REALTIME CAR DESTRUCTION IN UNITY

Harish N

ABSTRACT

Most games that involve you driving around in a car needs a destruction system. It can be simple as swapping out the models like in old GTA games, or it can involve using soft body physics to accurately simulate and create dents based on the forces acting on the car surface.

In this thesis I aim meet at the middle and create a system that has the ability to create dents depending on the collision like in soft body simulations while still staying performant and keeping it relatively simple.

CONTENTS

1. Introduction

A Car destruction can be split into 3 parts the

i) Mesh Deformation
ii) Parts detachment
iii) Audio

These three elements come together to create a believable destruction system for the cars.

The first part involves some vertex displacement to create dents upon collision.

The second part involves either detaching or shattering the object (in case of glass) if the part takes enough damage.

The sound effects are created upon every collision and a sound is played depending on the force of the collision.

This thesis only covers the car destruction part, but my car's wheel physics script and GitHub repo is included at the appendix.

2. Literature Review

This research was started because of a reddit post by studio tatsu where they showcased their real-time mesh destruction system

Studio tatsu

Unfortunately, they never describe how it is done or even have a project or demo download.

Next I made my first version of my destruction system with this,

Youtube tutorial about mesh deformation

But it was limiting in the fact it only supported lowpoly models with only one mesh.

Hence I started making one on my own and this is the result.

3. Methodology

3.1 Mesh Deformation

As mentions before we will be editing the Mesh data directly to visualize the vehicle damage. We first check the impulse of the collision. If it above a certain threshold we will continue to the next step. We loop through all the contact points.

Now for the actual mesh deformation. The car is made up of multiple parts hence we will need to keep track of all of them in a list or array. Now we loop through all the vertices. But we will be adding a minimum distance condition to prevent any unnecessary processing since a mesh may have a lot of vertices. Now since we have all the vertices within range we will be moving them the opposite direction of the collision normal, the distance it moves is depending on the distance from the contact point. We will be using the following formula for the displacement:

$$VertexPosition = VertexPosition + \{collision\ normal * (maxRadius - currentVertexDistance) * falloffMultiplier * \frac{(collisionForce - minDamge)}{(maxDamage - minDamage)}\}$$

$$\frac{(collisionForce - minDamge)}{(maxDamage - minDamage)}$$

This line gives the strength of the collision between 0 and 1.

After this we will recalculate the normal and bounds of the mesh to fix lighting and culling.

3.2 Parts detachment

Each detachable part has a script attached that tracks its health and decides when it should break off. During the loop when we go through each mesh, we affect the health depending upon the collision force. Once the health reaches 0, we enable physics and collision for that part. We use the following equation to calculate damage:

$$health = health - \frac{(collisionForce - minDamge)}{(maxDamage - minDamage)} * \text{maxDamage}$$

One unique case is with glasses where instead of detaching the part after collision we play a particle system that displays the glass shattering. Other than that, the rest of the process is the same.

3.3 Audio

Audio part is really simple, we just play audio upon each collision with a randomized pitch and audio level depending upon the impact of collision.

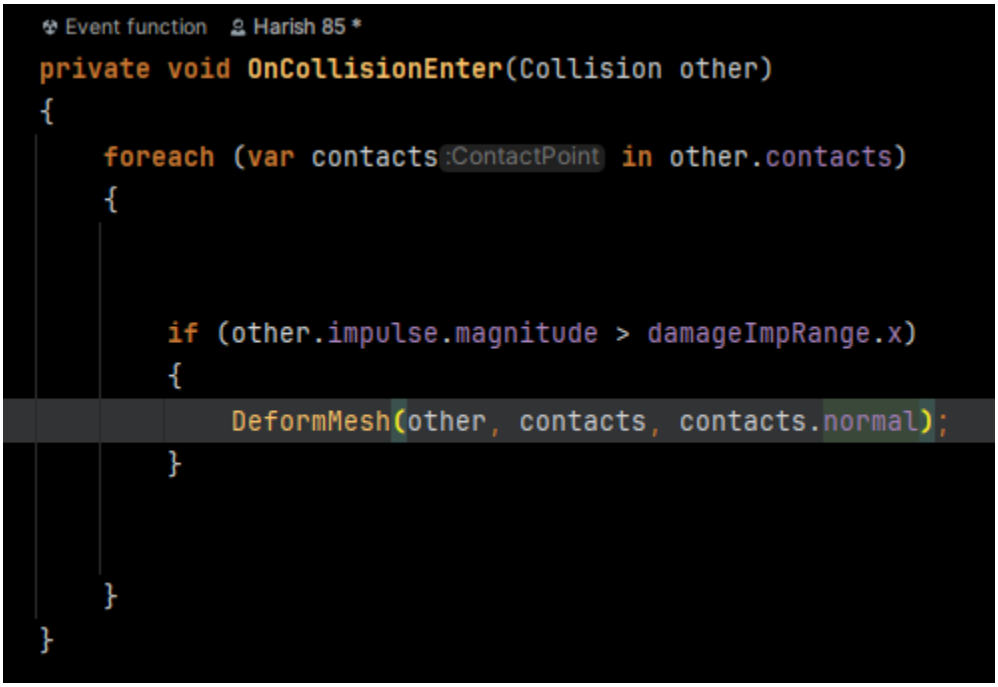
$$\frac{(collisionForce - minDamge)}{(maxDamage - minDamage)}$$

