**Проектная работа по теме**

Создание физической модели автомобиля с независимой подвеской

***Цель*:**

Получение физической модели автомобиля для дальнейших опытов и испытаний.

***Преимущества*:**

Компьютерная симуляция позволяет не создавать реальный прототип, а сделать опыты с реалистичной моделью. Это позволяет сэкономить средства, проводить изощренные тесты.

***Средства*:**

Моделирование проводилось на физическом движке “Unity 3D”.

**Содержание**

1. Введение.
2. Прототип.
3. Принцип работы.
4. Реализация.
5. Улучшение.
6. Добавление силы трения.
7. Создание утилиты для построения графиков.
8. Зависимость силы трения от наклона плоскости.
9. Переход на модель с лучшей совместимостью.
10. Настройка управления.
11. Добавление углов схождения.
12. Модернизация модели.
13. Добавление визуализации вектора ускорения.
14. Вывод.

**Введение**

Данный физический движок имеет ряд преимуществ перед другими: наличие большого количества физических элементов, широкая распространенность, относительная простота работы и то, что он бесплатный.

**Прототип**

За основу были взяты идеи независимой подвески автомобиля, т.к. она превосходит зависимую во многих ситуациях.

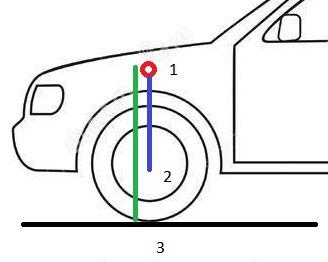
Известно, что в основе лежит пружина, которая удерживает кузов и амортизатор, который гасит колебания. Сила сопротивления амортизатора прямо пропорциональна скорости его сжатия, а силу упругости пружины пропорциональна длине её сжатия и направлена к исходному положению.

Из всех способов измерить длину пружины был выбран способ с помощью Raycast-инга. Рассматривался также и способ расчета длины на основе моделирования пружины, зная координаты её концов, но в условиях Unity он не так удобен.

**Принцип работы**

Стоит упомянуть, что вычисления проводятся 60 раз в секунду, и в каждый из этих моментов времени выводятся значения необходимых величин. Такая частота позволяет получить небольшую погрешность и высокую производительность. Точность можно увеличить, увеличив частоту обновления, но для этого необходимая бОльшая вычислительная мощность.

*Вот схема автомобиля сбоку*:



Точкой 1 обозначено место крепления амортизатора и пружины к кузову, такая конструкция используется в подвеска типа *MacPherson*. Точкой 2 обозначен уровень, на котором амортизатор крепится к рычагу, точкой 3 обозначена точка касания колесом земли. На рисунке амортизатор изображен перпендикулярно поверхности земли, но это можно рассматривать как уже проекции амортизатора на оси, поэтому он изображен перпендикулярно. Силу, которая в самой стойке можно найти, зная угол, под которым она крепится, а он уже выбирается человеком. Хотя перпендикулярное расположение стойки встречается на ЗАЗ 968.

**Реализация**

Также эта модель не учитывает расположение амортизатора на рычаге, так как в этом случае значения сил отличаются в отношении длин: от точки вращения рычага до точки крепления и длины рычага, а в расчетах учитываются уже получившиеся их отношения значения.

Из точки 1 в 3 будет испускаться луч длины, равной максимальной длины пружины плюс радиус колеса. В соответствии с этим будет вычисляться текущая длина пружины, равная разности луча и длины пружины в покое. Длина луча — расстояние от точки 1 до 3, которое не всегда равно сумме длины покоя и радиуса колеса. Зная текущую длину, можно вычислить удлинение пружины в данный момент времени: из длины покоя вычесть длину пружины.

Стоит обратить внимание, что знак удлинения имеет роль: если он положительный, то пружина сжата, если отрицательный, то пружина растянута. Это позволяет определить направление силы упругости.

Опишем это в следующем скрипте:

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class RaycastSuspension: MonoBehaviour

{

[Header("Suspension")]

public Rigidbody rb;

public float wheelRadius;

public float springElasticity;

public float damperElasticity;

public float restLength;

public float maxDelta;

private float springDelta;

private float springLength;

private float springForce;

private float damperForce;

private float suspensionForce;

private float speed;

private float prevLength;

voidStart()

{

}

void FixedUpdate()

{

if(Physics.Raycast(transform.position, -transform.up,outRaycastHit hit, restLength + wheelRadius + maxDelta))

{

springLength = hit.distance - wheelRadius;

springLength = Mathf.Clamp(springLength, springLength - maxDelta, springLength + maxDelta);

springDelta = restLength - springLength;

springForce = springElasticity \* springDelta;

speed = (springLength - prevLength) / Time.fixedDeltaTime;

damperForce = damperElasticity \* speed;

suspensionForce = springForce - damperForce;

rb.AddForceAtPosition(transform.up \* suspensionForce, hit.point);

Debug.DrawRay(transform.position, -transform.up.normalized \* springLength);

prevLength = springLength;

}

}

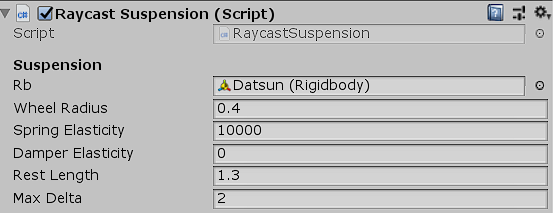
}

Теперь создадим модель автомобиля, добавим к ней 4 дочерних объекта, которые располагаются на местах крепления стоек к кузову. Добавим на эти объекты скрипты.

