1 Introduction

Object oriented programming (OOP) is a powerful programming concept that allows entities to have actions, to own other entities and to have properties. For example a car will own an engine (another entity), it has a quantity of fuel (a property) and it can start its engine (an action).

Because entities can own other entities, we need to be able to structure our information flow to keep it as hierarchical as possible. An engine is part of a car, it does not own a car, this needs to be reflected in our class structure. If an owned class required information from it's owner, it's owner should tell it (eg the engine can only run when there is fuel. How do we tell it when there is fuel?).

2 Case Study: Transmission System of a Car

Let's say we want to model the internals of a car with a high level of accuracy. How should we structure our classes? First let's list the physical objects that our system contains:

- A car
  - An Engine
  - A Transmission
    - A gearbox
    - A differential
  - Wheel1
  - Wheel2
  - Wheel3
  - Wheel4

These are physical objects we can touch. You can touch an engine, you can find the transmission on a car, etc. The indentations show ownership.

The first thing to do in any class is to contruct our class heirachy:

```
1 class Car:
2     def __init__(self)
3         self.engine = Engine()
4         self.transmission = Transmission()
5         self.wheel1 = Wheel('driven')
6         self.wheel2 = Wheel('driven')
7         self.wheel3 = Wheel('free')
8         self.wheel4 = Wheel('free')
```

So now that we've discovered what our system is, and made it in python, how do we pass data in and around it? At some point, the user puts his foot on the pedal, and that feeds fuel into the engine. So it makes sense that from outside, we need to run a function like:

```
1 car.press_pedal(100%) #The guys flooring it, Ok?
```

Now what does the car do with this data? Well, inside the press_pedal function, it passes it along to the engine. And it get's back ... the amount of torque the motor is outputting:

```
def press_pedal(self, throttle):
    engine_torque = self.engine.getTorque(throttle)
```

We can pass this data through the transmission and onto the wheels:

```
    wheel_torque = self.tranmsission.engine_torque_to_wheels(engine_torque)
    self.wheel1.apply_torque(wheel_torque/2)
    self.wheel2.apply_torque(wheel_torque/2)
```

And there we have a nice logical way to apply torque from one end of the car to the other.

Now this works for a simplistic car model, but let's say we want to add things like:

- The motors torque also depends on the engines RPM
- Drag from the car
- Weight distribution on the tires (and skidding)

2.1 Motor RPM

Somehow we need to figure out the RPM of the motor, so that the motor can calculate it's output torque more accurately. Well, this is regulated by the wheel speed and the transmission, so we can simply pass some data the other way through the transmission:

```
def calculate_rpm_of_motor(self):
    motor_speed = self.transmission.wheel_speed_to_motor_speed(wheel1.angular_velo
    return motor_speed
```

And then we can modify our press pedal function to:

```
def press_pedal(self, throttle):
    engine_speed = self.calculate_rpm_of_motor()
    engine_torque = self.engine.getTorque(throttle, engine_speed)
    wheel_torque = self.tranmsission.engine_torque_to_wheels(engine_torque)
    self.wheel1.apply_torque(wheel_torque/2)
    self.wheel2.apply_torque(wheel_torque/2)
```

2.2 More Accurate forces on car

So we've passed some data all the way along to the wheels and abandoned it there. If this is implemented in a game engine with a physics engine, it can be left there as well, because the wheels can apply a torque and you can set a physics parameter for the drag. If the physics engine doesn't expose the required parameters (eg Blender Game Engine[1]) then they have to be implemented manually.

[1] Actually, in exposing the Linear Damping it does. Any drag force can be simplified to a linear damping coefficient

Now the wheels need to apply a force onto the car body. But we need to keep the car's internals restricted to they operate in. (a wheel shouldn't know about the transmission etc.) So how do we deal with this? Well, we apply the torque to the wheels, and it makes sense that at the same time, we can get the force from each wheel back:

```
    wheel1_force = self.wheel2.apply_torque(wheel_torque/2)
```

And apply it on the body at the location of the wheel. How this is done depends vastly on the physics engine used. In BGE, you have to turn the force magnitude into a vector, and then apply it using the applyImpulse function - allowing for the framerate.

Now the drag of the car is simply a coefficient of the speed, and can be handled separately to the press_pedal function, effectively managing itself.

Now we can also calculate the weight distribution on each tire. This is closely coupled with the chassis roll. As the vehicle accelerates (remember that turning corners also induces an acceleration), the difference between it's center of gravity and the location of the force applied from it's wheels induces a rolling force. This alters the loading on each wheel. The math for finding the loading on each tire is well documented, and it should be run by the car class, with data being passed down into the ...

2.3 Suspension

At some point we need to handle suspension. This is a huge topic with people dedicating their lives to studying suspension systems. But, because this is only a game, we can assume that a suspension system acts a lot like ... a PID controller. Assuming that is implemented, we can integrate it into our car model like:

- A car
  - An Engine
  - A Transmission
    - A gearbox
    - A differential
  - Suspension1
    - Wheel1
  - Suspension2
    - Wheel2
  - Suspension3
    - Wheel3
  - Suspension4
    - Wheel4

To use the suspension, we may want to pass in the vertical force on each wheel, and get back the displacement:

```
def apply_force(self, force):
    return self.PID.update(force)
```

PID controllers are amazing. Note that this doesn't allow for the tire breaking contact with the ground.

2.4 More accurate tire model

Due to the amazingness of physics, when calculating the vertical force on a wheel, we can ignore the suspension - the forces are the same on both sides.

The thing to realize is that because we have the class structure, we can delegate these tasks to lower-level classes. Skidding can be abandoned to the wheels, where instead of the wheel.apply_torque function being:

```
def apply_torque(self, torque):
    return torque/self.radius
```

It now needs to know things such as it's static and dynamic friction coefficients, the force downwards on it and so on. A function that supports skidding would be:

```
def apply_torque(self, torque, vertical_force):
    max_static_force = self.static_coefficient * vertical_force
    min_kinetic_force = self.kinetic_coefficient * vertical_force
    applied_force = torque/self.radius

    if self.skidding == True:
        if applied_force < min_kinetic_force:
            self.skidding = False
        return min(applied_force, min_kinetic_force)
    else:
        if applied_force > max_static_force:
            self.skidding = True
        return min(applied_force, max_static_force)
```

This implements the histeresis seen in real systems.

2.5 Body Roll

In the section dealing with making more accurate forces on the car, we have to modelled our suspension and need to use this to find the angle of the chassis.

We know the vertical forces on each wheel, and hence we can calculate the compression of each suspension element. If this has been calculated correctly, the heights should make a flat platform. The angle of the chassis can then be calculate in the same way that you can find the normal of a plane passing through three points - simply discarding the fourth (or using it for error checking). This is well documented elsewhere: two vectors are made from three points. These are then cross-producted to give a normal vector.

The position of the body is simply the interpolation of the four points of the car.

2.6 Final notes on the car model

You may have noticed that by the time we've implemented body roll, there's not much that the physics engine is actually doing. What we've done here is write our own physics engine - which is far from ideal.