We use the same formula to calculate the volume of the audio.

4. Development Process

4.1 Mesh Deformation

In OnCollisionEnter we loop through all the contact points and then if the impulse is greater than the min damage we call the DeformMesh function



Now in the DeformMesh function we are going to loop through all the mesh filters and the through each vertex of the mesh. We convert the vertex position to world space and then we check if the vertex is within the range.

```
private void DeformMesh(Collision other, ContactPoint contacts, Vector3 normal)
{
    foreach (var m:MeshFilter in meshFilters)
    {
        Vector3[] vertices = m.mesh.vertices;

        for (int i = 0; i < vertices.Length; i++)
        {
            Vector3 worldPoint = m.transform.TransformPoint(vertices[i]);

            float dist = Vector3.Distance(a:worldPoint, b:contacts.point);

            //move the vertices within range in the direction of the normal
            if (dist < damageRadius)
            {
                // Move the vertices
            }
        }
    }
}
```

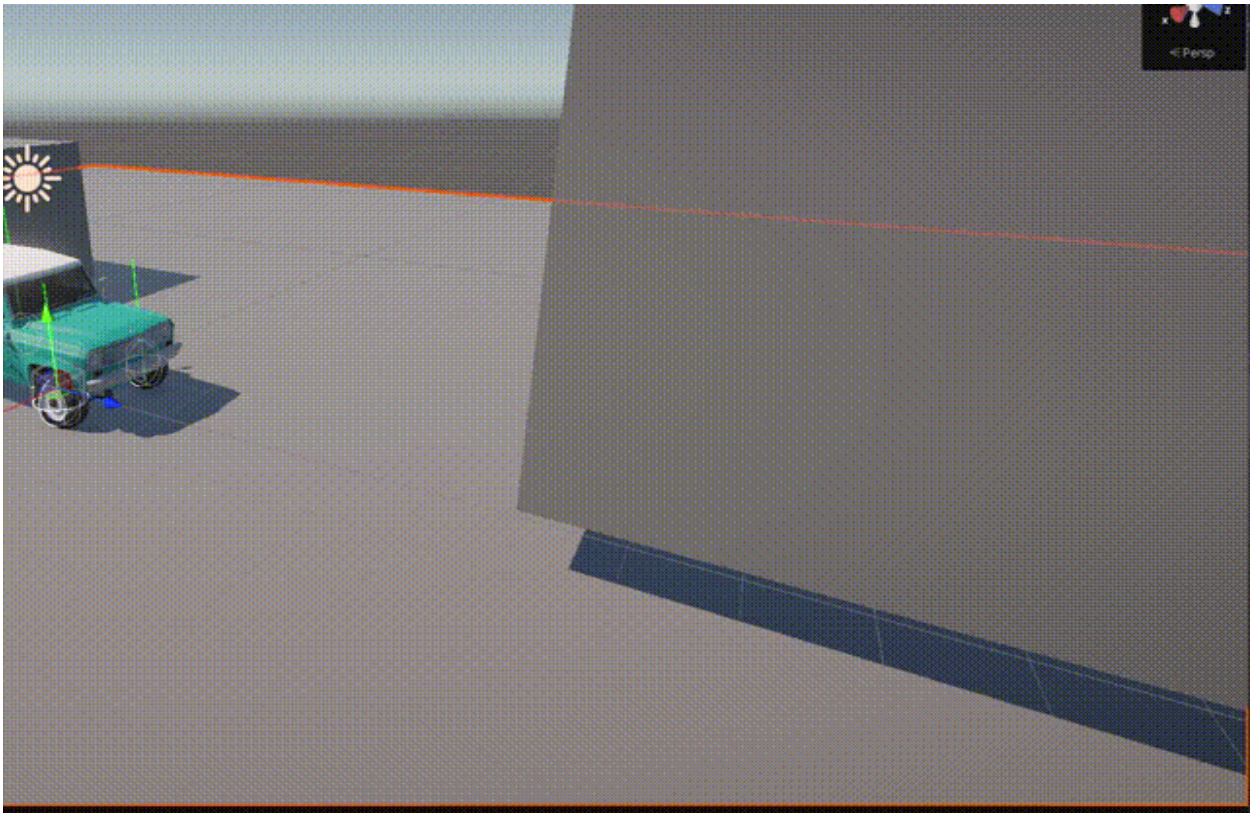
Now we use the formula described in methodology to move the vertex in the direction of the normal using the following lines of code.

```
vertices[i] += m.transform.InverseTransformDirection(normal) *
               (damageRadius - dist) * fallOffMul *
               Mathf.Clamp01((other.impulse.magnitude - damageImpRange.x) /
                             (damageImpRange.y - damageImpRange.x));
```

