if (Physics.CheckSphere(position, 2f)) {
                Debug.LogWarning("Spawn position is occupied");
                return false;
            }
```

```
        // Spawn vehicle
        RCCP_CarController vehicle = RCCP.SpawnRCC(prefab, position, rotation, true, true, true);

        if (vehicle != null) {
            Debug.Log($"Successfully spawned vehicle: {vehicle.name}");
            return true;
        } else {
            Debug.LogError("Failed to spawn vehicle");
            return false;
        }
    }
    catch (System.Exception e) {
        Debug.LogError($"Exception during vehicle spawn: {e.Message}");
        return false;
    }
}

public bool SafeRegisterVehicle(RCCP_CarController vehicle) {
    try {
        if (vehicle == null || !vehicle.gameObject.activeInHierarchy) {
            Debug.LogError("Cannot register null or inactive vehicle");
            return false;
        }

        RCCP.RegisterPlayerVehicle(vehicle, true, true);
```

```
            return true;
        }
        catch (System.Exception e) {
            Debug.LogError($"Exception during vehicle registration: {e.Message}");
            return false;
        }
    }
}
```

Performance Considerations

Memory Management

Efficient Vehicle Spawning

```
public class EfficientVehicleSpawner : MonoBehaviour {
    [Header("Performance Settings")]
    public int maxActiveVehicles = 10;
    public float vehicleCleanupDistance = 100f;

    private Queue<RCCP_CarController> vehiclePool = new Queue<RCCP_CarController>();
    private List<RCCP_CarController> activeVehicles = new List<RCCP_CarController>();

    void Update() {
        // Clean up distant vehicles
        CleanupDistantVehicles();
```

```
        // Maintain vehicle limit
        EnforceVehicleLimit();
    }

    void CleanupDistantVehicles() {
        var playerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;
        if (playerVehicle == null) return;

        for (int i = activeVehicles.Count - 1; i >= 0; i--) {
            var vehicle = activeVehicles[i];
            if (vehicle == null) {
                activeVehicles.RemoveAt(i);
                continue;
            }

            float distance = Vector3.Distance(playerVehicle.transform.position,
vehicle.transform.position);
            if (distance > vehicleCleanupDistance) {
                ReturnVehicleToPool(vehicle);
                activeVehicles.RemoveAt(i);
            }
        }
    }

    void EnforceVehicleLimit() {
        while (activeVehicles.Count > maxActiveVehicles) {
```

```
            var oldestVehicle = activeVehicles[0];
            ReturnVehicleToPool(oldestVehicle);
            activeVehicles.RemoveAt(0);
        }
    }

    void ReturnVehicleToPool(RCCP_CarController vehicle) {
        if (vehicle != null) {
            vehicle.gameObject.SetActive(false);
            vehiclePool.Enqueue(vehicle);
        }
    }
}
```

Memory Optimization Tips

- **Limit Active Vehicles** - Keep the number of active vehicles reasonable
- **Clean Skidmarks Regularly** - Use RCCP.CleanSkidmarks() periodically
- **Vehicle Pooling** - Reuse vehicle instances instead of constant spawning/destroying
- **LOD Management** - Use RCCP's built-in LOD system for distant vehicles

---

Advanced Implementation Patterns

Event-Driven Vehicle Management

```
public class EventDrivenVehicleManager : MonoBehaviour {
    [Header("Events")]
    public UnityEvent<RCCP_CarController> OnVehicleSpawned;
```

```
public UnityEvent<RCCP_CarController> OnVehicleDestroyed;
public UnityEvent<RCCP_CarController> OnPlayerVehicleChanged;

private RCCP_CarController lastPlayerVehicle;

void Start() {
    // Subscribe to RCCP events
    RCCP_Events.OnRCCPSpawned += OnVehicleSpawnedEvent;
    RCCP_Events.OnRCCPDestroyed += OnVehicleDestroyedEvent;
}

void Update() {
    // Check for player vehicle changes
    var currentPlayerVehicle = RCCP_SceneManager.Instance.activePlayerVehicle;
    if (currentPlayerVehicle != lastPlayerVehicle) {
        OnPlayerVehicleChangedEvent(currentPlayerVehicle);
        lastPlayerVehicle = currentPlayerVehicle;
    }
}

void OnVehicleSpawnedEvent(RCCP_CarController vehicle) {
    Debug.Log($"Vehicle spawned: {vehicle.name}");
    OnVehicleSpawned?.Invoke(vehicle);

    // Setup new vehicle
    SetupNewVehicle(vehicle);
```

```
    }

    void OnVehicleDestroyedEvent(RCCP_CarController vehicle) {
        Debug.Log($"Vehicle destroyed: {vehicle.name}");
        OnVehicleDestroyed?.Invoke(vehicle);
    }

    void OnPlayerVehicleChangedEvent(RCCP_CarController newPlayerVehicle) {
        Debug.Log($"Player vehicle changed to: {(newPlayerVehicle ? newPlayerVehicle.name : "None")}");
        OnPlayerVehicleChanged?.Invoke(newPlayerVehicle);
    }

    void SetupNewVehicle(RCCP_CarController vehicle) {
        // Add any default components or configurations
        // Set initial states
        // Configure vehicle-specific settings
    }

    void OnDestroy() {
        // Unsubscribe from events
        RCCP_Events.OnRCCPSpawned -= OnVehicleSpawnedEvent;
        RCCP_Events.OnRCCPDestroyed -= OnVehicleDestroyedEvent;
    }
}
```
