Term Project about car physics simulation in Unity

Realistic Multimedia and Animation (M.Sc. Informatics Engineering) 2022-2023

Instructor: Dr. Ioannis Pachoulakis

Michael Xirakis
Graduate Student of "Informatics Engineering"
Hellenic Mediterranean University
Department of Electrical & Computer Engineering
Heraklion, Greece 71410
Email: mth290@edu.hmu.gr

Project's Report

The purpose of this project is instructive and has to do with car physics development. Specifically, based on the knowledge gained from the semester courses, about Physics and Mathematics mainly used in video game development, we will use some fundamental laws of physics, such as Newton's laws, and some other concepts, such as permutation matrices from Linear Algebra, so we can build our game and see the practical implementation of the theory and if it produces real results.

We will mainly focus on four concepts that are based on pure physics and mathematics. These are the suspension, steering, wheels spinning, and an example of a translation matrix about the position of the popup headlights. There are also some other concepts that will be presented generally afterward such as engine torque and gear automatic change, but they are not based on pure physics, but mostly in a combination of experimentation, intuition, and tricks, to achieve a more visually pleasing result for the gamer. The first concept is about the car's suspension. Based on Hooke's law about the elasticity of objects, we will try to create a suspension system for the car. Hooke's law says that:

$$F_s = k * \delta x \Leftrightarrow$$

force = stiffness * distance

So the strain of the spring is calculated from the above equation (when it is stretched or compressed). In action what we want to do is to calculate the strain for all 4 wheels, depending on their position and wheel radius (because the front springs are lower than the rear and also the front wheels of the car have a smaller radius), using **Raycasting** to measure the distance from springs origin to the ground and then apply an opposite force to the **Rigidbody** component of the car to lift it. Of course, this could be done much easier with less effort, using the **Wheel Collider** component of Unity and setting up some parameters like **Damping**, but as mentioned earlier, the purpose is to implement as much physics as possible, by hand. The corresponding part of the code in Unity that implements this equation is in the "**Suspension.cs**" file:

```
if (Physics.Raycast(transform.position, -transform.up, out RaycastHit hit, maxLength + wheelRadius)) {
        lastLength = springLength;
        springLength = hit.distance - wheelRadius;
        springLength = Mathf.Clamp(springLength, minLength, maxLength);
        springVelocity = (lastLength - springLength) / Time.fixedDeltaTime;
        springForce = springStiffness * (restLength - springLength);
        damperForce = damperStiffness * springVelocity + Physics.gravity.y *
                Mathf.Sin(Mathf.Sqrt(springStiffness/1470) * Time.fixedDeltaTime);
        suspensionForce = (springForce + damperForce) * transform.up;
}
```

What we are doing here is keeping the previous length of the spring into the variable "**lastLength**", then storing the current (next frame) length depending on the distance between the spring origin minus the wheel radius into "**springLength**". The difference between these two variables shows the amount of strain (deformation) of the spring. We also use **Mathf.Clamp()** function to keep the stretching and compressing into some logical bounds (e.g. we don't want our springs to be stretched or compressed infinitely). Then we calculate the velocity of the spring, meaning how fast the spring comes to equilibrium in "**springVelocity**", which is equal to the deformation over some time. Then the "**springForce**" variable represents $\mathbf{F_s}$, which is proportional to "**springStiffness**" **(**which is **k** of Hooke's equation) multiplied by the distance **δx** (**restLength - springLength**). Another important parameter is to add some "*resistance*" to the spring force, which is referred to as "**damperForce**", otherwise the spring will oscillate back and forth forever (because there is no air friction into Unity's environment to act as "external friction" force and also the spring is not made of physical material to produce "internal-friction" force). For this reason, based on Stoke's law, we are using the following equation to calculate the "**damperForce**":

$$F_i = -b * v_x \Leftrightarrow$$

Internal friction = -damping coefficient * spring velocity

This equation helps us to model the loss of energy of the system over time in the form of heat (internal friction). We also add an external friction force to the "**damperForce**" based on class notes about mass and spring simulation in the **Easy Java Simulations** environment, which has the form of:

$$F_e(t) = A * \sin(\omega t) \Leftrightarrow$$

External friction = amplitude * sine of some frequency

This equation helps us to model in some way the loss of energy because of the external friction that happens due to several factors, e.g. wind friction. As **A** we are using "**Physics.gravity.y**" which is the gravity acceleration the on y-axis (-9.81 m/s^2). We define **ω** as:

$$\omega = \sqrt{(k / m)} \Leftrightarrow$$

Oscillation frequency = √(stiffness / mass of car)

Finally, the sum of "**springForce**" and "**damperForce**" multiplied by "**transform.up**" (in order for the force to be applied upwards) gives us the total suspension force.