Finally, we assign the vertices to the mesh and then recalculate the normal and bounds.

```
m.mesh.vertices = vertices;
m.mesh.RecalculateNormals();
m.mesh.RecalculateBounds();
```

Now we can test out the deformation



As you can see the car looks pretty beaten up. I removed the base of the car from being deformed since I wanted the car to still look drivable.

The front glass is sticking up since we don't have a detachment system yet, so we will start working on it now.

4.2 Parts detachment

First, we create a script called BreakableCarPart which will be attacked to the object that has the mesh. The script is very simple, it has a float for health and a Boolean to check if it is broken. It also has a OnBreak function that is called once the health goes below 0 where we enable its collider and set it's isKinematic of the rigid body to false.

We could use its mesh collider but instead I created a mesh that wraps around the car to use it as a mesh as a mesh collider. So only one collision exists and the parts use primitives for colliders.

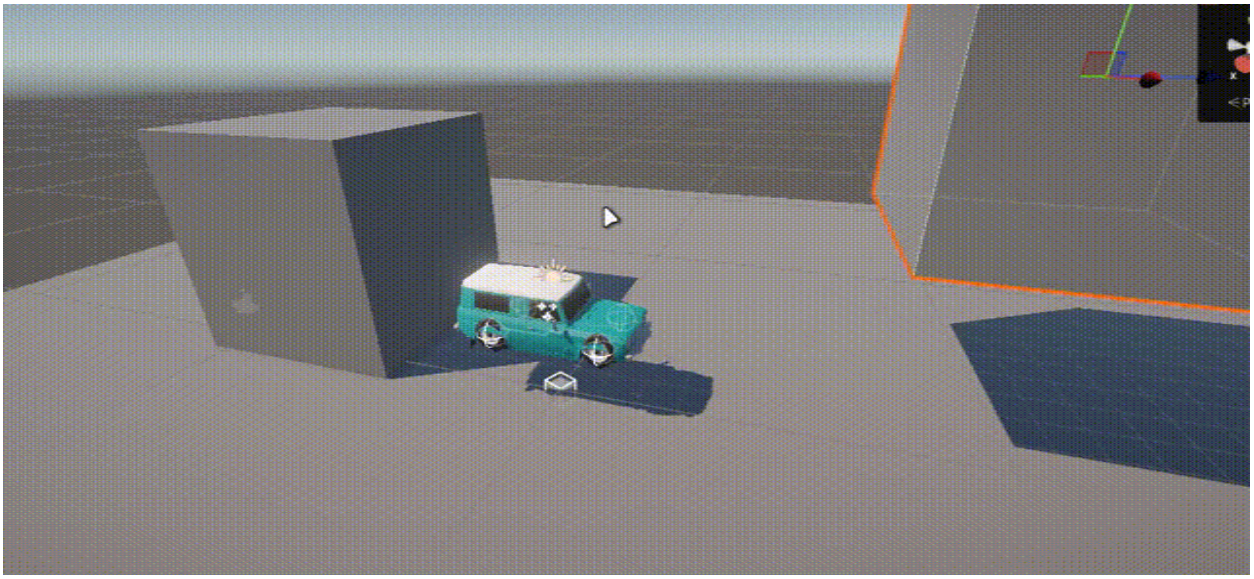
Here is the script:

```
public class BreakableCarPart : MonoBehaviour
{
    public float health;  Changed in 1+ assets
    private bool isBroken = false;

    Event function  Harish 85
    void Update()
    {
        if (isBroken)
            return;

        if(health <= 0)
        {
            OnBreak();
            isBroken = true;
        }
    }
    Frequently called  1 usage  1 override  Harish 85
    public virtual void OnBreak()
    {
        if (TryGetComponent(out Rigidbody rb))
        {
            rb.isKinematic = false;
            GetComponent<Collider>().enabled = true;
        }
    }
}
```

