Table of Contents

---

1. Overview

The `RCC_DashboardObjects` script provides **3D dashboard** functionality for Realistic Car Controller (RCC) vehicles. Unlike a typical UI-based dashboard, this script manipulates actual **GameObjects** (e.g., 3D needles and dials) in the scene, rotating them according to the vehicle's RPM, speed, fuel, or heat level. It also manages **interior lights** that can be toggled on/off depending on whether the vehicle headlights are on.

This system allows for visually authentic dashboards or cockpit interiors where needles and lights physically update in the 3D space of the vehicle model.

---

2. Class Declaration

```
public class RCC_DashboardObjects : RCC_Core {
    // ...
}
```

- Inherits from `RCC_Core`, a base class in RCC.
- Relies on a reference to an **RCC_CarControllerV4** (usually `CarController` in `RCC_Core`) for real-time data.

---

3. Nested Classes and Structures

`RCC_DashboardObjects` uses several **serializable** nested classes to handle different types of dials and lights. Each nested class organizes its logic for initialization and updating.

3.1 `RPMDial`

```
[System.Serializable]
public class RPMDial {
    public GameObject dial;
    public float multiplier = .05f;
    public RotateAround rotateAround = RotateAround.Z;
    private Quaternion dialOrgRotation = Quaternion.identity;
    public Text text;
    public void Init() { ... }
    public void Update(float value) { ... }
}
```

- **`dial`**: The GameObject representing the RPM needle or dial in 3D space.
- **`multiplier`**: Multiplies the incoming RPM value to determine dial rotation.
- **`rotateAround`**: Defines which axis (X, Y, or Z) to rotate around.
- **`dialOrgRotation`**: Stores the **original local rotation** of the dial so that updates can be relative to its starting orientation.
- **`text`**: (Optional) UI Text to display numeric RPM data.

The **`Init()`** method records `dialOrgRotation`, while **`Update(float value)`** applies a rotation transform based on `value` (the vehicle's current RPM).

3.2 `SpeedoMeterDial`

```
[System.Serializable]
public class SpeedoMeterDial {
    public GameObject dial;
    public float multiplier = 1f;
    public RotateAround rotateAround = RotateAround.Z;
    private Quaternion dialOrgRotation = Quaternion.identity;
    public Text text;
    public void Init() { ... }
    public void Update(float value) { ... }
}
```

- Almost identical to `RPMDial`, but intended for the vehicle's **speed** (in KMH or MPH).
- The `multiplier` determines how many degrees per unit of speed.

3.3 `FuelDial`

```
[System.Serializable]
public class FuelDial {
    public GameObject dial;
    public float multiplier = .1f;
    public RotateAround rotateAround = RotateAround.Z;
    private Quaternion dialOrgRotation = Quaternion.identity;
    public Text text;
    public void Init() { ... }
    public void Update(float value) { ... }
}
```

- Handles **fuel** level representation.
- Uses the same approach as RPM and speed dials.

3.4 `HeatDial`

```
[System.Serializable]
public class HeatDial {
    public GameObject dial;
    public float multiplier = .1f;
    public RotateAround rotateAround = RotateAround.Z;
    private Quaternion dialOrgRotation = Quaternion.identity;
    public Text text;
    public void Init() { ... }
    public void Update(float value) { ... }
}
```

- Shows **engine heat** levels on a dial.

3.5 `InteriorLight`

```
[System.Serializable]
public class InteriorLight {
    public Light light;
    public float intensity = 1f;
    public LightRenderMode renderMode = LightRenderMode.Auto;
    public void Update(bool state) { ... }
}
```

- **`light`**: A reference to a Unity `Light` component inside the vehicle's interior.
- **`intensity`**: How bright the light should be when active.
- **`renderMode`**: Unity's render mode (Auto, Important, NotImportant).
- **`Update(bool state)`**: Sets `light.intensity` to `intensity` if `state` is `true`; otherwise sets `0f` for off.

---

4. Primary Fields and Properties

4.1 Dial Fields

Inside `RCC_DashboardObjects`, you'll find:

public RPMDial rPMDial = new RPMDial();
public SpeedoMeterDial speedDial = new SpeedoMeterDial();
public FuelDial fuelDial = new FuelDial();
public HeatDial heatDial = new HeatDial();

Each of these fields corresponds to one of the nested dial classes. They are displayed in the Inspector where you can assign the respective `dial` GameObjects, set multipliers, and optionally assign a `Text` UI component.

