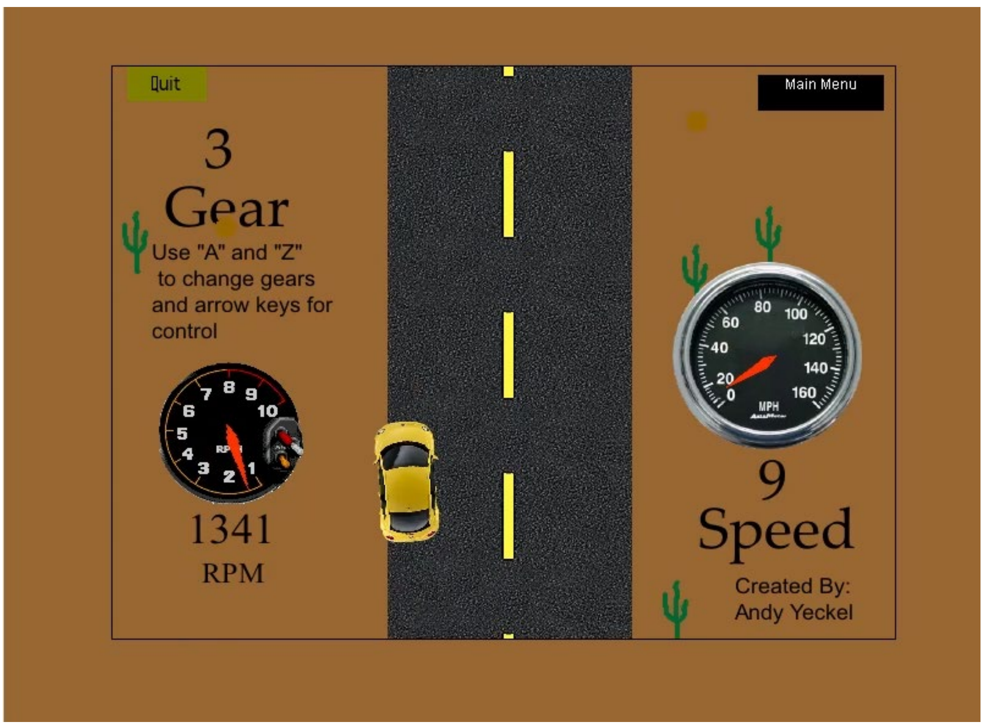
Technical Summary: Car Game with Realistic Physics and Dynamic Wrapping (December 2001-January 2002)



Overview

This project demonstrates an advanced 2D interactive car simulation using **Macromedia Director 8.5**. It incorporates **realistic motion physics**, **friction-based deceleration**, and **infinite wrapping mechanics**, creating a visually immersive and tactile experience. The game simulates speed, steering, and dynamic wrapping to create the illusion of acceleration, where yellow road lines appear closer together as the car moves faster. This project served as an early exploration of physics-driven interactivity and foundational principles later applied to UI carousels and other interactive systems.

Core Features

1. Realistic Motion Physics

- **Dynamic Speed Adjustment**:
    - Speed (`mygearspeed`) is proportional to user input and gradually adjusted over time using a **friction model**:

      ```
      mygearspeed = mygearspeed * frictionFactor;
      ```

    - Friction ensures smooth deceleration, mimicking real-world motion resistance.
- **Torque and Acceleration**:
    - Acceleration and deceleration are influenced by a torque-based formula, providing a responsive and tactile driving experience:

      ```
      torque = (wheelspeed + mygear) / (wheelspeed + 2);
      horsepower = (wheelspeed + mygear) * (torque * hp);
      ```

---

2. Infinite Wrapping Mechanics

- The game implements a **wrapping system** for the yellow road lines and the car itself:
    - Lines that move out of bounds (`<= -200`) are repositioned to the other end (`+700`) for seamless looping:

      ```
      if sprite(me.spritenum).locv <= -200 then
          sprite(me.spritenum).locv = sprite(me.spritenum).locv + 700;
      end if;
      ```

    - This ensures that the stage maintains continuous motion, creating the illusion of an infinite road.