For the glass we use a inherit from the BreakableCarPart. We override the OnBreak function and then play a particle system and disable the mesh

```
public class GlassBreakable : BreakableCarPart
{
    public ParticleSystem glassParticle;  Changed in 1 asset

    Frequently called  0+1 usages  Harish 85 *
    public override void OnBreak()
    {
        isBroken = true;
        glassParticle.Play();
        GetComponent<MeshRenderer>().enabled = false;
    }
}
```

When we put this all together, we get the following outcome.



4.3 Audio

This is the easiest part out of all since we already have all the underlying code almost ready. We just choose a random collision sound track from a list and then play it upon collision with a volume depending on the impact force.

Here is its code:

```
var audioSource = impactSounds[UnityEngine.Random.Range(0, impactSounds.Count)];
audioSource.pitch = UnityEngine.Random.Range(.8f, 1.2f);
audioSource.volume = Mathf.Clamp01((other.impulse.magnitude - damageImpRange.x) /
                                   (damageImpRange.y - damageImpRange.x)) + .3f;
audioSource.Play();
```

I am adding .3f to the volume to ensure the volume isn't too low since for this condition to occur we need to already be over the minimum impact.

You can find the whole script in the appendix.

5. Results and Optimization:

5.1 Mesh vertices optimization

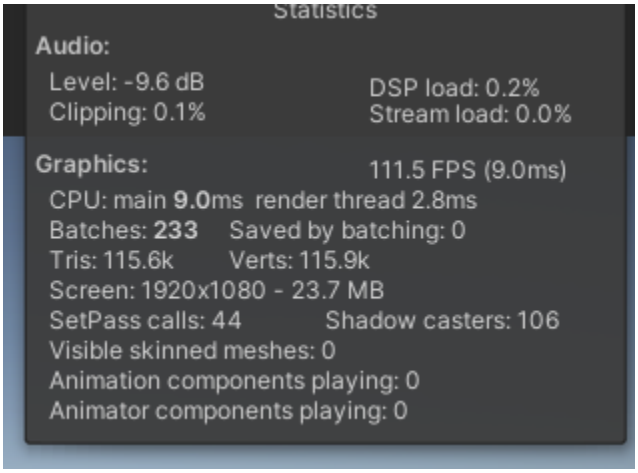


Figure 1 (While Idle)

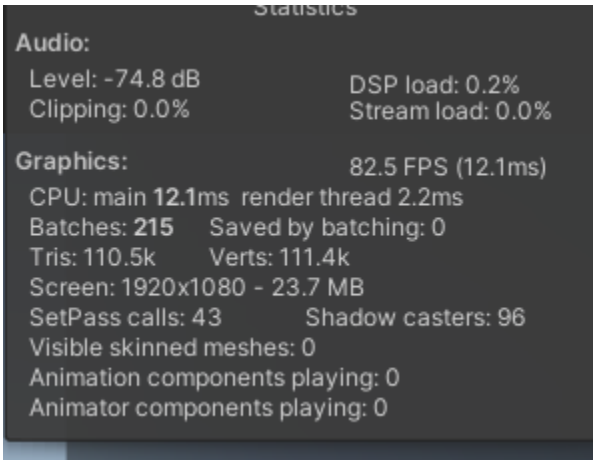


Figure 2 (During Collision)

Currently I am getting an FPS of around 100 - 120 fps in my laptop. But during collisions with high impact, we get around 70 - 80 fps. That's a 20-fps drop. So, let's fix it.