One thing that we have to mention is that the value of some variables such as stiffness and damping is empirical and comes up from experimentation after running the game multiple times with different values to see the results. For different types of cars, there are different types of suspension settings, e.g. the suspension of a lightweight racing car has different stiffness and damping, than that of a heavy SUV.

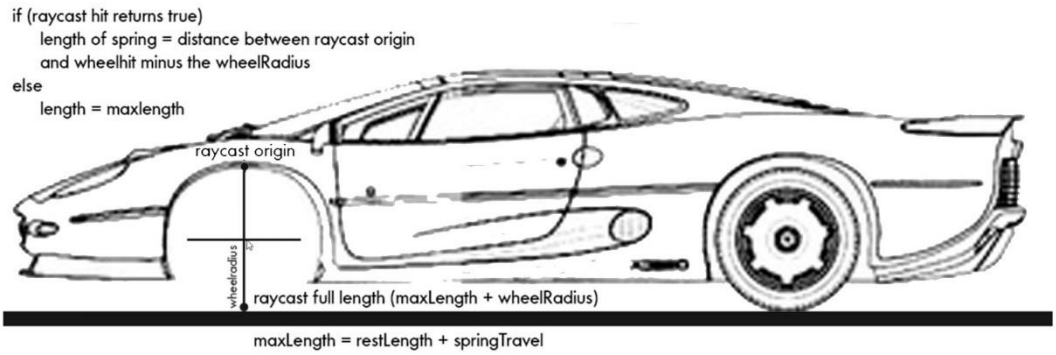


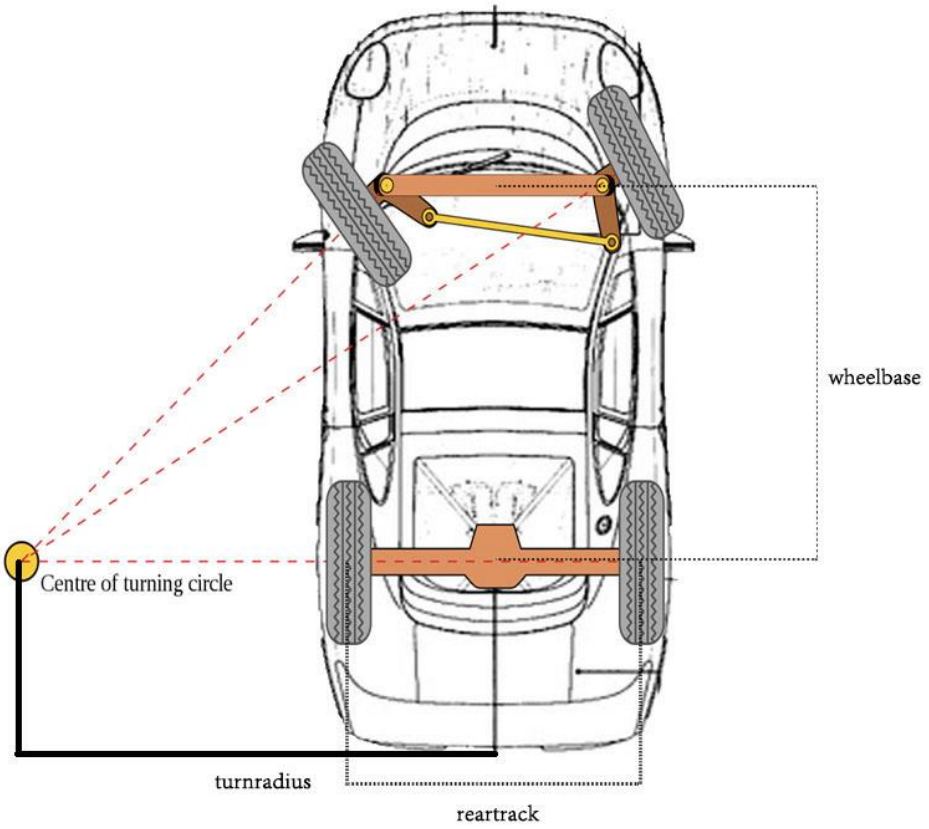
Figure1. *The suspension mechanism of the car*