---

3. Visual Feedback and Speed Simulation

- **Yellow Line Compression**:
  - As the car accelerates, the wrapping mechanism causes the yellow road lines to appear closer together, simulating the effect of increased speed.
- **Tachometer and Steering Feedback**:
  - On-screen indicators provide real-time updates on engine speed (`tach`) and steering adjustments:

    ```
    tach = integer(850 + ((speed + 1) / (mygear - tire_grip) /
    (torque + 1)) * 600);
    ```

- **Smooth Steering**:
  - Steering is influenced by speed and gear state, resulting in tighter turns at lower speeds and wider turns at higher speeds:

    ```
    steering = max(-1, min(1, steering));
    angle = angle + ((steering * actual_speed) * 0.0005);
    ```

---

4. Interactivity

- **User Input**:
  - Players control speed, steering, and braking using keyboard inputs, which dynamically adjust the car's motion and direction.
- **Dynamic Response**:
  - The car's behavior is affected by real-time calculations of speed, torque, and friction, creating a highly responsive experience.

Significance

1. Early Exploration of Physics-Driven Interactivity

- This project predates many modern physics-based ui systems, demonstrating an early understanding of motion and tactile feedback in a 2D environment.
- The inclusion of **realistic friction**, **proportional acceleration**, and **wrapping mechanics** set it apart from typical side-scrollers of the time.

2. Foundation for Later Innovations

- The project directly influenced later works, such as the **Sixspeedmedia carousel**, by introducing:
  - **Friction models** for smooth transitions.
  - **Infinite wrapping** for continuous motion.
  - **Proportional motion dynamics** tied to user input.

3. Multi-Platform Compatibility

- Designed for both **in-browser (via Shockwave)** and **standalone (projector)**
- Shockwave enabled rich interactivity in web browsers, while projectors provided high-performance, standalone functionality.

4. Historical Significance

- Developed in January 2001,

Technical Highlights

Feature	Implementation
Friction Model	Dynamic speed decay using `mygearspeed * frictionFactor`.
Infinite Wrapping	Road lines loop seamlessly using boundary checks and repositioning logic.
Dynamic Speed Adjustment	Speed influenced by torque, friction, and user input.
Yellow Line Compression	Faster wrapping creates the visual effect of lines getting closer together.
User Control	Keyboard inputs dynamically adjust speed, steering, and braking.
Responsive Feedback	Real-time updates for tachometer, speed, and steering.

Conclusion

This car game has realistic motion mechanics and interactive feedback. Its wrapping system and tactile motion physics make it a precursor to modern inertia based UI/UX paradigms and interactive carousels.

Global Variables

```
global mygearspeed, speed, wrapHeight, stageHeight
```

- **mygearspeed**: The primary speed affecting the sprite's movement.
- **speed**: A second speed variable (used in `updateUI`), which might be used for display or other logic.
- **wrapHeight**: The amount of vertical space used for sprite wrapping (essentially how far the sprite must travel before it snaps back).

- **stageHeight**: The visible height of the stage (calculated at runtime).

`beginSprite` Handler

```
on beginSprite me
  setupSprite(me)
end
```

- When a Director movie starts (or the sprite first appears on stage), `beginSprite` is called.
- It calls `setupSprite(me)` to initialize sprite properties and set up the global variables.

`setupSprite` Handler

```
on setupSprite me
  global stageHeight, wrapHeight
  spriteRef = sprite(me.spritenum)

  -- Set registration point and calculate stage and wrap dimensions
  spriteRef.member.regPoint = point(0, 0)
  stageHeight = the stageBottom - the stageTop
  wrapHeight = spriteRef.height - stageHeight

  -- Initialize global variables
  mygearspeed = 0
  speed = 0
end
```

1. **Obtain a reference to the current sprite**:

   ```
   spriteRef = sprite(me.spritenum)
   ```

2. **Set the registration point**:

   ```
   spriteRef.member.regPoint = point(0, 0)
   ```

   This makes the top-left corner of the member its reference point.