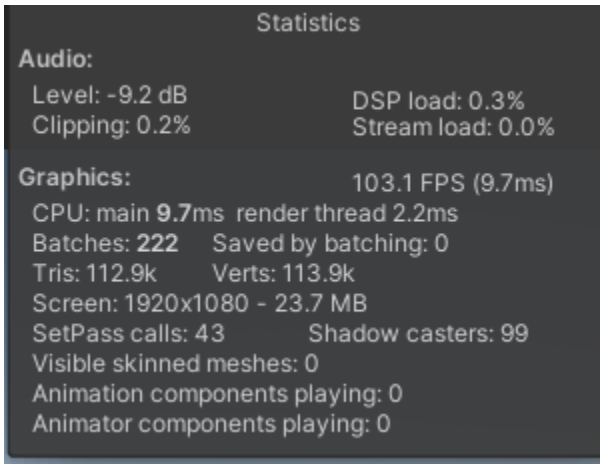
You might have noticed we are getting the vertices form the mesh every time a collision occurs while we can just cache it once and update it during collision. So we should really be caching it.

So I am going to be changing the code to cache the vertices in the start function. I am also going to be using the newer GetVertices and SetVertices function in mesh. Here is the updated code.

```
void Start()
{
    verticesList = new List<List<Vector3>>();
    int i = 0;
    foreach (var m:MeshFilter in meshFilters)
    {
        verticesList.Add(item:new List<Vector3>());
        m.mesh.GetVertices(verticesList[i]);
        i++;
    }
}
```

```
List<Vector3> vertices = verticesList[m];

for (int i = 0; i < vertices.Count; i++)
{
    Vector3 worldPoint = meshFilters[m].transform.TransformPoint(vertices[i]);
    //Debug.DrawRay(worldPoint, Vector3.up, Color.green, 10f);
    float dist = Vector3.Distance(a:worldPoint, b:contacts.point);
    //Debug.Log(dist);
    //move the vertices within range in the direction of the normal
    if (dist < damageRadius)
    {

        if (!didDamage && meshFilters[m].TryGetComponent(out BreakableCarPart br ))
        {
            didDamage = true;
            br.health -=
                Mathf.Clamp01(other.impulse.magnitude - damageImpRange.x / (damageImpRange.y - damageImpRange.x)) * maxDamage;
        }
        //adjust for distance and fall off and impulse and impulse range
        vertices[i] += meshFilters[m].transform.InverseTransformDirection(normal) *
                       (damageRadius - dist) * fallOffMul *
                       Mathf.Clamp01((other.impulse.magnitude - damageImpRange.x) /
                                     (damageImpRange.y - damageImpRange.x));
    }

}
meshFilters[m].mesh.SetVertices(vertices);
//meshCollider.sharedMesh = m.mesh;
meshFilters[m].mesh.RecalculateNormals();
meshFilters[m].mesh.RecalculateBounds();
```

This in itself improved the frame rate during collision to be consistently above 90.



During collision after Update.

5.2 Unity Jobs Optimization

Unity jobs lets us make use of multiple CPU threads , thus by offloading the vertex deformation to multiple threads we can drastically improve fps. Here is the code updated to use jobs

```
    private void DeformMesh(Collision other, ContactPoint contacts, Vector3 normal)
    {
        var audioSource = impactSounds[UnityEngine.Random.Range(0,
impactSounds.Count)];
        audioSource.pitch = UnityEngine.Random.Range(.8f, 1.2f);
        audioSource.volume = Mathf.Clamp01((other.impulse.magnitude -
damageImpRange.x) /
                                           (damageImpRange.y - damageImpRange.x)) +
.3f;
        audioSource.Play();


        for(int m = 0; m < meshFilters.Count; m++)
        {
           NativeArray<int> didDamage = new NativeArray<int>(1, Allocator.TempJob);

               NativeArray<Vector3> v = new
NativeArray<Vector3>(verticesList[m].ToArray(), Allocator.TempJob);

                DeformJob job = new DeformJob()
                {
                    impulse = other.impulse.magnitude,
                    contacts = contacts,
                    normal =
meshFilters[m].transform.InverseTransformDirection(normal),
                    vertices = v.Reinterpret<float3>(),
                    damageRadius = damageRadius,
                    localToWorldMatrix = meshFilters[m].transform.localToWorldMatrix,
                    didDamage = didDamage,
                    damageImpRange = damageImpRange,
                    fallOffMul = fallOffMul
                };
                JobHandle jobHandle= job.Schedule(verticesList[m].Count,64);
                jobHandle.Complete();
                job.vertices.Reinterpret<Vector3>().CopyTo(v);


                if (job.didDamage[0] == 1)
                {
```

```
                    if (meshFilters[m].TryGetComponent(out BreakableCarPart br))
                    {

                        br.health -=
                            Mathf.Clamp01(other.impulse.magnitude - damageImpRange.x /
(damageImpRange.y - damageImpRange.x)) *
                            maxDamage;
                    }
                }

                verticesList[m] = new List<Vector3>(v.ToArray());
            meshFilters[m].mesh.SetVertices(v);
            //meshCollider.sharedMesh = m.mesh;
            meshFilters[m].mesh.RecalculateNormals();
            meshFilters[m].mesh.RecalculateBounds();

        }
    }



[BurstCompile]
struct DeformJob : IJobParallelFor
{
    public float impulse;
    public ContactPoint contacts;
    public float3 normal;
    public NativeArray<float3> vertices;
    public float damageRadius;
    public Matrix4x4 localToWorldMatrix;
    [NativeDisableParallelForRestriction]
    public NativeArray<int> didDamage;
    public float2 damageImpRange;
    public float fallOffMul;

    public void Execute(int index)
    {
        //Vector3 worldPoint = meshFilters[m].transform.TransformPoint(vertices[i]);
        Vector3 worldPoint = localToWorldMatrix.MultiplyPoint(vertices[index]);
        //Debug.DrawRay(worldPoint, Vector3.up, Color.green, 10f);
        float dist = Vector3.Distance(worldPoint, contacts.point);
        //Debug.Log(dist);
        //move the vertices within range in the direction of the normal
        if (dist < damageRadius)
        {
            didDamage[0] = 1;
            //adjust for distance and fall off and impulse and impulse range
            vertices[index] += (normal) *
                            (damageRadius - dist) * fallOffMul *
                            Mathf.Clamp01((impulse - damageImpRange.x) /
                                          (damageImpRange.y - damageImpRange.x));
        }

    }
}
```