*В окне сцены должна появиться такая картина:*



*А на каждом из колес появится скрипт со следующими полями:*



*Вот что они значат*:

Rb– Rigidbody объект, который является автомобилем. Именно к нему и применяются все силы.

Wheel Radius –радиус колеса.

Spring Elasticity –жесткость пружины в H / m.

Damper Elasticity –жесткость амортизатора в H \* c / m.

Rest Length –длина пружины в покое.

Max Delta –максимальное изменение длины пружины.

*Вот как вычисляется сила амортизатора:*

Берется длина пружины в два соседних момента времени, находится её изменение за этот промежуток и оно делится на время этого промежутка. Так как обработка происходит в методе FixedUpdate,то этот промежуток одинаков между кадрами, и расчеты получаются логичными.

Пока что установим значение жесткости амортизаторов на каждом из колес в 0 и запустим программу.

*Результат*:



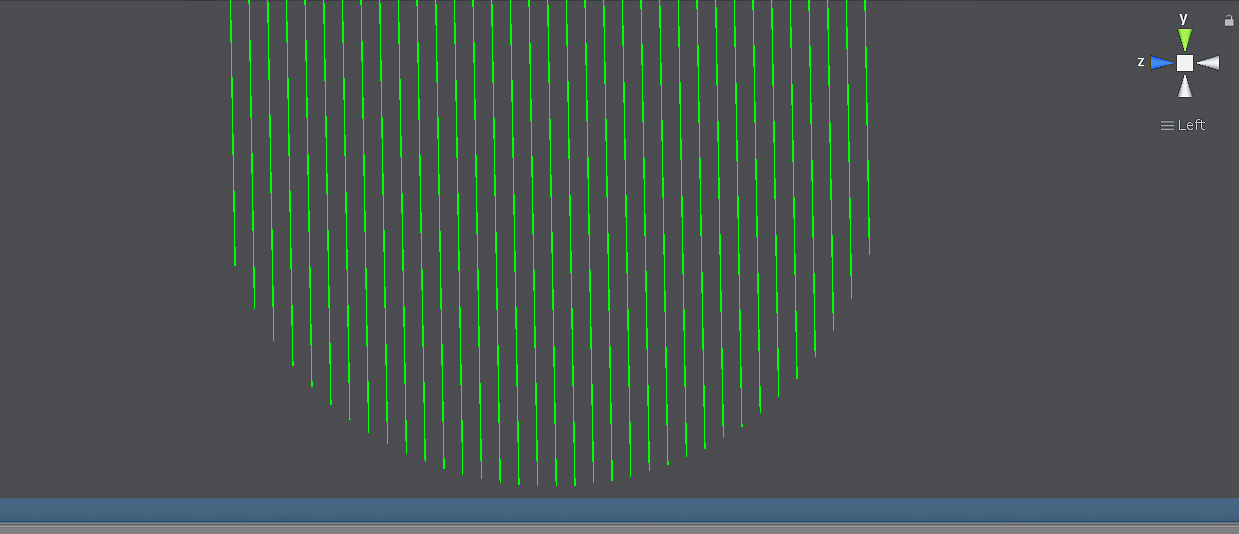
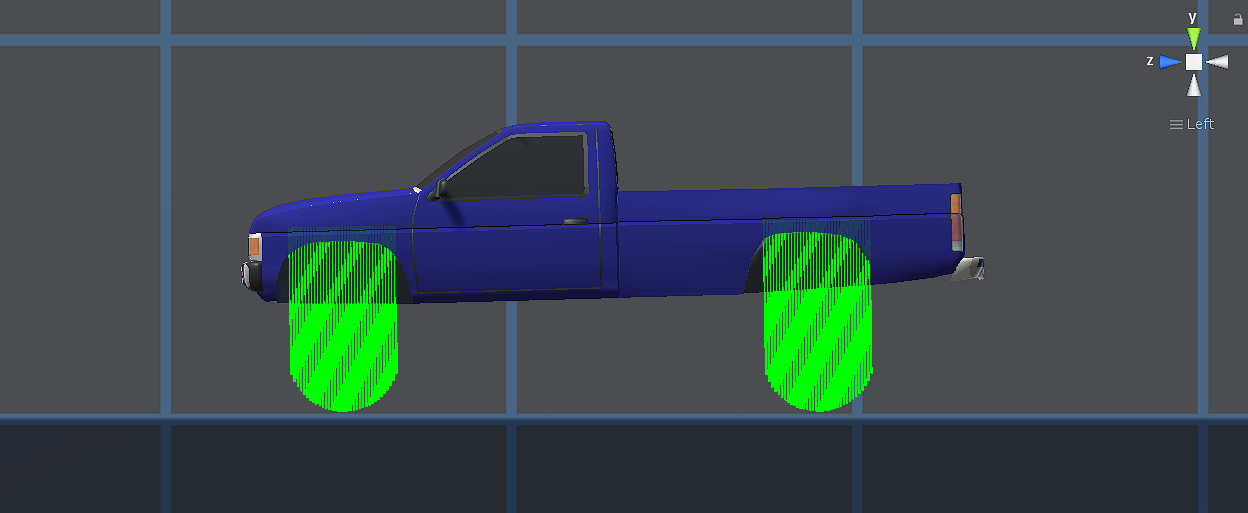
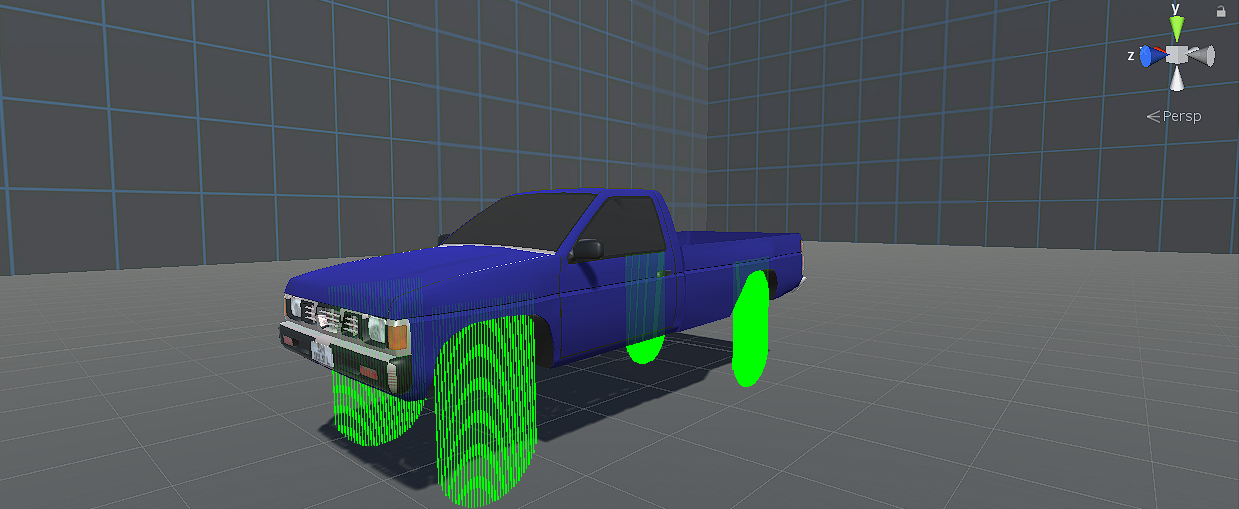
Автомобиль будет держаться в воздухе, но он будет бесконечно долго колебаться в пространстве. Это из-за отсутствия сопротивления. Установив значение на 2000, увидим, что он со временем прекращает свои колебания.