The next concept is about the Ackerman-Steering mechanism of the car. When a car turns, the inner and outer wheels trace different arcs, otherwise they slip. What Ackerman's mechanism does is to rotate the inner wheel of the car during a turn with a bigger angle around its y-axis, than the outer wheel, in order for the car to be rotated around the middle point of the rear axis. This the helps car to improve its maneuverability (turning without any slip), as the angular velocities (spinning) of the inner and the outer wheels are different during turns (matching in this way with their arcs). The code in Unity that implements this mechanism is in the "**CarController.cs**" file.

```
if (steerInput > 0) // turns right
        {
            ackermannAngleLeft  = Mathf.Rad2Deg * Mathf.Atan(wheelBase /
(turnRadius + (rearTrack / 2))) * steerInput;
            ackermannAngleRight = Mathf.Rad2Deg * Mathf.Atan(wheelBase /
(turnRadius - (rearTrack / 2))) * steerInput;
        }
else if(steerInput < 0) // turns left
        {
            ackermannAngleLeft  = Mathf.Rad2Deg * Mathf.Atan(wheelBase /
(turnRadius - (rearTrack / 2))) * steerInput;
            ackermannAngleRight = Mathf.Rad2Deg * Mathf.Atan(wheelBase /
(turnRadius + (rearTrack / 2))) * steerInput;
        }
else
        {
            ackermannAngleLeft  = 0;
            ackermannAngleRight = 0;
        }
```

And the visualization of how this works in Figure 2.

If steering right
Rad2Deg * Atan (wheelBase in meter / (turnRadius in meters + (rearTrack in meters / 2 to get center)) * steerInput for left wheel
Rad2Deg * Atan (wheelBase in meter / (turnRadius in meters - (rearTrack in meters / 2 to get center)) * steerInput for right wheel
(Rad2Deg is necessary because Atan uses Radians as its output value, therefore we want to translate that to Degrees)



$$\delta_{f,in} = \tan^{-1}\left(\frac{L}{R - \frac{T}{2}}\right) \quad \delta_{f,out} = \tan^{-1}\left(\frac{L}{R + \frac{T}{2}}\right)$$

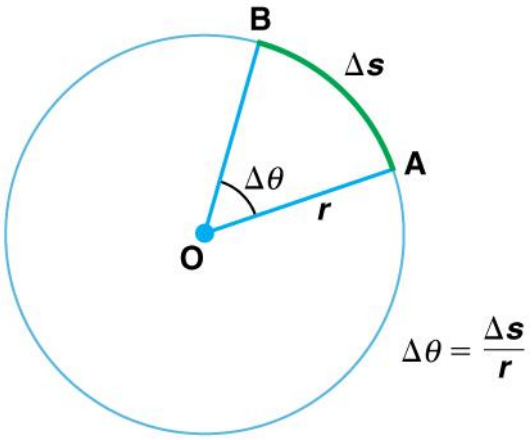
Figure2. *The Ackerman Steering mechanism and the mathematic formula of the inner and outer wheel angles. The difference between the steering angle of the front wheels and the input steer angle is known as Dynamic Toe.*

Although the formula is straightforward, if we set the real values of the car according to the manufacturer's specifications, the behavior of the car in the game may not be so pleasant to the user, as the car either turns steeply or very little, depending on its suspension and its acceleration, making its handling a challenge. A potential solution to this problem could be to use a dynamic steering system that variables depending on the suspension and acceleration settings.