4.2 Interior Lights Array

public InteriorLight[] interiorLights = new InteriorLight[0];

This array can contain multiple `InteriorLight` objects for different dashboard or cockpit lights, enabling them to collectively toggle.

4.3 Enums

`public enum RotateAround { X, Y, Z }`

- Defines the axis on which the dial will rotate.
- Each dial class uses this setting to decide which axis to apply the rotation in `Update()`.

---

5. Lifecycle Methods

5.1 `Awake()`

private void Awake() {

```
    rPMDial.Init();
    speedDial.Init();
    fuelDial.Init();
    heatDial.Init();
}
```

- Calls `Init()` on each dial class to store their original local rotations.
- Ensures no rotation occurs before the script knows the starting orientation of the dials.

5.2 `Update()`

```
private void Update() {
    if (!CarController)
        return;

    Dials();
    Lights();
}
```

- Checks if a valid `CarController` is present. This is typically the `RCC_CarControllerV4` reference from `RCC_Core`.
- If no vehicle is assigned, it does nothing. Otherwise, calls two internal methods: `Dials()` and `Lights()`.

---

6. Key Internal Methods

6.1 `Dials()`

```
private void Dials() {
    if (rPMDial.dial != null)
        rPMDial.Update(CarController.engineRPM);

    if (speedDial.dial != null)
        speedDial.Update(CarController.speed);

    if (fuelDial.dial != null)
        fuelDial.Update(CarController.fuelTank);

    if (heatDial.dial != null)
        heatDial.Update(CarController.engineHeat);
}
```

- Reads **engineRPM**, **speed**, **fuelTank**, **engineHeat** from `CarController`.
- Passes them to each dial's `Update()` method.

6.2 `Lights()`

```
private void Lights() {
    for (int i = 0; i < interiorLights.Length; i++)
        interiorLights[i].Update(CarController.lowBeamHeadLightsOn);
}
```

- Loops over `interiorLights`, updating each one based on whether the vehicle's low-beam headlights are active (`CarController.lowBeamHeadLightsOn`).
- You could adapt this if you want the interior lights to respond differently (e.g., also turn on with high beams or door opening).

---

7. Usage Notes and Best Practices

1. **Assigning the Vehicle**
    - `CarController` is inherited from `RCC_Core` (e.g., `public RCC_CarControllerV4 CarController`). Make sure this is set either by auto-assignment or manually.
2. **Configuring Dials**
    - Each dial's `multiplier` may need tweaking. For instance, if your speedo dial only goes up to 220 and the 3D needle has a specific angle range, adjust accordingly.
    - Make sure your 3D models' local zero rotation aligns with how you want the dial to "start."
3. **Interior Lights**
    - Typically used for minor cockpit illumination. Assign an actual **Unity Light** component to `interiorLights[i].light`.
    - If you have multiple lights (e.g., overhead, footwell), add them to the array.
4. **Forward vs. Negative Rotation**
    - Dial rotation is done via `Quaternion.AngleAxis(-multiplier * value, axis)`. If your needle moves in the wrong direction, adjust the sign or axis.
5. **Text Fields**
    - Optional: If you assign a `Text` field in each dial, the script displays the raw numeric value.

    - If you prefer an alternate format (e.g., “120 KM/H”), modify the `.text = value.ToString("F0")` line.
6. **Performance**
    - Only a handful of rotation operations run per frame—this is generally lightweight.
    - For complex dashboards, keep in mind that multiple `Light` components can affect rendering performance.
7. **Extensions**
    - Additional dials (e.g., turbo pressure or NoS level) can be implemented similarly—create another nested dial class or replicate an existing one.

---

8. Summary

`RCC_DashboardObjects` brings **physical, in-game dashboards** to life by rotating 3D needle GameObjects for engine RPM, speed, fuel, and engine heat. It also manages **interior lighting** based on the vehicle’s headlight state. Each dial encapsulates rotation logic, simplifying setup and making it easy to tailor for different vehicle cockpit designs.

Use this script when you need a **realistic, interactive interior** for player-driven or AI-driven vehicles where gauges are 3D objects rather than 2D UI elements. Simply attach this script inside your vehicle’s dashboard hierarchy, link the appropriate dial GameObjects, and enjoy a fully animated, immersive cockpit experience.