**Улучшение**

В реальности автомобиль касается колесами, в на лучом, поэтому реализуем аналогичное. Создадим массив лучей, который будет создавать что-то подобное кругу.

Для этого в момент запуска скрипта будем решать уравнение окружности, в связи с получившимися координатами испускать луч соответственной длины.

*В результате получится вот это:*



*Объяснение алгоритма:*

Для начала решается уравнение окружности в зависимости от кол-ва лучей, этот параметр задается вручную. Этот блок находится в методе Start и выполняется только 1 раз, поэтому это не сильно портит производительность. Затем полученные значения добавляются в массив, и каждый кадр происходит чтение из этого массива из испускание луча из необходимой точки нужной длины. Находится луч наименьшей длины и относительно него считаются все оставшиеся силы.

*Вот скрипт:*

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Suspension: MonoBehaviour

{

[Header("Suspension")]

public Rigidbody rb;

public float wheelRadius;

public float springElasticity;

public float damperElasticity;

public float restLength;

public float maxDelta;

private float springDelta;

private float springLength;

private float springForce;

private float damperForce;

private float suspensionForce;

private RaycastHit outHit;

private float speed;

private float prevLength;

private bool hitted;

private Vector3 hitPoint;

[Header("Anti-stuck")]

public float raysNumber;

private float[] maxLengths;

private Vector3 centerPos;

private Vector3 rayPoint;

private float stuckDelta;

private floa tmaxLength;

void Start()

{

maxLengths = new float[(int)raysNumber];

for(int i = 0; i < (int)raysNumber; i++)

{

rayPoint = transform.position;

rayPoint.z -= wheelRadius;

if(i < raysNumber / 2)

{

maxLength = Mathf.Sqrt((wheelRadius \* wheelRadius) - Mathf.Pow(wheelRadius - (i \* wheelRadius \* 2 / raysNumber), 2f));

}

else

{

maxLength = Mathf.Sqrt((wheelRadius \* wheelRadius) - Mathf.Pow(wheelRadius - ((raysNumber - i) \* 2 \* wheelRadius / raysNumber), 2f));

}

maxLengths[i] = maxLength;

}

}

void FixedUpdate()

{

springLength = restLength + maxDelta + 0.1f;

hitted = false;

N = rb.mass \* 9.8f / 4f;

for (int i = 1; i < raysNumber; i++)