The third concept is about the spinning of the car wheels. To achieve this we use some basic knowledge from rotational dynamics. We know that the rotation angle is given by:

$$\Delta\theta = \Delta s / r \Leftrightarrow$$

Angular displacement = Displacement / radius



The distance traveled in a circular path is the green arc length Δs, as shown on the left. For one complete revolution, the arc length is the circumference of a circle of radius **r**, equals **2π**, as if we integrate both sides of the equation, we have:

$$\int_0^{2\pi} \Delta\theta = \frac{1}{r}\int_0^{2\pi r} \Delta s \;\Rightarrow\; 2\pi\, rad = 1\, revolution$$

The code in Unity that implements this equation is part of the "**CarController.cs**" file:

```
        // Calculate displacement vector
        Vector3 currentPosition = transform.position;
        Vector3 displacement = currentPosition - lastPosition;

        float anglef = Mathf.Sign(Vector3.Dot(displacement,
transform.forward)) * Mathf.Rad2Deg * displacement.magnitude / 0.38f;

        float angler = Mathf.Sign(Vector3.Dot(displacement,
transform.forward)) * Mathf.Rad2Deg * displacement.magnitude / 0.41f;

        // Apply rotation to the wheel mesh
        wheels[0].Rotate(Vector3.right, anglef);
        wheels[1].Rotate(Vector3.right, anglef);
        wheels[2].Rotate(Vector3.right, angler);
        wheels[3].Rotate(Vector3.right, angler);

        // Update the last position and time for the next frame
        lastPosition = currentPosition;
```

The first thing we have to do is to calculate the displacement. This is equal to the difference in the car's position in two consecutive frames. The next step is to check the car's direction and if it goes backward. For a moving car, there are two types of forces applied to its wheels, the translational which has the same direction of the car's way, and the rotational, caused by the friction between the wheels and the ground. These two forces are of the same magnitude (when they do not slip) but in opposite directions.

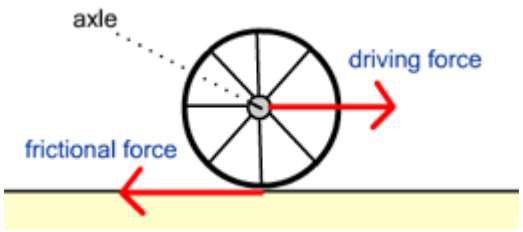


Figure 3. *The two types of forces applied on wheels*

For this reason, this step is very important, as if we use the displacement directly, then the car wheels will rotate towards the car's direction, giving an unnatural result. The direction of the car is calculated as the dot product of the displacement and the unit vector, represented as **(0, 0, ,1)**, along the z-axis of its local coordinate system. We ensure in this way that the dot product will be always positive if the car is moving forward in every direction and negative if the car reverses. So we multiply with the sign

of **Δs**, in order **Δθ** to have the correct direction before the wheels rotate. We also convert the radians to degrees, otherwise, the wheels will not rotate. And finally, one more detail is that we have different angular displacements for the front and the rear wheels, as they have different radii (front wheels smaller than rear).

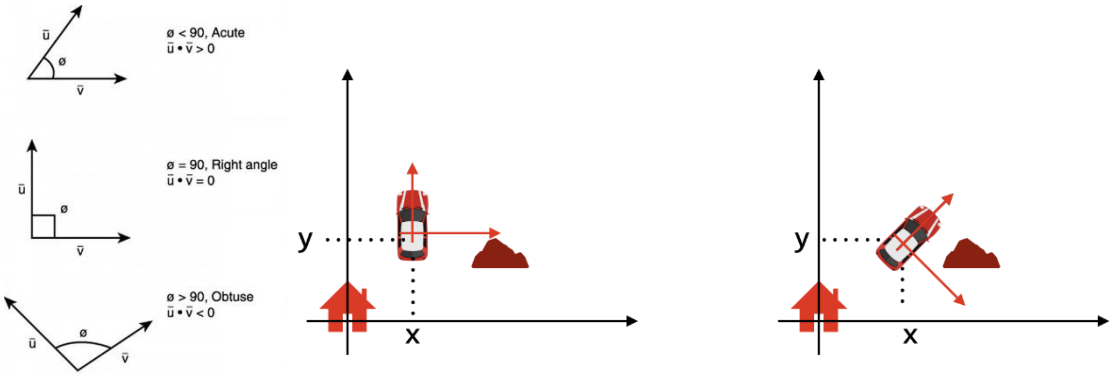


Figure 4. *Property of dot product and car's local coordinate system with red axes visualized, help us to understand better how the direction of the car is calculated according to its displacement.*