3. **Calculate stage and wrap dimensions**:

```
stageHeight = the stageBottom - the stageTop
wrapHeight = spriteRef.height - stageHeight
```

- `stageHeight` is how tall the visible stage area is.
- `wrapHeight` is how much vertical space above/below is needed so that if the sprite goes out of view, it can "wrap" back around.

4. **Initialize speeds**:

```
mygearspeed = 0
speed = 0
```

Both speed variables start at zero.

`enterFrame` Handler

```
on enterFrame me
  moveSprite(me)
  applyFriction()
end
```

- **Called every frame** of the Director movie (roughly equivalent to "update" in modern game engines).
- Calls `moveSprite(me)` to shift the sprite's position.
- Calls `applyFriction()` to gradually slow down the speed value.

`moveSprite` Handler

```
on moveSprite me
  spriteRef = sprite(me.spritenum)

  -- Calculate movement amount based on speed
  moveAmount = -mygearspeed / 3
  spriteRef.locv = spriteRef.locv + moveAmount

  -- Wrap sprite if it moves out of bounds
  if spriteRef.locv > 0 then
    spriteRef.locv = spriteRef.locv - wrapHeight
  else if spriteRef.locv < -wrapHeight then
```

```
    spriteRef.locv = spriteRef.locv + wrapHeight
  end if
end
```

1. **Get current sprite reference**:

```
spriteRef = sprite(me.spritenum)
```

2. **Calculate movement**:

```
moveAmount = -mygearspeed / 3
spriteRef.locv = spriteRef.locv + moveAmount
```

   - The `locv` property is the vertical location of the sprite.
   - Dividing `mygearspeed` by 3 makes the sprite move more slowly than the raw speed value might suggest (i.e., a damping effect).
   - The negative sign (`-mygearspeed`) inverts the direction if needed.

3. **Vertical wrapping**:

```
if spriteRef.locv > 0 then
  spriteRef.locv = spriteRef.locv - wrapHeight
else if spriteRef.locv < -wrapHeight then
  spriteRef.locv = spriteRef.locv + wrapHeight
end if
```

   - If the sprite moves too far down (locv > 0), it repositions the sprite higher up by subtracting `wrapHeight`.
   - If it moves too far up (locv < -wrapHeight), it repositions it back down by adding `wrapHeight`.
   - This "wraps" the sprite so it continuously scrolls in a loop.

`applyFriction`

```
on applyFriction
  global mygearspeed
  frictionFactor = 0.97  -- Adjust this to control friction
  mygearspeed = mygearspeed * frictionFactor

  -- Ensure speed doesn't drop below a minimum threshold
  if abs(mygearspeed) < 0.1 then
    mygearspeed = 0
```

```
  end if
end
```

- **Gradually reduces `mygearspeed`** each frame:

  ```
  mygearspeed = mygearspeed * frictionFactor
  ```

  This creates a deceleration effect.

- **Minimum threshold**:

  ```
  if abs(mygearspeed) < 0.1 then
    mygearspeed = 0
  end if
  ```

  Prevents small floating-point values from never reaching zero, so the sprite can come to a complete stop.

`updateUI`

```
on updateUI
  -- Example: Update text fields or visual indicators
  member(18).text = string(-mygearspeed)
  member(19).text = "Speed: " & string(speed)
end
```

- This is an optional UI update routine.
- Could be called elsewhere in the code (e.g., in `enterFrame` or when needed).
- Shows how you might display the current speed values in text members (e.g., text fields on stage).

Summary

1. **Initialization**: When the sprite starts, `setupSprite` calculates the stage size, sets the sprite's registration point, and initializes the global speed variables.
2. **Movement & Wrapping**: Every frame, `moveSprite` adjusts the sprite's vertical position based on `mygearspeed`, then checks if the sprite is off-screen, wrapping it around if necessary.

3. **Friction**: After movement each frame, `applyFriction` reduces `mygearspeed`, creating a smooth deceleration.
4. **UI Updates**: An optional `updateUI` routine can display or log the current speed values.

In essence, this code continuously moves a sprite in a loop (like a scrolling background). The friction and wrapping give it a smooth, never-ending scrolling effect.