{

rayPoint = transform.position;

rayPoint += transform.forward.normalized \* (i \* (wheelRadius \* 2) / raysNumber - wheelRadius);

if (Physics.Raycast(rayPoint, -transform.up ,out RaycastHit hit, restLength + maxLengths[i]))

{

hitted = true;

if (hit.distance < springLength)

{

springLength = hit.distance - wheelRadius;

hitPoint = hit.point;

Normal = hit.normal;

outHit = hit;

}

}

Debug.DrawRay(rayPoint, -transform.up \* (springLength + maxLengths[i]), Color.green);

}

if (hitted)

{

springDelta = restLength - springLength;

springForce = springElasticity \* springDelta;

speed = (springLength - prevLength) / Time.fixedDeltaTime;

damperForce = damperElasticity \* speed;

suspensionForce = springForce - damperForce;

gravityForce = Vector3.down \* rb.mass \* 9.8f / 4f;

normalForce = outHit.normal.normalized \* N;

rb.AddForceAtPosition(transform.up \* suspensionForce, hitPoint);

prevLength = springLength;

}

}

}

Данная реализация не подразумевает деформацию шины колеса, поэтому условно принимается, что колесо всегда заезжает на препятствие. Поэтому тестировать необходимо на гладких поверхностях.

**Добавление силы трения**

При запуске программы, если наклонить поверхность, на которой стоит автомобиль, то автомобиль начнет двигаться в направлении наклона. Это из-за отсутствия сил трения. Приступим к их реализации.

Как известно, сила трения покоя направлена против равнодействующей силы. Но, к сожалению, Unity не предоставляет нам её, поэтому выведем её сами. В данном случае она будет равна векторный сумме силы тяжести, силы реакции опоры, силы упругости пружины и силы сопротивления амортизатора. Перпендикуляр к поверхности найдем с помощью того самого короткого луча, выбранного ранее. Его поле normal возвращает вектор, перпендикулярный к поверхности, которой он коснулся. Метод normalized возвращает вектор, совпадающий по направлению, но имеющий длину 1. Таким образом, умножив данный вектор на значение силы реакции опоры, которое в данные случае равно 1/4 силы тяжести автомобиля, так как 4 колеса. Вектор силы тяжести можно получить, умножив Vector3.down на вес автомобиля на данном колесе. Силы подвески уже известны, и остается найти только векторную сумму, сложив эти переменные.

Сделаем коэффициент трения покоя публичной переменной, значение которой можно будет изменять. Таким образом, длина вектора силы трения покоя будет равна произведению длины вектора реакции опоры на коэффициент, а направлена она будет против равнодействующей силы (именно для этого мы её и находили). Как только равнодействующая силы окажется больше силы трения покоя, тело начнет движение, а трение покоя сменится трением скольжения, а до того момента сила трения будет равна равнодействующей силе.

*После изменений скрипт будет выглядеть так:*

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class LocalSuspension : MonoBehaviour

{

[Header("Local suspension")]

public Rigidbody rb;

public float wheelRadius;

public float springElasticity;

public float damperElasticity;

public float restLength;

public float maxDelta;

private float springDelta;

private float springLength;

private float springForce;

private float damperForce;

private float suspensionForce;

private RaycastHit outHit;

private float speed;

private float prevLength;

private bool hitted;

private Vector3 hitPoint;

[Header("Anti-stuck")]

public float raysNumber;

private float[] maxLengths;

private Vector3 centerPos;

private Vector3 rayPoint;

private float stuckDelta;

private float maxLength;

[Header("Friction system")]

public float staticFriction;

public float dynamicFriction;

private Vector3 Normal;

private float frictionForce;

private Vector3 accelFriction;

private float N;

private Vector3 resDir;

private Vector3 speedDir;

private Vector3 frictionDirection;

private Vector3 gravityForce;

private Vector3 normalForce;

private Vector3 resForce;

void Start()

{

maxLengths = new float[(int)raysNumber];

for (int i = 0; i < (int)raysNumber; i++)

{

rayPoint = transform.position;

rayPoint.z -= wheelRadius;

if (i < raysNumber / 2)

{

maxLength = Mathf.Sqrt((wheelRadius \* wheelRadius) - Mathf.Pow(wheelRadius - (i \* wheelRadius \* 2 / raysNumber), 2f));

}

else

{

maxLength = Mathf.Sqrt((wheelRadius \* wheelRadius) - Mathf.Pow(wheelRadius - ((raysNumber - i) \* 2 \* wheelRadius / raysNumber), 2f));

}

maxLengths[i] = maxLength;

}

}

void FixedUpdate()

{

springLength = restLength + maxDelta + 0.1f;

hitted = false;

N = rb.mass \* 9.8f / 4f;

for (int i = 1; i < raysNumber; i++)

{

rayPoint = transform.position;

rayPoint += transform.forward.normalized \* (i \* (wheelRadius \* 2) / raysNumber - wheelRadius);

if (Physics.Raycast(rayPoint, -transform.up, out RaycastHit hit, restLength + maxLengths[i]))

{

hitted = true;

if (hit.distance < springLength)

{

springLength = hit.distance - wheelRadius;

hitPoint = hit.point;