The last concept that we will see, has to do with translation matrices. Specifically, we want to use again the dot product and also create the translational matrix in order to calculate the new position of the light covers when headlights are on (because our car has popup headlights). The code that does that, is part of the "**OpenParts.cs**" file:

```
void Start()
    {
        initialPosition = lights.localPosition;
        translationMatrix = fillMatrix(translationMatrix, 0.0f, -
0.1912899f, 0.0f);
        expanded = ExpandVector3ToVector4(initialPosition);
        result = DotProduct(translationMatrix, expanded);

        final.x = result.x;
        final.y = result.y;
        final.z = result.z;
    }

public Matrix4x4 fillMatrix(Matrix4x4 translationMatrix, float x, float y,
float z)
    {
        translationMatrix[0, 0] = 1.0f;
        translationMatrix[0, 1] = 0.0f;
        translationMatrix[0, 2] = 0.0f;
        translationMatrix[0, 3] = x;

        translationMatrix[1, 0] = 0.0f;
        translationMatrix[1, 1] = 1.0f;
        translationMatrix[1, 2] = 0.0f;
        translationMatrix[1, 3] = y;

        translationMatrix[2, 0] = 0.0f;
        translationMatrix[2, 1] = 0.0f;
        translationMatrix[2, 2] = 1.0f;
        translationMatrix[2, 3] = z;

        translationMatrix[3, 0] = 0.0f;
        translationMatrix[3, 1] = 0.0f;
        translationMatrix[3, 2] = 0.0f;
```

```
        translationMatrix[3, 3] = 1.0f;

        return translationMatrix;
    }

    public Vector4 ExpandVector3ToVector4(Vector3 vector3)
    {
        return new Vector4(vector3.x, vector3.y, vector3.z, 1.0f);
    }

    public Vector4 DotProduct(Matrix4x4 matrix, Vector4 vector)
    {
        Vector4 row1 = new Vector4(matrix.m00, matrix.m01, matrix.m02,
matrix.m03);
        Vector4 row2 = new Vector4(matrix.m10, matrix.m11, matrix.m12,
matrix.m13);
        Vector4 row3 = new Vector4(matrix.m20, matrix.m12, matrix.m22,
matrix.m23);
        Vector4 row4 = new Vector4(matrix.m30, matrix.m31, matrix.m32,
matrix.m33);

        Vector4 result = new Vector4(
            Vector4.Dot(row1, vector),
            Vector4.Dot(row2, vector),
            Vector4.Dot(row3, vector),
            Vector4.Dot(row4, vector)
        );

        return result;
    }
```

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix}$$

Figure 5. *The dot product between the translational matrix and the position vector of the popup headlights of Jaguar XJ220. The result of the dot product moves the light cover downwards, revealing headlights.*

The way that the code works is to first capture the initial position of the cover (when lights are off). Then create the translational matrix with the necessary parameters. Because we want to move the cover downwards along to y-axis only, we set the y-axis parameter to a negative value (e.g.**-0.19f**) and the other parameters are zero. After that, we expand the captured position vector by adding one more element with value **1** (**neutral element**), for the multiplication to be done. Next, the result is stored in a new **Vector4,** and the new position vector after lights are on, consists of the three first elements of the **Vector4**. Of course, the whole process is only educative, to demonstrate how an object's translation happens. This can be done much easier with very few lines of code, using Unity's ready functions.

A few things about the controlling of the car, have to do mainly with its acceleration and braking forces applied to it. Sadly the lack of time and access to

detailed information about some technical specifications of the car, like its engine architecture, do not allow us to accurately simulate the acceleration and the braking forces, applied to it. Mostly these two concepts have to be done intuitively into the game, knowing the very basics of how a car works and not based on strict mathematic equations (like **Ackerman Steering**). For example, to produce an appropriate amount of force that moves the car, we first try to simulate its throttle based on the user's input. Then proportionally to the throttle and the current gear, we try to generate the right number of RPMs of the engine, for the car to accelerate and decelerate more naturally. We also try to change the gears automatically according to the current RPMs. Some challenges about this task have to do with the right ratio of increasing or decreasing RPMs, e.g. if the user does not give any input or the car crashes with an obstacle. Another thing we attempted is to somehow simulate the car's slipping, when the handbrake is pressed, by applying some torque to it.

In general, we mostly based on two very popular racing games nowadays, "Forza Motorsport 7" and "Forza Horizon 5". Using them as a reference point we try to develop our own game. The project of course is not yet completed, but is a try to understand more about real-world physics and how we can simulate it in a virtual environment, like the Unity game engine, to produce some natural-looking results.