By doing this we are receiving an average of 120 fps with very minimal impact on collisions.

Hence final Editor results:

Idle Fps: 120 – 140

Idle Frame Time: 7.3 ms

Fps during collision: 120-130 fps

Frame Time during collision: around 7.6 ms

The Fps is Build reaches over 200 frames per second.

6. Discussion

But our current solution is not realistically simulating the collision damage. It is currently tailored in way that even after severe collisions the general structure of the car remains the same. It is made this way because we can't generate collision shapes dynamically in unity. Also, for my need the car should remain drivable despite however damaged it is.

There is still a lot that can be done here to improve, we can use soft bodies to deform the mesh thus being physically realistic.

If you are looking into creating realistic destruction Studio Tatsu has created a soft body system for car mesh destruction and have some amazing results.

7. Conclusion

Thus, we have created a semi realistic car destruction system with support for detachable parts through the help of basic mesh vertex manipulation.

This can be used in a large range of games from vehicular combat games like blur and demolition derby, open-world games, racing game or any game that features car driving.

Thus, this helps to enhance the immersion in the driving part of any game while also giving the player feedback for their driving.

8. References

Mesh deformation tutorial by Catlike Coding

Unity official Mesh documentation

Unity forum post about vertex access optimization

Arcade Car physics tutorial

Raycast based car physics

9. Appendices

Github Repository

9.1 Scripts

Destruction Script

```
using System;
using System.Collections;
using System.Collections.Generic;
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using Unity.Mathematics;
using UnityEngine;

public class DestructionV3Job : MonoBehaviour
{
    [SerializeField] private List<MeshFilter> meshFilters;
    [SerializeField] private float damageRadius = .4f;
    [SerializeField] private float minCollisionImpulse = 10000f;
    [SerializeField] private float fallOffMul = 1f;
    [SerializeField] private Vector2 damageImpRange = new Vector2(15000, 20000);
    [SerializeField] private float maxDamage = 10;

    [SerializeField] private float damageDisp = .5f;
    [SerializeField] private Rigidbody rb;
    [SerializeField] private MeshCollider meshCollider;

    [SerializeField] private List<AudioSource> impactSounds;

    List<List<Vector3>> verticesList = new List<List<Vector3>>();

    void Start()
    {
        verticesList = new List<List<Vector3>>();
        int i = 0;
        foreach (var m in meshFilters)
        {
            verticesList.Add(new List<Vector3>());
            m.mesh.GetVertices(verticesList[i]);
            i++;
        }
    }

    // Update is called once per frame
    void Update()
    {

    }

    private void OnCollisionEnter(Collision other)
    {
        foreach (var contacts in other.contacts)
        {
```

```
            if (other.impulse.magnitude > damageImpRange.x)
            {
                DeformMesh(other, contacts, contacts.normal);
            }


        }
    }

    private void DeformMesh(Collision other, ContactPoint contacts, Vector3 normal)
    {
        var audioSource = impactSounds[UnityEngine.Random.Range(0, impactSounds.Count)];
        audioSource.pitch = UnityEngine.Random.Range(.8f, 1.2f);
        audioSource.volume = Mathf.Clamp01((other.impulse.magnitude - damageImpRange.x) /
                                           (damageImpRange.y - damageImpRange.x)) + .3f;
        audioSource.Play();


        for(int m = 0; m < meshFilters.Count; m++)
        {
           NativeArray<int> didDamage = new NativeArray<int>(1, Allocator.TempJob);

               NativeArray<Vector3> v = new NativeArray<Vector3>(verticesList[m].ToArray(),
Allocator.TempJob);

                DeformJob job = new DeformJob()
                {
                    impulse = other.impulse.magnitude,
                    contacts = contacts,