Normal = hit.normal;

outHit = hit;

}

}

Debug.DrawRay(rayPoint, -transform.up \* (springLength + maxLengths[i]), Color.green);

}

if (hitted)

{

springDelta = restLength - springLength;

springForce = springElasticity \* springDelta;

speed = (springLength - prevLength) / Time.fixedDeltaTime;

damperForce = damperElasticity \* speed;

suspensionForce = springForce - damperForce;

gravityForce = Vector3.down \* rb.mass \* 9.8f / 4f;

normalForce = outHit.normal.normalized \* N;

resForce = gravityForce + normalForce + transform.up \* suspensionForce;

frictionDirection = -resForce;

frictionForce = N \* staticFriction;

if (resForce.magnitude > frictionForce)

{

frictionForce = N \* dynamicFriction;

frictionDirection = -rb.velocity.normalized \* frictionForce;

rb.AddForceAtPosition(frictionDirection, transform.position);

}

else

{

rb.AddForceAtPosition(frictionDirection, transform.position);

}

rb.AddForceAtPosition(transform.up \* suspensionForce, hitPoint);

prevLength = springLength;

}

}

}

**Создание утилиты для построения графиков**

Теперь можно построить различные графики зависимости: силы трения от равнодействующей, скорости от силы трения и другие. К сожалению, Unity не предоставляет стандартных средств построения графиков, поэтому напишем его с нуля.

Графики будут строиться в реальном времени, поверх происходящего в программе. Для этого напишем скрипт.

Так как для каждого графика необходимо проводить вычисления отдельно, то удобнее создать отдельный класс с нужными методами, а потом обращаться к ним.

Для инициализации объекта этого класса необходим спрайт, который будет отрисовываться и объект с компонентом RectTransfrom, именно на нем и происходит отрисовка.

*Таким образом выглядит скрипт:*

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class GraphRenderer : MonoBehaviour

{

private Sprite circleSprite;

private RectTransform graphContainer;

private int sepCount = 0;

private float width;

private float height;

public GraphRenderer(Sprite circleSprite, RectTransform graphContainer)

{

this.circleSprite = circleSprite;

this.graphContainer = graphContainer;

this.width = graphContainer.sizeDelta.x;

this.height = graphContainer.sizeDelta.y;

}

public GameObject CreateCircle(Vector2 pos, bool flag)

{

float x = pos.x;

float y = pos.y;

GameObject node = new GameObject("node", typeof(Image));

node.transform.SetParent(graphContainer, false);

node.GetComponent<Image>().sprite = circleSprite;

RectTransform rectTransform = node.GetComponent<RectTransform>();

rectTransform.anchoredPosition = pos;

rectTransform.sizeDelta = new Vector2(11, 11);

rectTransform.anchorMin = new Vector2(0, 0);

rectTransform.anchorMax = new Vector2(0, 0);

if (flag)

{

return node;

}

else

{

return null;

}

}

public void ShowGraph(List<Vector2> l)

{

for (int i = 0; i < l.Count; i++)

{

CreateCircle(l[i], false);

}

}

private Vector2 posX;

private Vector2 posY;

public void CreateSeparator(Vector2 pos, Vector2 info)

{

posX.y = -10f;

posX.x = pos.x;

posY.x = -10;

posY.y = pos.y;

GameObject labelX = new GameObject("labelX");

labelX.transform.SetParent(graphContainer, false);

labelX.AddComponent<RectTransform>();

RectTransform xTransform = labelX.GetComponent<RectTransform>();

xTransform.anchorMin = Vector2.zero;

xTransform.anchorMax = Vector2.zero;

labelX.AddComponent<Text>();

Text labelXText = labelX.GetComponent<Text>();

labelXText.text = info.x.ToString();

labelXText.font = Resources.GetBuiltinResource(typeof(Font), "Arial.ttf") as Font;

labelXText.horizontalOverflow = HorizontalWrapMode.Overflow;

labelXText.verticalOverflow = VerticalWrapMode.Overflow;

labelXText.alignment = TextAnchor.MiddleCenter;

xTransform.sizeDelta = new Vector2(11f, 11f);

xTransform.anchoredPosition = posX;

GameObject labelY = new GameObject("labelY");

labelY.transform.SetParent(graphContainer, false);

labelY.AddComponent<RectTransform>();

RectTransform yTransform = labelY.GetComponent<RectTransform>();

yTransform.anchorMin = Vector2.zero;

yTransform.anchorMax = Vector2.zero;

labelY.AddComponent<Text>();

Text labelYText = labelY.GetComponent<Text>();

labelYText.text = info.y.ToString();

labelYText.font = Resources.GetBuiltinResource(typeof(Font), "Arial.ttf") as Font;

labelYText.horizontalOverflow = HorizontalWrapMode.Overflow;

labelYText.verticalOverflow = VerticalWrapMode.Overflow;

labelYText.alignment = TextAnchor.MiddleCenter;

yTransform.sizeDelta = new Vector2(5f, 5f);

yTransform.anchoredPosition = posY;

sepCount += 1;

}

}

Метод *CreateCirle* принимает два параметра: координаты точки, которую надо отобразить и булевое значение о том, надо ли возвращать созданный объект.

Метод *CreateSeparator* создает разделители на осях координат, они нужны для удобного просмотра графиков. Он принимает координаты точки на экране и текст, который должен быть отображен на разделителе.

Теперь для проверки будем измерять скорость автомобиля каждую секунду при свободном падении.

*Вот скрипт (но метод CreateSeparator еще не доработан, поэтому он не принимает параметров и значение выводятся неудобным способом):*

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class AccelerationDiagram: MonoBehaviour

{

privateGraphRenderer gr;

private float nextActionTime;

public Rigidbody rb;

public Sprite circleSprite;

publicRectTransform rectTransform;

void Start()

{

gr = new GraphRenderer(circleSprite, rectTransform);

nextActionTime = 0f;

}

private float scale =2f;

void FixedUpdate()

{

if(Time.time > nextActionTime)

{

gr.CreateCircle(new Vector2(nextActionTime \*10\* scale, rb.velocity.magnitude \* scale),false);

gr.CreateSeparator(new Vector2(nextActionTime \*10\* scale, rb.velocity.magnitude \* scale));

nextActionTime +=1f;

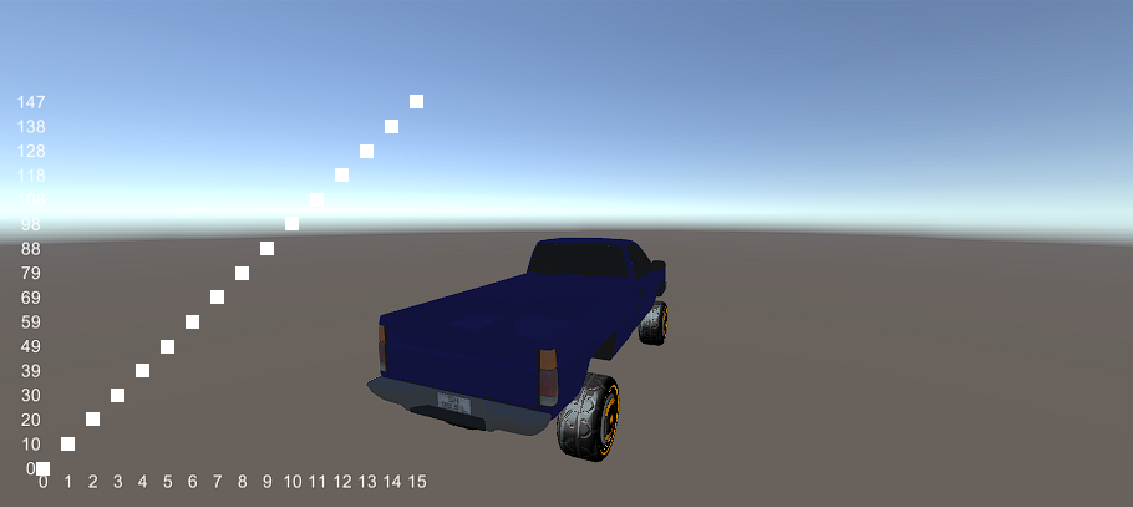
}

}

}

В дальнейшем будет использоваться аналогичные скрипты, но с другими вычислениями. Добавим камеру дочерним объектом к автомобилю и отправим его в свободное падение.

*В результате будет такая картина:*



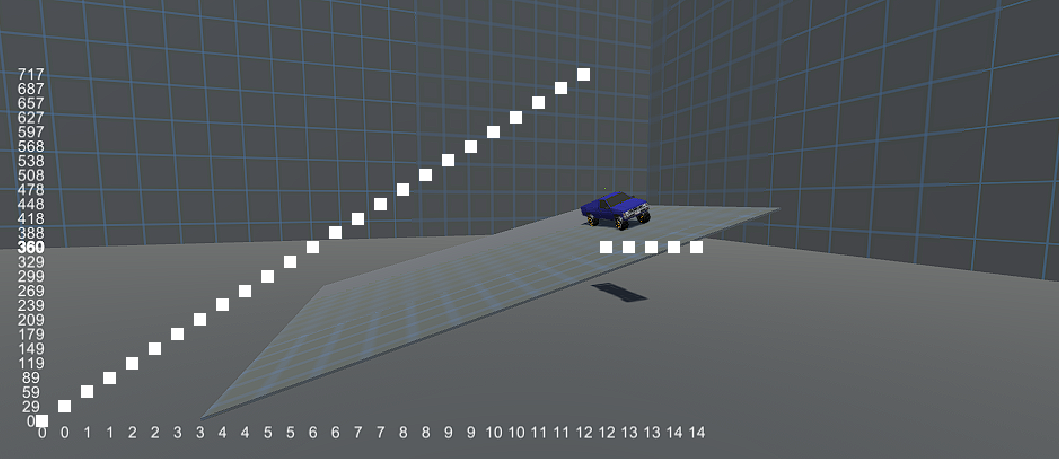
Это график зависимости скорости от времени, по горизонтальной оси отмечены секунды, а по вертикальной скорость в метрах в секунду. Из этого графика можно сделать вывод, что он графики строятся правильно и можно строить что-то другое.

Например, график зависимости силы трения от угла поверхности (автомобиль находится на наклонной плоскости и изменяется угол наклона).

**Зависимость силы трения от наклона плоскости**

Пусть скрипт наклоняет плоскость на 1 градус в секунду. Будем отмечать угол на горизонтальной оси. На вертикальной будет значение силы трения, сопротивляющейся равнодействующей. Стоит отметить, что автомобиль будет стоять поперек наклонной плоскость, так как именно в другом случае будет оказывать свое влияние силы трения качения, которая лучше реализована в Wheel Collider -ах.

*Вот получившийся график*:



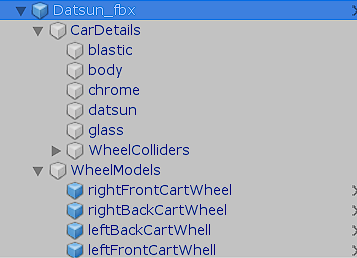
Из него видно, что пока наклон плоскости был меньше 12 градусов, сила трения покоя была равна равнодействующей, и автомобиль оставался в покое. Как только наклон увеличился, сила трения стала трением покоя и тело начало движение.

**Переход на модель с лучшей совместимостью**

Теперь, реализовав основные механики (чтобы анализировать их значение), которые скрыты в силу особенностей Unity, имея возможность строить графики, можно приступить к реализации модели на Wheel Collider -ах, которая в условия этого движка работает эффективнее и оптимизированнее.

Для этого создадим объект автомобиля, добавим к нему пустой дочерний объект, в котором и будут коллайдеры, создадим аналогичные для частей кузова автомобиля и моделей колес. Дочерними к коллайдерам создадим 4 пустых объекта, на которые и добавим нужный компонент. Аналогично поступим и с моделями.

*В итоге в окне сцены должна быть такая картина:*



**Настройка управления**

Приступим к написанию скрипта для настройки коллайдеров. Для начала, сделаем пикап полноприводным. Для этого необходима тяга на всех 4 колесах (так как проще всего сделать все 3 дифференциала заблокированными, но легко изменить коэффициент блокировки, так как известны скорости каждого из колес). Если необходимо, чтобы тяга на колеса не передавалась, то нужно установить значение параметра torque на 0 на каждом из нужных колес.

Чтобы можно было удобно управлять тягой, сделаем управление ей с помощью стрелок клавиатуры (но для действительно точных расчетов нужно задавать параметры программно).

using UnityEngine;

using System.Collections;

public class CarController:MonoBehaviour

{

public Rigidbody rb;

public GameObject model;

private WheelCollider wh;

public float torque;

private Quaternion rotation;

private Vector3 position;

void Drive(float accel)

{

accel = Mathf.Clamp(accel,-1,1);

if(accel > 0)

{

wh.motorTorque = torque \* accel;

}

else

{

wh.brakeTorque = torque \* (-accel);

}

wh.GetWorldPose(outposition,outrotation);

model.transform.position = position;

model.transform.rotation = rotation;

}

float accel;

void Start()

{

wh = GetComponent<WheelCollider>();

}

void Update()

{

accel = Input.GetAxis("Vertical");

Drive(accel);

}

}

**Добавление углов схождения**

Метод Drive и передает тягу на колеса, в зависимости от нажатой клавиши. Параметр torque задается человеком, он влияет на величину ускорения автомобиля. Если переменная accel отрицательная, то автомобиль начнет замедляться до остановки.

Запустив программу, увидим это, но автомобиль не может изменить направление своего движения, поэтому добавим его.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class WheelSteering: MonoBehaviour

{

public float TOE;

public float left;//1 - left, 0 - right

private WheelCollider wc;

private floatside;

void Turn(float side,float left)

{

side = Mathf.Clamp(side,-1,1);

wc.steerAngle =45f\* side;

if(left ==1)

{

wc.steerAngle += TOE;

}

else

{

wc.steerAngle -= TOE;

}

}

void Start()

{

wc = GetComponent<WheelCollider>();

}

void Update()

{

side = Input.GetAxis("Horizontal");

Turn(side, left);

}

}

Поле ТОЕ - угол схождения колес в градусах, а значение left означает, левое колесо или правое, это нужно для определения углов.

*Вот несколько примеров с разными значениями угла схождения:*

*0 градусов:*



*25 градусов:*



*-25 градусов:*



С учетом данной физ. модели это - единственный угол, который можно описать, а его изменение одно из самых значительных, напрямую влияющее на устойчивость автомобиля. Углы развала нельзя изменить из-за того, что модель колеса принимается как очень твердое тело (хоть и противоречит идеям силы трения качения), поэтому при наклоне колеса, пятно контакта существенно уменьшается, из-за чего тесты становятся необъективными. Разработка упругого колеса требует отдельного внимания.

**Модернизация модели**

Модель Unity использует всего один луч для проверки столкновений, поэтому для езды требуется идеально ровная поверхность, а иногда могут возникать такие неприятности:



Для избежания этого напишем скрипт.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

[RequireComponent(typeof(WheelCollider))]

public class AntiStuck:MonoBehaviour

{

public Transform wheelPosition;

public int raysNumber;

public float raysMaxAngle;

private WheelCollider wh;

private float originRadius;

private Vector3 rayDirection;

void Start()

{

wh = GetComponent<WheelCollider>();

originRadius = wh.radius;

}

void Update()

{

float radius Offset = 0f;

for(int i = 0; i <= raysNumber; i++)

{

rayDirection = Quaternion.AngleAxis(wh.steerAngle, transform.up) \* Quaternion.AngleAxis(i \* (raysMaxAngle / raysNumber), transform.right) \* transform.forward;

if(Physics.Raycast(wheelPosition.position, rayDirection.normalized,out RaycastHit hit, wh.radius))

{

radiusOffset = Mathf.Max(radiusOffset, wh.radius - hit.distance);

}

Debug.DrawRay(wheelPosition.position, rayDirection.normalized \* originRadius, Color.blue);

}

wh.radius = Mathf.LerpUnclamped(wh.radius, originRadius + radiusOffset, Time.deltaTime \*10f);

}

}

*После этого в аналогичных ситуациях будет следующий результат:*



**Визуализация вектора ускорения**

Еще одной интересной функцией я посчитал визуализацию вектора ускорения каждый момент времени. Для этого будем брать два момента времени, находить разность значений скорости в эти моменты и делить разность на промежуток времени между этими моментами.

Получив вектор ускорения, добавим его как вторую координату к компоненту Line Renderer объекта Line, который ранее должен быть добавлен к модели (красным цветом обозначено начало вектора, а желтым - конец).

*Опишем в следующем скрипте:*

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class AccelerationVector : MonoBehaviour

{

public Rigidbody rb;

public LineRenderer line;

public float updateTime;

private float nextActionTime;

private Vector3 curSpeed;

private Vector3 prevSpeed;

private Vector3 accel;

void Start()

{

curSpeed = rb.velocity;

}

void FixedUpdate()

{

if (Time.time > nextActionTime)

{

curSpeed = rb.velocity;

accel = (curSpeed - prevSpeed) / updateTime;

line.SetPosition(1, accel);

prevSpeed = curSpeed;

nextActionTime += updateTime;

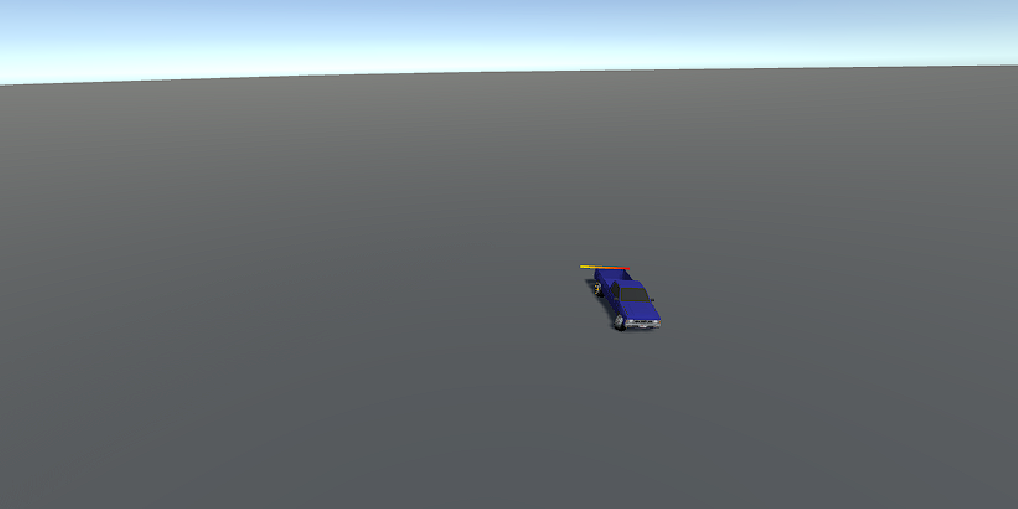
}

}

}

Но, проделывая данную операцию каждые 0.02 секунды, мы получим непригодный для восприятия результат: луч будет постоянной дергаться из стороны в сторону, поэтому добавим параметр, отвечающий за частоту обновления, одно из самых удобных значений - 0.25 секунды.

Теперь попробуем двигаться с постоянной скоростью по кругу, результатом будет следующее:



Как видим, вектор ускорения направлен к центру импровизированной окружности, т.е. это - центростремительное ускорение.

**Вывод**

Была сделана физическая модель автомобиля, продемонстрированы различные явления. С помощью этой модели можно проводить различные испытания новшеств в транспорте, экономя на затратах по воспроизведению реальных экземпляров. Например, анализ поведения машины при разных значениях жесткости амортизаторов и пружин, расположением кузова, материалом шин.

P.S. в примерах кода нет комментариев, так как алгоритмы работы были объяснены в тексте.

***Источники***:

1. https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%B2%D0%B5%D1%81%D0%BA%D0%B0\_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%BE%D0%B1%D0%B8%D0%BB%D1%8F
2. https://ru.wikipedia.org/wiki/%D0%90%D0%BC%D0%BE%D1%80%D1%82%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80
3. https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D1%81%D1%81%D0%B8
4. https://docs.unity3d.com/ru/current/Manual/UnityManual.html