                    normal = meshFilters[m].transform.InverseTransformDirection(normal),
                    vertices = v.Reinterpret<float3>(),
                    damageRadius = damageRadius,
                    localToWorldMatrix = meshFilters[m].transform.localToWorldMatrix,
                    didDamage = didDamage,
                    damageImpRange = damageImpRange,
                    fallOffMul = fallOffMul
                };
                JobHandle jobHandle= job.Schedule(verticesList[m].Count,64);
                jobHandle.Complete();
                job.vertices.Reinterpret<Vector3>().CopyTo(v);


                if (job.didDamage[0] == 1)
                {
                    if (meshFilters[m].TryGetComponent(out BreakableCarPart br))
                    {

                        br.health -=
                            Mathf.Clamp01(other.impulse.magnitude - damageImpRange.x /
(damageImpRange.y - damageImpRange.x)) *
                            maxDamage;
                    }
                }

                verticesList[m] = new List<Vector3>(v.ToArray());
            meshFilters[m].mesh.SetVertices(v);
            //meshCollider.sharedMesh = m.mesh;
            meshFilters[m].mesh.RecalculateNormals();
            meshFilters[m].mesh.RecalculateBounds();

        }
    }


}

[BurstCompile]
struct DeformJob : IJobParallelFor
{
    public float impulse;
    public ContactPoint contacts;
```

```
    public float3 normal;
    public NativeArray<float3> vertices;
    public float damageRadius;
    public Matrix4x4 localToWorldMatrix;
    [NativeDisableParallelForRestriction]
    public NativeArray<int> didDamage;
    public float2 damageImpRange;
    public float fallOffMul;

    public void Execute(int index)
    {
        //Vector3 worldPoint = meshFilters[m].transform.TransformPoint(vertices[i]);
        Vector3 worldPoint = localToWorldMatrix.MultiplyPoint(vertices[index]);
        //Debug.DrawRay(worldPoint, Vector3.up, Color.green, 10f);
        float dist = Vector3.Distance(worldPoint, contacts.point);
        //Debug.Log(dist);
        //move the vertices within range in the direction of the normal
        if (dist < damageRadius)
        {
            didDamage[0] = 1;
            //adjust for distance and fall off and impulse and impulse range
            vertices[index] += (normal) *
                               (damageRadius - dist) * fallOffMul *
                               Mathf.Clamp01((impulse - damageImpRange.x) /
                                             (damageImpRange.y - damageImpRange.x));
        }

    }
}
```

Breakable Car Part:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BreakableCarPart : MonoBehaviour
{
    public float health;
    protected bool isBroken = false;

    void Update()
    {
        if (isBroken)
            return;

        if(health <= 0)
        {
            OnBreak();
            isBroken = true;
        }
    }
    public virtual void OnBreak()
    {
        if (TryGetComponent(out Rigidbody rb))
        {
            rb.isKinematic = false;
            GetComponent<Collider>().enabled = true;
        }
    }
}
```

Glass Breakable:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class GlassBreakable : BreakableCarPart
{
    public ParticleSystem glassParticle;

    public override void OnBreak()
    {
        isBroken = true;
        glassParticle.Play();
        GetComponent<MeshRenderer>().enabled = false;
    }
}
```

Wheel:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Input = UnityEngine.Windows.Input;

public class Wheel : MonoBehaviour
{
    private Rigidbody rb;
    [SerializeField] private GameObject wheelMesh;

    [Header("Suspension")]
    public float restLength;
    public float springTravel;
    public float stiffness;
    public float damperStiffness;


    private float minLength;
    private float maxLength;
    private float lastLength;
    private float springLength;
    private float springForce;
    private float springVelocity;
    private float damperForce;

    [Header("Acceleration and Deceleration")]
    public float acceleration;

    public float friction = .1f;
    private float input;

    public float brakeFriciton = 1;

    [Header("Wheel")]
    public float wheelRadius;



    [Header("Steering")] public float maxSteerAngle;
    public float antiSkid = 0.5f;

    // Start is called before the first frame update
    void Start()
    {
        rb = transform.parent.GetComponent<Rigidbody>();
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        //move this to start
        minLength = restLength - springTravel;
```

```
        maxLength = restLength + springTravel;
        WheelPhysics();

    }
    Vector3 prevVelocity;


    private void WheelPhysics(){
        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 = stiffness * (restLength - springLength);

            damperForce = damperStiffness * springVelocity;

            Vector3 suspensionForce = transform.up * (springForce + damperForce);

             //calc forwarrd force based on input
            Vector3 forwardForce = transform.forward * input * acceleration;
            Vector3 vel = (rb.GetPointVelocity(hit.point));
            //friction force
            Vector3 forwardFriction;
            Vector3 rightForce;
            rightForce = transform.right * Vector3.Dot(vel,transform.right) ;
            Vector3 antiSkidForce = -rightForce * antiSkid ;
            if (brakeInput > 0.5)
            {
                forwardFriction = -transform.forward * Vector3.Dot(vel, transform.forward) *
brakeFriciton;

            }
            else
            {
                forwardFriction = -transform.forward * Vector3.Dot(vel,transform.forward) * friction ;
            }
            //anti skid force




            Debug.DrawLine(hit.point,hit.point + suspensionForce/rb.mass,Color.green);
            Debug.DrawLine(hit.point,hit.point + forwardForce,Color.red);
            Debug.DrawLine(hit.point,hit.point + antiSkidForce,Color.blue);
            Debug.DrawLine(hit.point,hit.point + forwardFriction,Color.yellow);

            rb.AddForceAtPosition(suspensionForce,hit.point);
            rb.AddForceAtPosition(forwardForce + antiSkidForce +
forwardFriction,hit.point,ForceMode.Acceleration);

            prevVelocity = vel;
        }
        UpdateWheelMesh(springLength);
    }

    public void SetDriveInput(float driveInput){
       input = driveInput;
    }

    public void SetSteering(float steerInput)
    {
        transform.localEulerAngles = new Vector3(0, maxSteerAngle * steerInput, 0);
    }

    float brakeInput;
    public void SetBrake(float brakeInput)
    {
        this.brakeInput = brakeInput;
    }

    private void UpdateWheelMesh(float wheelHeight)
    {
```

```
        wheelMesh.transform.position = transform.position - transform.up * wheelHeight;

        //rotate wheel mesh
        int dir = Vector3.Dot(transform.forward,rb.velocity) > 0 ? 1 : -1;
        wheelMesh.transform.Rotate(Vector3.forward,rb.velocity.magnitude *dir* Time.deltaTime * 360 /
(2 * Mathf.PI * wheelRadius));

    }

    private void OnDrawGizmos()
    {
        Gizmos.DrawLine(transform.position,transform.position - transform.up * (springLength +
wheelRadius));
        Gizmos.DrawWireSphere(transform.position - (springLength*transform.up),wheelRadius);
        Gizmos.DrawLine(transform.position,transform.position + transform.forward * input * 2);
    }
}
```

Car Controller:

```
using System;
using System.Collections;
using System.Collections.Generic;

using UnityEngine;


public class CarController : MonoBehaviour
{
    public Wheel[] frontWheels;
    public Wheel[] rearWheels;


    // Start is called before the first frame update
    void Start()
    {


    }

    private void Update()
    {
        OnForwardBackward(Input.GetAxis("Vertical"));
        //OnBrake(Input.GetAxis("Vertical") * -1);
        OnSteer(Input.GetAxis("Horizontal"));
    }

    private void OnBrake(float arg0)
        {
            foreach (var w in rearWheels)
            {
                if (arg0 > 0)
                {
                    w.SetBrake(1);
                }
                else
                {
                    w.SetBrake(0);
                }
            }
        }



        public void OnForwardBackward(float value)
        {
            foreach (var w in rearWheels)
```

```
            {
                w.SetDriveInput(value);
//              Debug.Log(value);
            }
        }



        public void OnSteer(float value)
        {
            foreach (var w in frontWheels)
            {
                w.SetSteering(value);
            }
        }


    }
```

9.2 Inspector Values

Destruction Script

Wheel Script: