Smekens Robin

Real Time Car Damage

Graduation work 2018-19

Digital Arts and Entertainment

Howest.be

CONTENTS

1. ABSTRACT

The aim of this paper is the research and implementation of different techniques for real time car damage. Trying out techniques like morph targets, skeletal meshes, procedural meshes, material vertex displacement. Going over the weak points and strong points of each technique. While also showing different aspects of real time car damage such as breaking glass/mirrors. Breaking hinges of things like doors, car hoods, car trunk when to great of a force is applied.

2. INTRODUCTION

Real time car damage has been done countless times before. Almost every racing game will have some sort of car damaging system. 3D Real time car damage is observable in games like GTA, Dirt Rally. Some games like Wreckfest and BeamNG have their main focus on realistic car damage. So a good way to simulate real time car damage is something sought after.

The aim of this paper is to research different ways to make realistic permanent dents in a mesh. Techniques like Morph Targets, Skeletal Meshes, Procedural mesh Component, Material Vertex Displacement. Describe and test situations on where each technique might be a good option to use.

How can car suspension be made so it interacts correctly with physics forces/impacts to make it behave realistically? The car should bounce when dropped from a height and should wobble when rolling over little bumps. Disconnecting parts of a car like hood, doors, trunk when hinges are to damaged or too much of a force is applied on certain car parts. As well as researching different techniques of breaking glass/mirrors via Materials, Particles, Destructible Meshes.

Researching how to save positional data & impact forces that can be used in a material for vertex displacement to create small dents in a mesh.

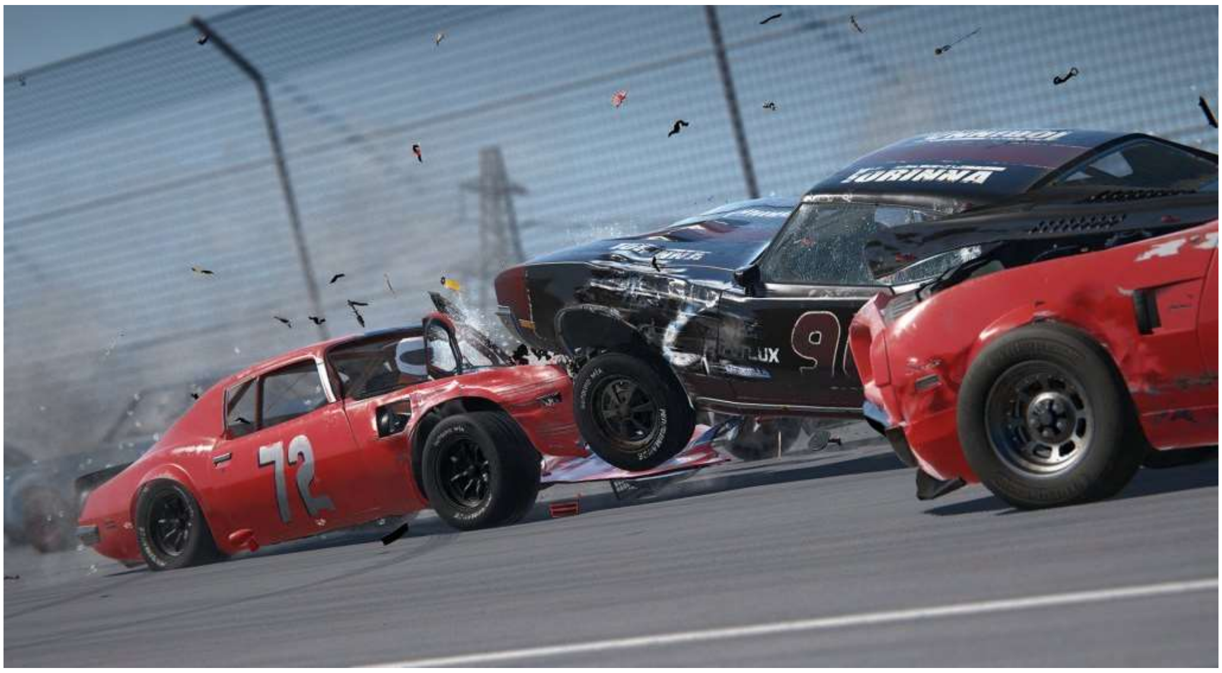
3. RESEARCH

3.1 DEFORMATION IN GAMES

3.1.1 WRECKFEST

Wreckfest (2018) is a racing video game which as a cross between FlatOut and Destruction Derby (1995) which is the first notable vehicular combat racing video game. Wreckfest takes car damage to a completely different level though. They use soft-body damage modelling which enables location-based damage that affects the driving dynamics in a realistic fashion. The damage in Wreckfest is realistic in the beginning but after a lot of collisions the deformation starts to become unrealistic. An example of this can be seen in figure 1.

Figure 1



Realistic frontal collision in Wreckfest

3.1.2 GRAND THEFT AUTO

Even though GTA isn’t primarily a racing game, vehicle damage is a fundamental aspect in the GTA series. Being able to destroy and damage vehicles has been featured in every GTA game. Cars can receive damage from sources like crashes or gunfire which provides the player with realism and challenge. In figure 2 you can see the car deformation in GTA 3 and in figure 3 you can see an example of car deformation in GTA 4.

Figure 2

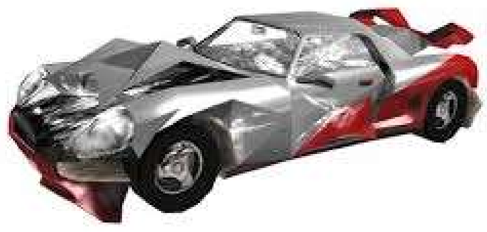
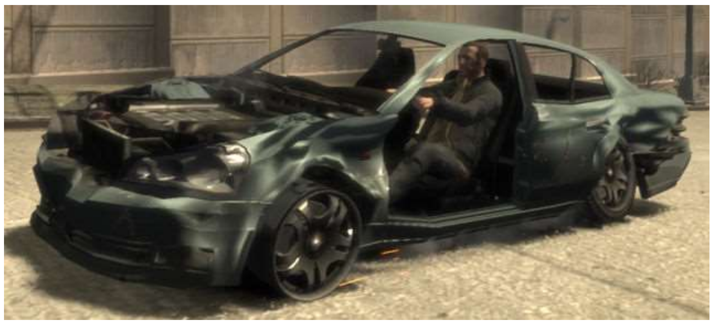

GTA III Morph Targets

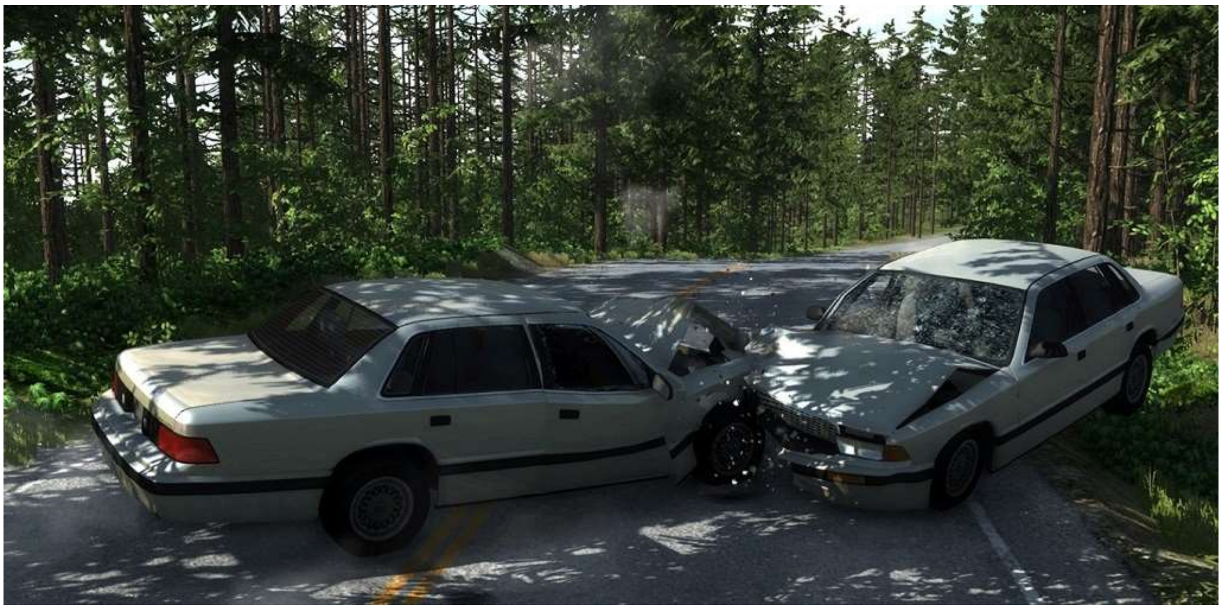
Figure 3


GTA IV dynamic deformation

3.1.3 BEAMNG.DRIVE

BeamNG.drive is a vehicle simulation video game that uses soft-body physics to simulate its vehicles. Compared to GTA and Wreckfest, BeamNG focuses mainly on real time deformation of cars. The game itself is a sandbox where you can drive certain parkours and try to finish the race with the extremely damaged car. . An example of this can be seen in figure 4.

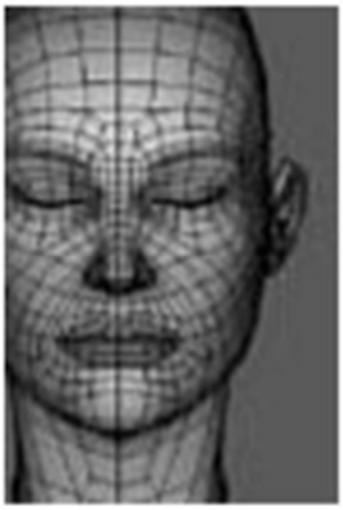
Figure 4



3.2 TECHNIQUES

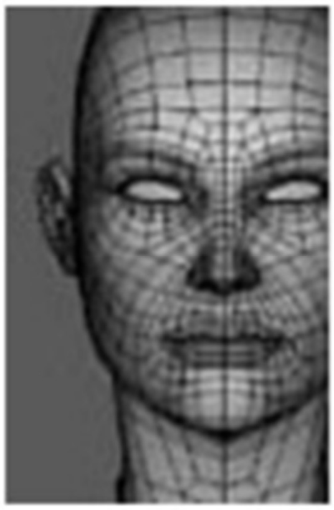
3.2.1 MORPH TARGETS/BLEND SHAPES

Widely used animation method that is mostly used for facial animation. Applied by many commercial animation packages like 3Ds Max, Maya and Blender. It involves creating a distorted version of an object and saving it as a morph target. Then interpolating between these distorted version based on a interpolation parameter. It’s a time-consuming method because you have to model each version that you want to be able to blend too. The blending itself is usually done using linear interpolation and this is not the most realistic way of blending between morph targets for realistic facial animations. Other interpolation techniques were designed for higher quality animations. Such as bilinear interpolation, pairwise interpolation or n-dimensional interpolation. [1]



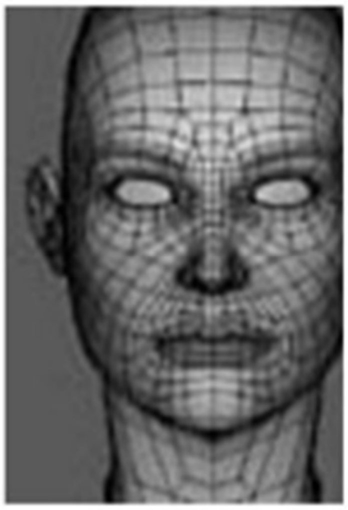
Interpolate Value = 0.0

Closed Eyes



Interpolate Value = 0.5

Interpolated Image



Interpolate Value = 1.0

Open Eyes

Even though the most important use case for morph targets is facial animation. It can be applied to car damage as well. You create different damaged versions of car parts then via colliders define hit boxes. Let the hit boxes calculate the interpolation value for the morph targets when they receive a hit. Although the damage would look realistic. It would always return the same result because all the deformations are pre-calculated.

3.2.2 SKELETAL MESHES

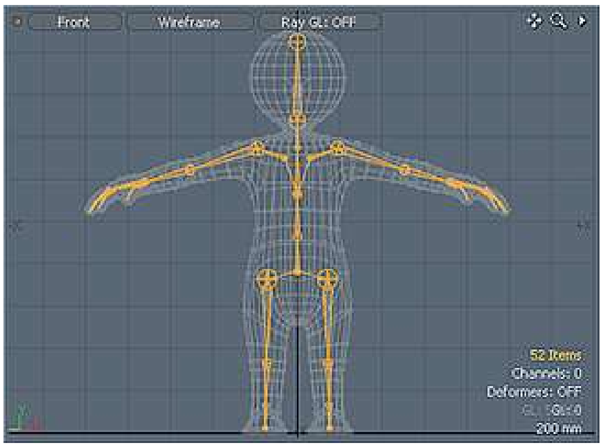


Figure 5

A surface representation used to draw the character mesh and a hierarchical set of interconnected bones used to animate the mesh (figure 5). This technique is used primarily for animating organic objects but can also be used to control deformation of an object like doors, robotic arm. Usually an object is rigged and then animated. Animation data is saved per bone so you can manipulate the bones in real time. To use skeletal meshes for dynamic car damage you can give the bones a collider and give each bone high dampening. This way bones don't move but when a heavy object is thrown towards it the bones will move.

To get somewhat realistic deformation of a skeletal mesh you need a lot of bones. Having a lot of bones in each part of the car mesh takes a huge hit on performance [2]. Especially if you are simulating multiple cars at the same time. You also have very limited control on how the deformation.

3.2.3 PROCEDURAL MESHES

Generate and change meshes/collision at runtime. This works by copying all the data of a static mesh asset and making a procedural mesh from this data. Then when hit events are triggered moving vertices based on the force and where it was hit. For this you need to loop over an array of all the vertices in this mesh which could lead to a serious performance hit. This can be optimized by splitting up parts of a procedural mesh into smaller sections. Then we can update smaller sections of the mesh thus needing to loop over less vertices. Which can drastically improve performance.

Collision is really accurate since the procedural mesh component creates collision at runtime if vertices are updated.

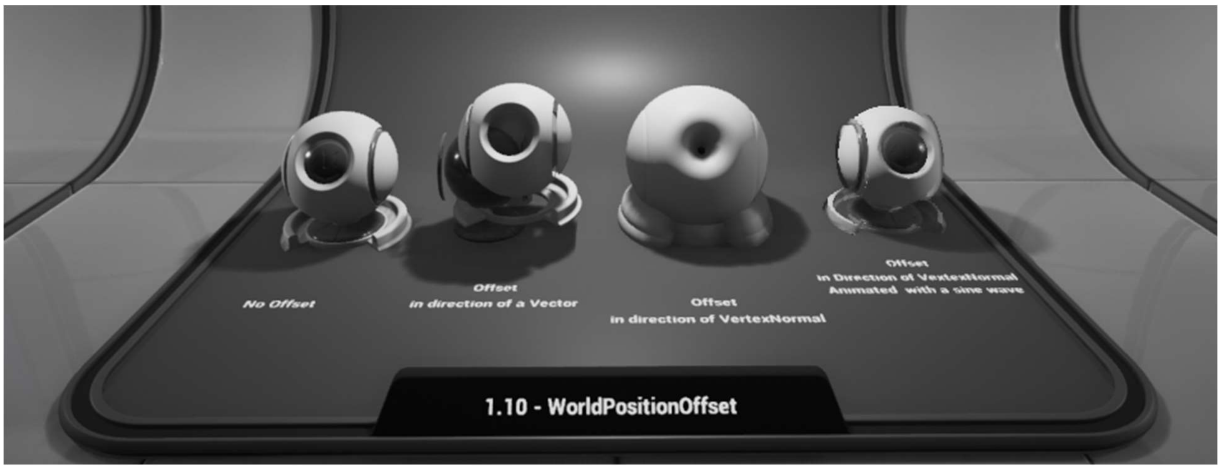
The need to create your own deformation formula. Based on the impact point, impact force, vertex position and stiffness of the material that you have vertices need to move in different ways. Creating a good looking deformation formula that creates realistic bending of the mesh is needed [3].

3.2.4 MATERIAL VERTEX DISPLACMENT

Material/Shader allows for moving the vertices on the GPU. This is useful for making objects move, change shape, rotate and more. Note that moving vertices on the GPU does not change the collision. The object bounds stay the same. The renderer still uses those original bounds. This means that you may see culling and shadowing errors when you get really close to the object. The original bounds can be changed but this can have an impact on performance or lead to shadow mapping errors.

Because collision isn't updated when the vertices are moved this is not a good method to make car deformation. But it is something nice to add more depth to car deformation. Materials make it possible to add in localized dents, scratches. Without a big performance hit because it's calculated on the GPU. The mesh can look even better if you allow for tessellated vertices to be moved. Some examples of this can be seen in figure 6.

Figure 6


No Offset
Offset
in direction of a Vector
Offset
in direction of VertexNormal
Offset
in Direction of VertexNormal
Animated with a sine wave
1.10 - WorldPositionOffset

How do we save impact data?

3.2.4.1 MATERIAL PARAMETER COLLECTION

This would be the obvious choice. You can easily save vector and scalar parameters. Which can be referenced in any material (max 2 collections per material). But we want a separate parameter collection per car part and since parameter collections can not be created at runtime. A collection can have up to 1024 scalar parameters and 1024 vector parameters. So in general using a vector parameter collection doesn't seem like a good technique to save dent information. Especially if you will have multiple cars in a scene. Since all the cars in the scene would have to share 2 parameter collections. Which could lead to each car part having very little amount of dents that could be saved.

3.2.4.2 TEXTURES

Save impact position and impact forces into render targets. You can create render targets at runtime and assign them to a dynamic material so each car part could have it's own render target. You can then have different sizes of render targets for car parts that are bigger or smaller. Big issue with render targets and especially saving impact locations into them is you have to set the individual value of a pixel. This could be very heavy on the system. One more issue is to read all the impact data from the texture in a UE4 material you need to use custom HLSL code.

3.2.4.3 DATATABLES

Data tables in UE4 are like excel tables they can be dynamically generated via C++ but there is currently no way to use this data in the material editor or any way to pass it to the material editor. This technique can not be used because of this.

3.3 GLASS

Cars use laminated glass for windshields and door windows. Laminated glass is well know for staying together when it is broken, reducing the likelihood of glass fall-out and reducing injures. [4] This is really important when creating the effect of car glass breaking. Since it shatters completely differently then normal glass. There is not one technique that can produce all of this so we will be needing a combination of techniques.

Figure 7

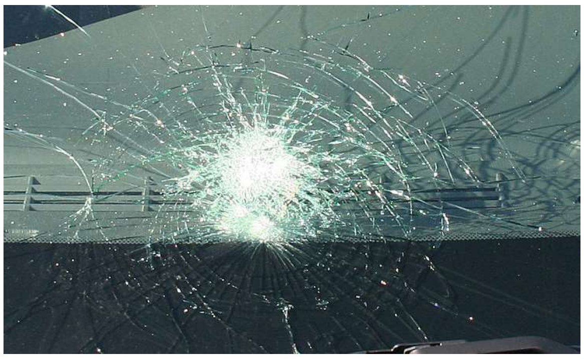
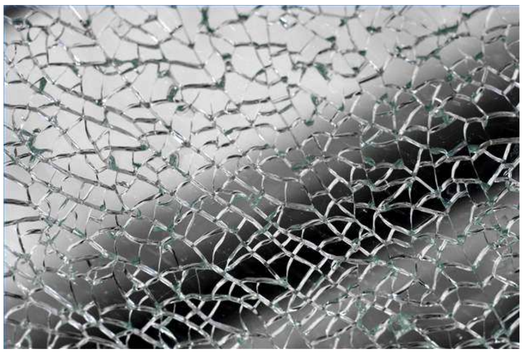


Figure 8



3.3.1 MATERIALS

Whenever you want to create a visual effect creating a shader for it is a good option. The approach I wanted to try was to render impacts to a render target as a mask. Then use this mask in the material to render impacts. Materials are purely visual and can be used to create effects like figure 7 & 8**.**

To create all the tiny cracks in the glass around the impact we can use Voronoi noise to fake it. We can generate a diffuse and normal from the Voronoi noise.

3.3.2 DESTRUCTIBLE MESHES

To create actual holes in glass a good option would be to use destructible meshes. Destructible meshes are pre-calculated fractured objects. What happens is when the destructible mesh receives a hit it replaces the static mesh with the fractured mesh. Based on some parameters that define how the chunks should behave you can create holes in this destructible mesh by throwing physics objects at destructible mesh. When safety glass breaks it shatters into thousands of small little pieces. To generate and simulate physics on this many pieces would be impossible at a decent framerate. This is why the combination of shattering in the material fakes the look of there being a lot more pieces.

3.3.3 PARTICLES

To make the glass impact more realistic creating a particle that spits out glass splinters when there is an impact is a must.

4. CASE STUDY

4.1 MORPH TARGETS/BLEND SHAPES

4.1.1 QUICK INTRODUCTION

A morph target is a deformed version of a shape. Morph targets make it possible to blend between multiple deformed versions of a shape. This is often used for facial features such as blinking, raising eyebrows, frowning but is also often used to damage objects. Multiple morph targets can be used to create complex vertex-driven animation.

4.1.2 Creating morph targets

4.1.2.1 3DS MAX CREATING FBX

Most 3D modeling software has a way of creating morph targets but this will describe the way to do it in 3Ds Max 2019. Create a base mesh this will be the default "pose" of the mesh. Create one or more variations of this mesh. Apply the "Morpher" modifier to the base mesh and assign all the variations as morph targets. Select the base mesh and export the mesh as a .FBX file. Base mesh can be seen in figure 9 and the distorted version can be seen in figure 10.

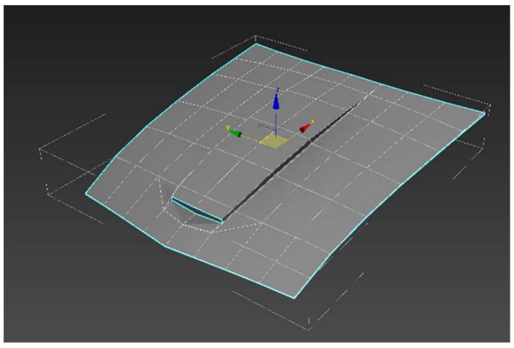


Figure 9

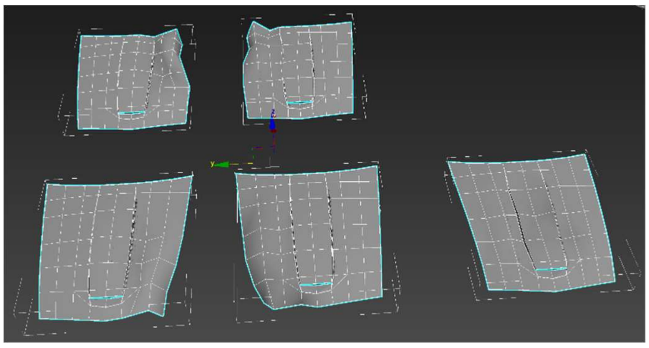


Figure 10

4.1.2.2 IMPORTING IN UNREAL ENGINE

Click the Import button in the content browser. Navigate to and select the FBX file you want to import in the file browser that opens. Choose the appropriate settings in the Import dialog. Make sure that Import Morph Targets is enabled.

You can check if the import was successful by adjusting the Morph Target Weight sliders in the Skeletal Mesh Preview editor(open the Skeletal Mesh Preview Editor by double-clicking the skeletal mesh asset in the content browser). The sliders goes from -1.0 to 1.0 using negative values inverts the normal result. This can give weird artifacts/results but might be useful in some cases. Unreal also has a nice tool to debug morph targets . It lets you view a heat map (figure 12) of what vertices change for each morph target. How to open the tool can be seen in figure 11.

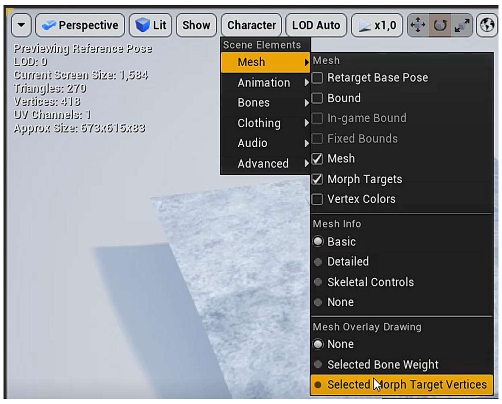


Figure 11

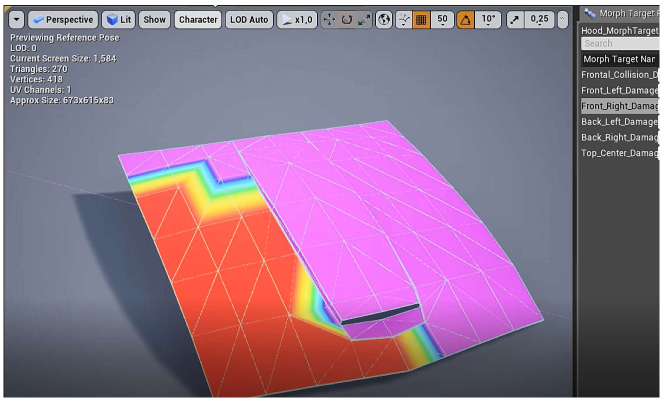


Figure 12

4.1.2.3 USING MORPH TARGETS

Too change the value of morph targets via blueprints you only need one node (figure 13). This node requires the skeletal mesh that needs to be edited. The name of the morph target. This is annoying if you have a lot of morph targets. Since you have to manually type in all the names or have an array of all the names. The value is the new interpolation point the morph target is going to be set to.

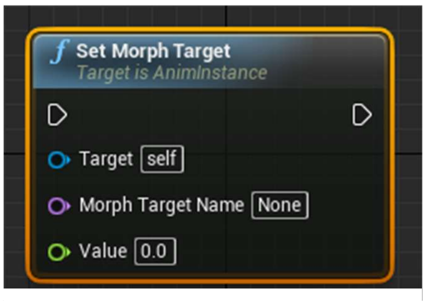


Figure 13

4.1.3 IMPLEMENTATION

I created a custom system that changes morph targets based on which colliders was hit. I created a collision part blueprint that has an array of morph targets and how much influence this collision part has on each morph target.

Connecting the collision part and the collider itself is just hooking up the function too the begin overlap collision event that each collider has in Unreal Engine. When the collider begin overlap collision event is triggered the collision part will handle this event. The collision part does a damage calculation based on the impact velocity and will update it's own morph targets based on the influence it has. The system itself is pretty modular and you can easily swap out skeletal meshes if it follows the same naming conventions.

4.1.4 COLLISION

4.1.4.1 PRE-DEFINED COLLSION STATES

Keep track of the total damage done to parts and toggle pre-defined collision volumes when the damage has reached a certain threshold. This is a time consuming method that's difficult to get right.

- Position colliders that physics objects can interact with when the object has no damage (figure 14)
- Position colliders that physics objects can interact with when the object is fully damaged (figure 15)
- Create array to store no damage colliders

- Create array to store fully damaged colliders
- Toggle colliders when damage of a certain morph target is too high (use the arrays for this)

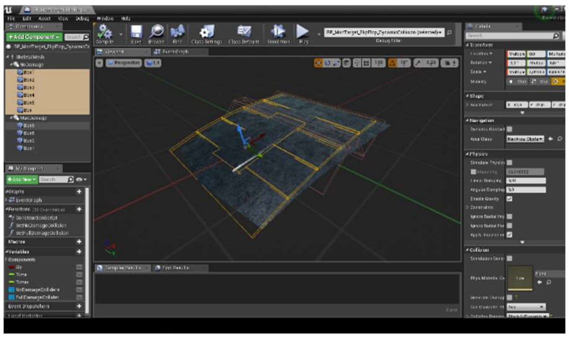


Figure 14

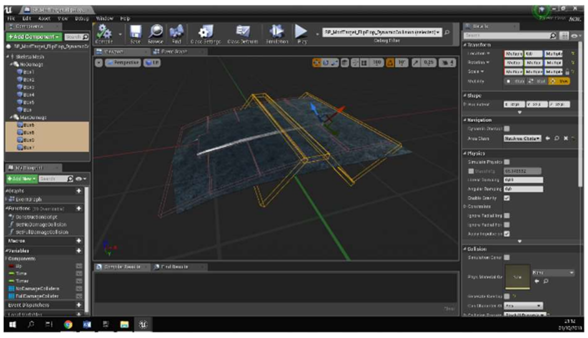
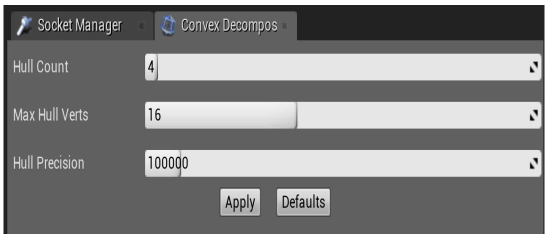


Figure 15

4.1.4.2 MODELLED COLLSION

To import the collision of a morph target export the deformed model as an fbx file. Import the file as a static mesh and bake a collision (figure 16) in Unreal Engine for this static mesh. This is a much nicer workflow and it's a lot easier/faster to generate accurate collision. You still have to swap between these different colliders via a damage threshold but instead of having 16 different box colliders you now have 2 colliders to toggle between and it gives a more accurate result.

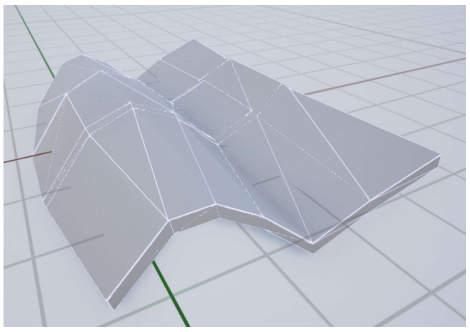
Figure 16



Socket Manager | Convex Decompos

Hull Count: 4
Max Hull Verts: 16
Hull Precision: 100000

Figure 17



4.1.5 ANNOYANCES

To use materials with morph targets you need to toggle a checkbox in the material itself. "Use With Morph Targets". If you don't set this checkbox too true from the moment any of the morph targets values change the material instance will be removed from the skeletal mesh.

4.1.6 WEAK/STRONG POINTS

Strong Points:

- With good deformation models can look really realistic
- Quick to debug
- Easy to implement
- Modular

Weak Points:

- Same result every time
- Inaccurate collision
- Time consuming

4.1.6 END NOTE

Morph targets can be a good way of simulating car damage if the game focus is not on car damage. The deformation this technique brings to the table is passible if done correctly with realistic damaged car models.

4.2 SKELETAL MESHES

4.2.1 QUICK INTRODUCTION

Create a bone structure in meshes in your 3D modeling software of choice. Then you apply colliders to each individual and let I simulate physics with high dampening. You need a lot of bones per skeletal mesh to get some realistic results though.

4.2.2 CREATING IN 3DS MAX

Create a bone structure for each individual car part. You can layout the bones anyway you want. Don't forget the end notches on the bones. Unreal will remove the last bones in the hierarchy so you can't use those. You can see the bone structure in figure 18.

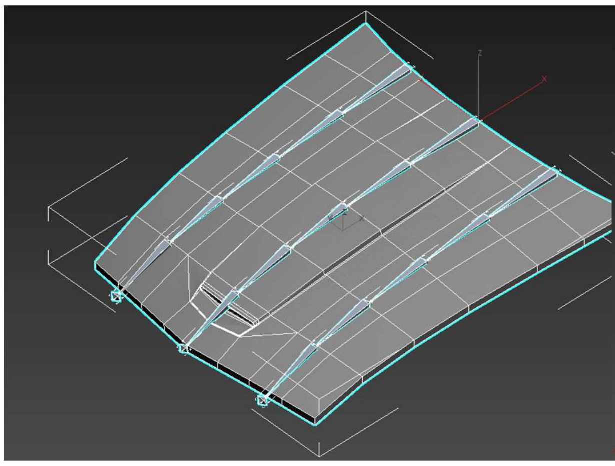


Figure 18

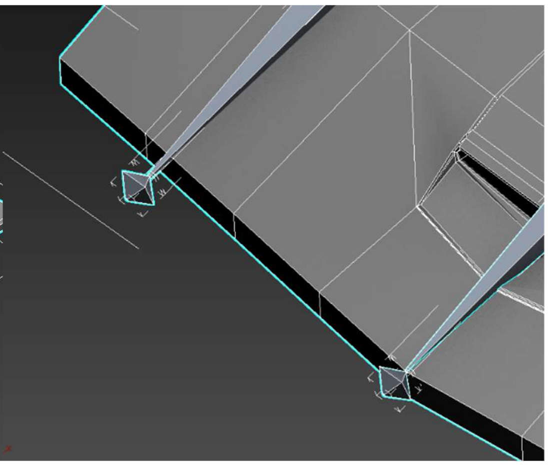


Figure 19

4.2.3 IMPORTING AND TESTING

All you have to do is import a skeletal mesh. Apply colliders to each bone as seen in figure 20 (make sure they don't overlap or the physics will break). Turn up the Linear & Angular dampening of each bone (figure 21). This will cause the skeletal mesh to deform like metal. Throw the skeletal mesh into the scene and your done. If you throw physics objects at this mesh it will deform nicely.

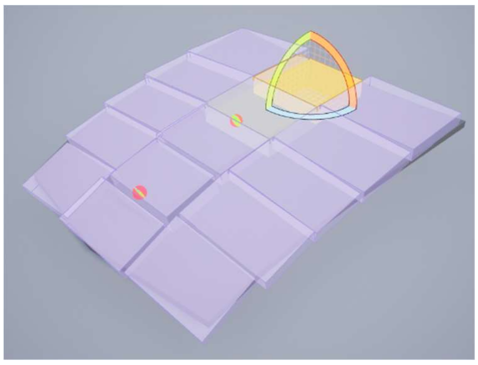
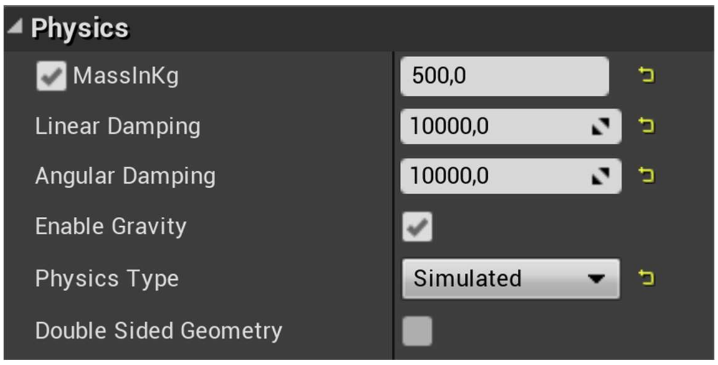


Figure 20



Physics

✓ MassInKg	500,0
Linear Damping	10000,0
Angular Damping	10000,0
Enable Gravity	✓
Physics Type	Simulated
Double Sided Geometry	

Figure 21

4.2.4 ANNOYANCES

Moving the skeletal mesh around breaks the collision of the bones. So what you have to do is teleport the skeletal mesh because this won't break the deformation. To make the skeletal mesh hood react to physics without breaking the bone hierarchy. You need to simulate the physics with physics joints using an invisible hood. Then take the rotation and translation from the simulated hood and apply them too the skeletal mesh.

4.2.5 WEAK/STRONG POINTS

Strong Points:

- Produces nice result out of the box

Weak Points:

- Time consuming to model
- Time consuming to add colliders
- Not modular
- Can't be improved a lot
- Huge performance hit

4.2.6 END NOTE

Skeletal meshes are easy to implement but they are time consuming to set-up. You don't have a lot of control about how the deformation behaves but it gives a good result just using this set-up. Even though there isn't a lot you can do to improve this effect . Skeletal meshes is a viable cheat technique if you don't need to simulate multiple cars.

4.3 PROCEDURAL MESH COMPONENT

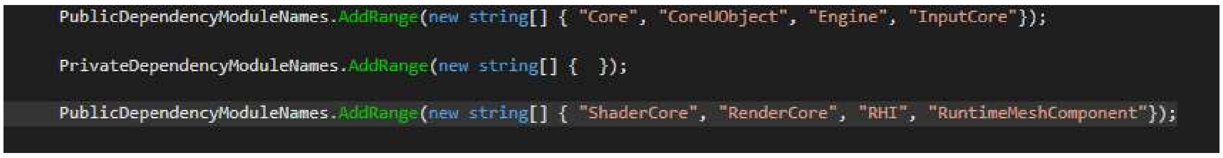
4.3.1 QUICK INTRODUCTION

Copy all the data from a static mesh to a procedural mesh. The vertices of a procedural mesh can be moved at runtime. As well as collision cooking at runtime. This can be used to create dents in a mesh with very accurate collision.

Instead of using UE4 build in procedural mesh component I opted to use the Runtime Mesh Component plugin. Since it far surpasses unreal engines counterpart in both features and efficiency. It on average uses 1/3 the memory of the Procedural Mesh Component, while also being more efficient to render and faster to update mesh data.

4.3.2 SETTING UP

To install the runtime mesh component plugin you will need to add multiple modules in order to get the new component working. Modules needed are ShaderCore, RenderCore, RHI, RuntimeMeshComponent. The last line in figure 22 needs to be written in the build.cs file.



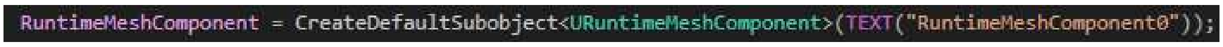
```
    PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore"});

    PrivateDependencyModuleNames.AddRange(new string[] {  });

    PublicDependencyModuleNames.AddRange(new string[] { "ShaderCore", "RenderCore", "RHI", "RuntimeMeshComponent"});
```

Figure 22

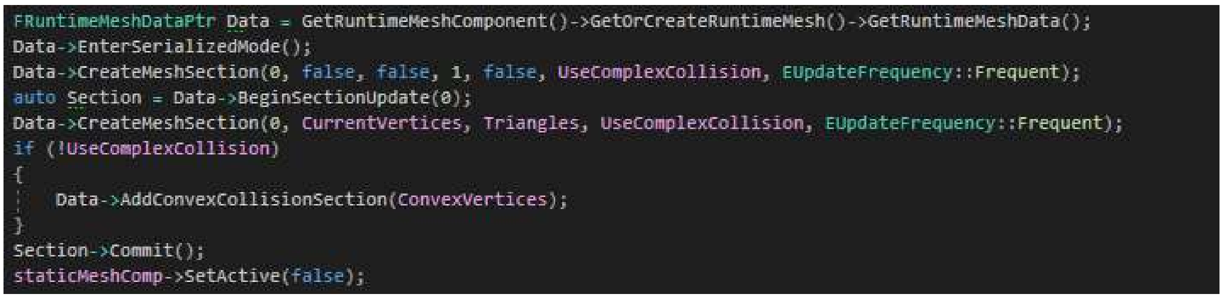
You have to spawn the runtime mesh component in order to use it this is done with the code in figure 23**.**



```
RuntimeMeshComponent = CreateDefaultSubobject<URuntimeMeshComponent>(TEXT("RuntimeMeshComponent0"));
```

Figure 23

Retrieve all data such as vertex positions, normal, tangents, texture coordinates, vertex buffer, index buffer. Then create a mesh section in the procedural mesh component based on this retrieved data. The way to do this can be seen in figure 24**.** The most important part is the "Data->CreateMeshSection" this is the function that actually creates the mesh and cooks collision for this object.

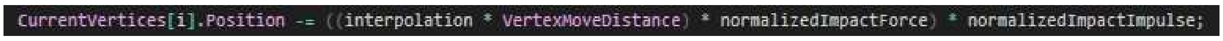


```
FRuntimeMeshDataPtr Data = GetRuntimeMeshComponent()->GetOrCreateRuntimeMesh()->GetRuntimeMeshData();
Data->EnterSerializedMode();
Data->CreateMeshSection(0, false, false, 1, false, UseComplexCollision, EUpdateFrequency::Frequent);
auto Section = Data->BeginSectionUpdate(0);
Data->CreateMeshSection(0, CurrentVertices, Triangles, UseComplexCollision, EUpdateFrequency::Frequent);
if (!UseComplexCollision)
{
    Data->AddConvexCollisionSection(ConvexVertices);
}
Section->Commit();
staticMeshComp->SetActive(false);
```

Figure 24

4.3.3 USING THE RUNTIME MESH COMPONENT

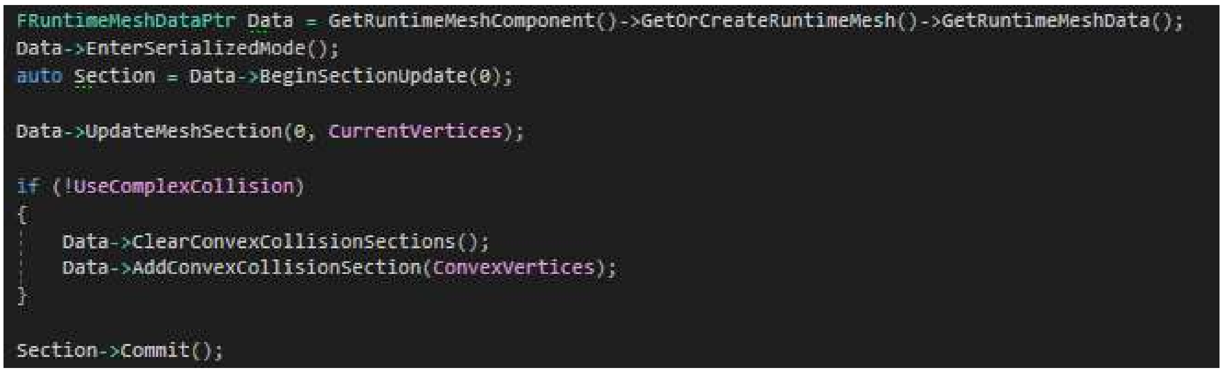
Now that we can copy a static mesh and create a procedural mesh with accurate collision now we have to enable the hit event of the object and based on the hit move certain vertices. This can be done by looping over all the vertices of the procedural mesh and updating the position. Moving the vertex is based on your own formula (figure 25).



```
CurrentVertices[i].Position -= ((interpolation * VertexMoveDistance) * normalizedImpactForce) * normalizedImpactImpulse;
```

Figure 25

After moving the vertices the procedural mesh can be updated by simply calling “Data->UpdateMeshSection” (figure 26). Note that the data has to be serialized before updating mesh data.



```
FRuntimeMeshDataPtr Data = GetRuntimeMeshComponent()->GetOrCreateRuntimeMesh()->GetRuntimeMeshData();
Data->EnterSerializedMode();
auto Section = Data->BeginSectionUpdate(0);

Data->UpdateMeshSection(0, CurrentVertices);

if (!UseComplexCollision)
{
    Data->ClearConvexCollisionSections();
    Data->AddConvexCollisionSection(ConvexVertices);
}

Section->Commit();
```

Figure 26

4.3.4 ANNOYANCES

Simulating physics on a runtime mesh is quite tricky. There are two possible routes you can take with complex collision or with convex collision.

Convex Collision:
This supports physics out of the box but you don’t have accurate collision. It’s also very hard to get nice convex collision because it doesn’t generate it automatically. There is one small problem you have to enable the physics at runtime otherwise it doesn’t work.

Complex Collision:
You can’t simulate physics when complex collision is enabled so what you have to do is simulate the static mesh with convex collision. Then replicate the changes in position and rotation from the static mesh onto the runtime mesh.

4.3.5 WEAK/STRONG POINTS

Strong Points:

- Accurate Collision
- Flexible
- Allows for optimization

Weak Points:

- Self made formula not realistic
- Per vertex iteration is slow
- No documentation

4.3.6 END NOTE

Procedural mesh seems like a technique you can take really far if you have time to advance the vertex moving formula. As well as something that can be highly optimized by creating different sections in on mesh. This would minimize the amount of vertices it needs to iterate.

4.4 MATERIAL VERTEX DISPLACEMENT

4.4.1 QUICK INTRODUCTION

Achieving small additional denting on a mesh. Fully on the GPU by using a mix of render targets and material vertex displacement. Every object will creates its own render target with different sizes.

4.4.2 SAVING POSITION AND IMPACT DATA

Using render targets to save and read impact data. We can write a color to a specific pixel on a render target. This pixel can then be read in a shader to then move certain vertices. A render target full of impact data might look something like this. Each object could have different size render targets based on how much detail this object requires.

A texture filled with impact data might look something like figure 27.

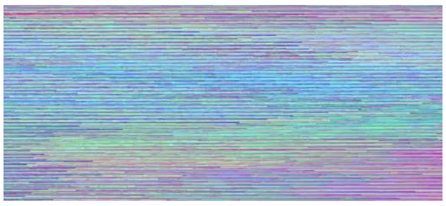
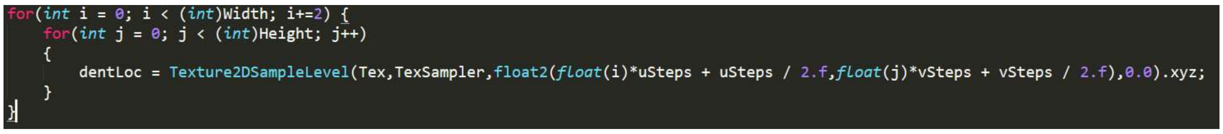


Figure 27

4.4.3 READING POSITION AND IMPACT DATA

We need to get access to every pixel in a texture individually. This means that we will have to loop over every pixel for each vertex. The only way to do this is via a custom HLSL node in the material editor. Looping over every pixel might look something like figure 29.



```
for(int i = 0; i < (int)Width; i+=2) {
    for(int j = 0; j < (int)Height; j++)
    {
        dentLoc = Texture2DSampleLevel(Tex,TexSampler,float2(float(i)*uSteps + uSteps / 2.f,float(j)*vSteps + vSteps / 2.f),0.0).xyz;
    }
}
```

Figure 29

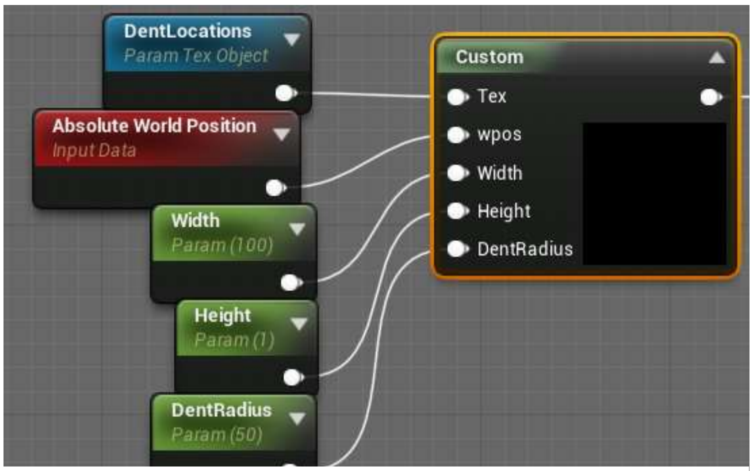


Figure 28

The width and height of the texture have to be known in order to sample the right pixel. You can either make the width and height static but then every object that has this material will need to have the same size texture. But we can create scalar parameters which you then change via a blueprints or C++ to process different sized textures for different kind of objects. In the end I used a DentLocations parameter wich is the rendertarget that holds impact data. Each object has it's own dynamically made texture. Absolute World Position data to move the position data from world space to local space. Width and Height of the texture. DentRadius which is how far from the impact vertices should be affected. All these variables are set via blueprint or C++ for each individual object.

Using the output of this custom node you can displace the vertices which gives the result seen in figure 30. This is an extreme result of the vertex displacement. To actually achieve smaller dents you could limit how much the vertices can move.

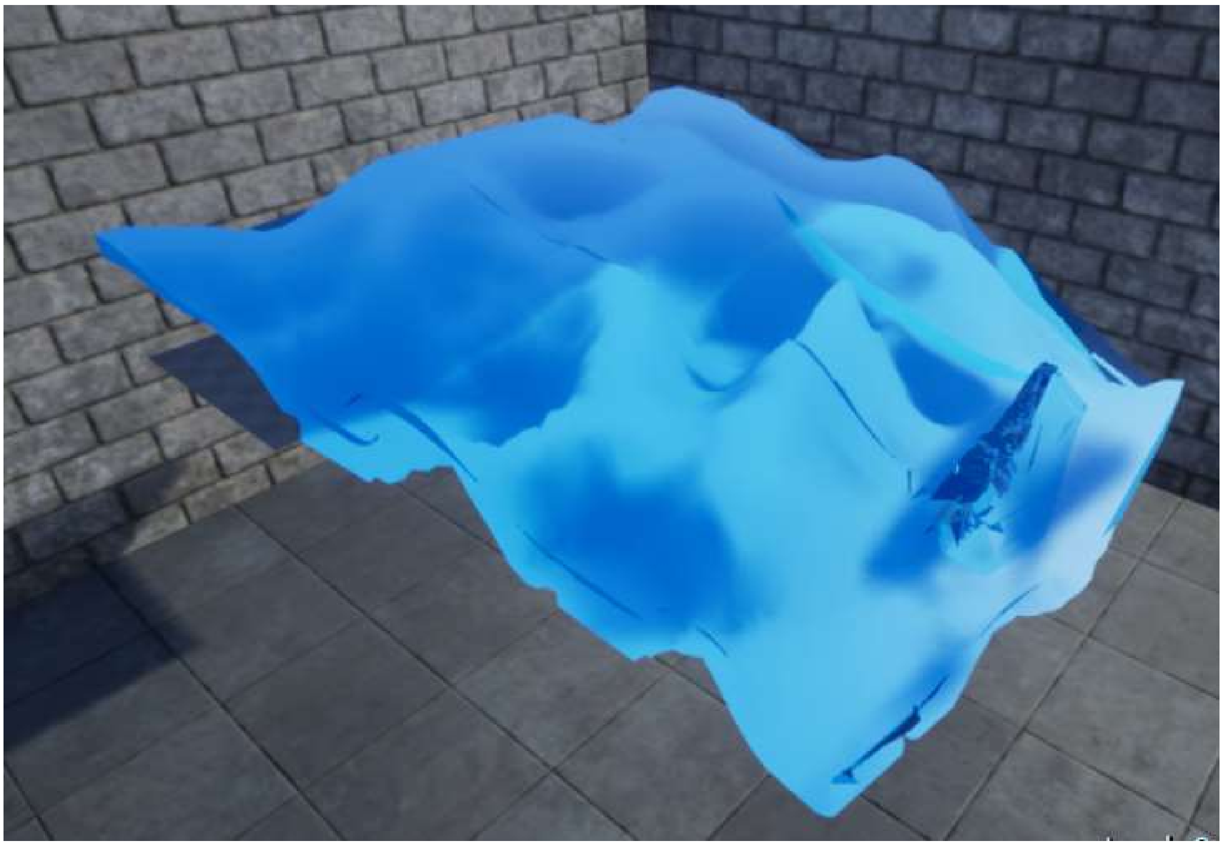


Figure 30

4.4.4 ANNOYANCES

Because the need for a custom HLSL node unreal engine isn't able to optimize the material and you can't check the performance of the material. Because the vertex displacement is calculated on the GPU the bounds of the object does not change which might lead to the object flickering when the object is reaching the edges of the camera.

4.4.5 WEAK/STRONG POINTS

Strong Points:

- Tessellation for higher detail
- Fast on the GPU

Weak Points:

- No collision
- No optimization because of HLSL node

4.4.6 END NOTE

This might be nice as an additional effect but I don't know if the effect it produces is really worth the overhead it produces since there is no way to check the performance.

4.5 GLASS

4.5.1 DESTRUCTIBLE MESHES

Destructible meshes are pre-calculated fractured versions of static meshes. Fracturing of these objects can be done with a tool made by NVIDIA. The fracturing can be customized a little bit by adjusting parameters. The technique used for fracturing these object is the Voronoi algorithm. The Voronoi algorithm shatters static meshes like (figure 31). The pattern looks somewhat alike with safety glass breaking. The picture (figure 32) has about 200 chunks that are pre-calculated which strikes a good balance between performance and looks. Fracturing it into more chunks will cause unreal engine to lag.

Figure 31

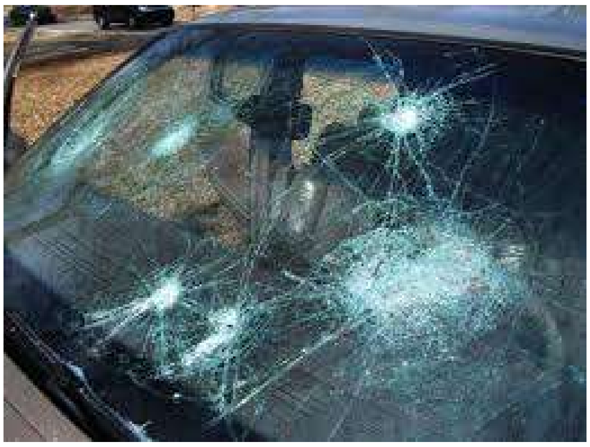
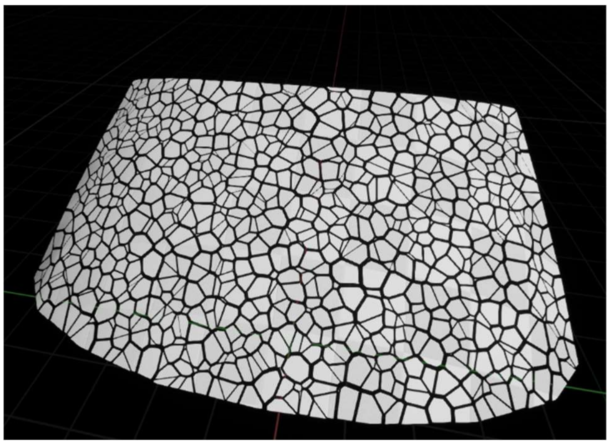
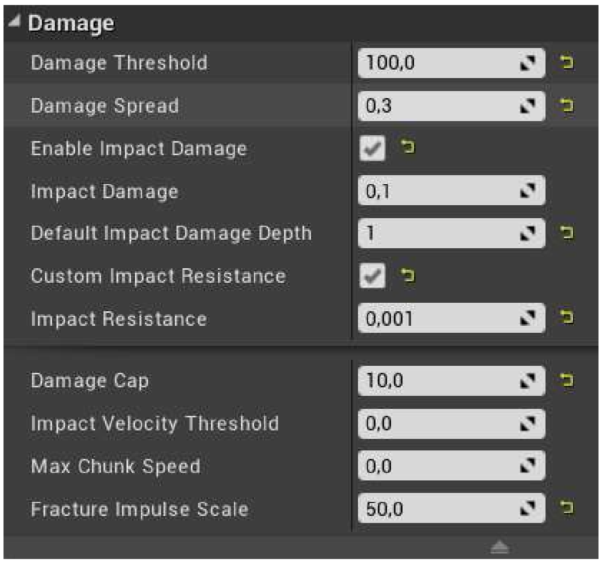


Figure 32



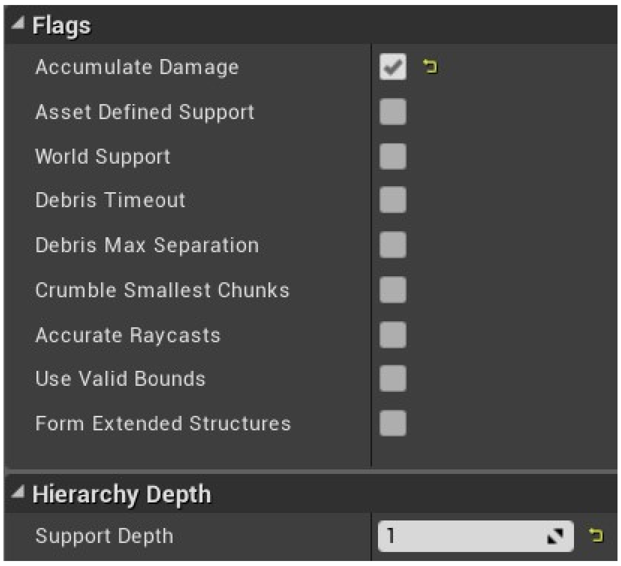
To make a destructible mesh react like glass that's being shattered we need to tweak a lot of parameters shown in (figure 33).



Damage

Damage Threshold	100,0
Damage Spread	0,3
Enable Impact Damage	✓
Impact Damage	0,1
Default Impact Damage Depth	1
Custom Impact Resistance	✓
Impact Resistance	0,001
Damage Cap	10,0
Impact Velocity Threshold	0,0
Max Chunk Speed	0,0
Fracture Impulse Scale	50,0

Figure 33



Flags

Accumulate Damage	✓
Asset Defined Support	
World Support	
Debris Timeout	
Debris Max Separation	
Crumble Smallest Chunks	
Accurate Raycasts	
Use Valid Bounds	
Form Extended Structures	

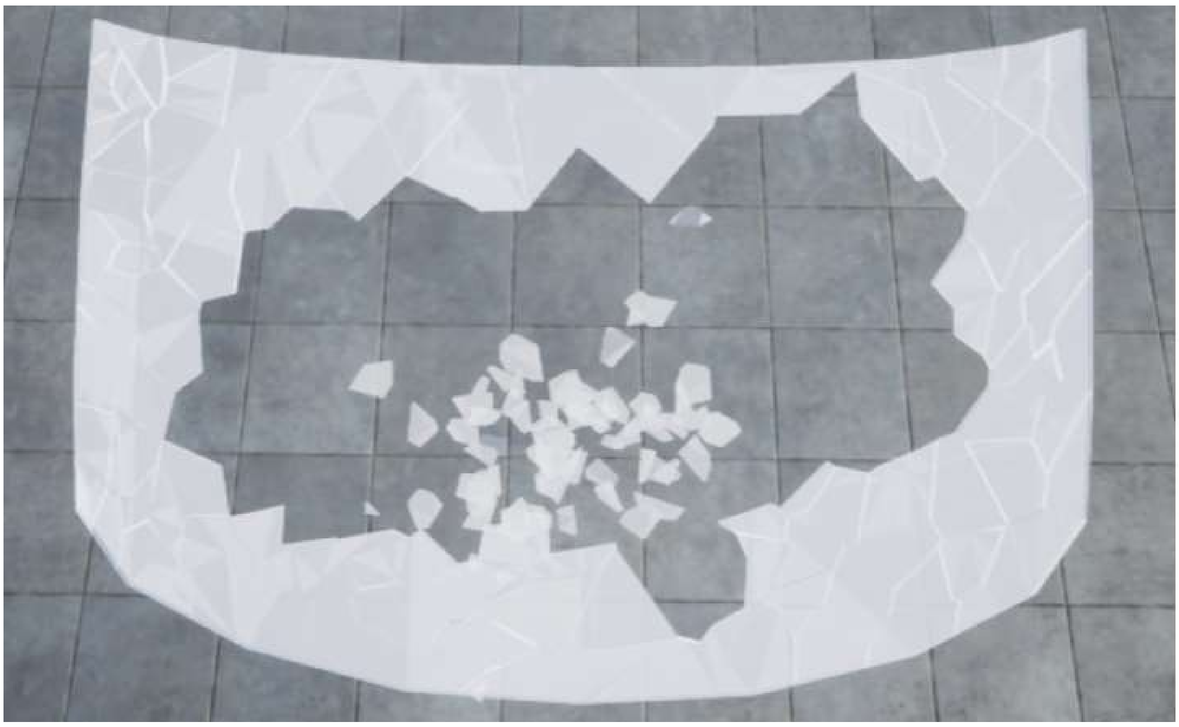
Hierarchy Depth

Support Depth	1

Figure 34

As well as one more very important step. If you would simulate the destructible mesh now. From the moment the destructible mesh gets hit all the chunks will be simulated. This is not what we want. We want certain parts to stick together and they only simulate physics once a big enough force is applied to the chunk. To make this work you have to select some chunks that you don't want to fall (preferably corners). Set the chunk parameters of these selected chunks to "Do Not Fracture".

Figure 35



As you can see in (figure 35) the result the destructible mesh produces is a good representation of glass being destroyed. But on its own it still doesn't come close to being safety glass, so a combination of methods will be needed.

4.5.2 MATERIALS

Using a combination of render targets and materials to create the effect of glass breaking. When the static mesh gets an impact event retrieve the UV location of the impact hit. Draw the impact brush onto the render target using this UV location. Via the glass material use the render target where hits are being rendered too to create the effect.

4.5.2.1 IMPACT RENDERING

To render impacts on the windows I created some materials that could be drawn to render targets. There are a couple of problems that need to be tackled. Such as how to get the UV location of a hit on a mesh. As well as render a material to a render target so we can use it in the glass material.

To get UV location of a hit on a static mesh there is a handy function that Unreal Engine provides called “Find Collision UV” (figure 36). This only works though if you use complex collision on the mesh. The “Find Collision UV” function gets the UV coordinates of the collider. If the static mesh has a generated convex collider it will return 0,0 coordinates since colliders do not have UV coordinates.

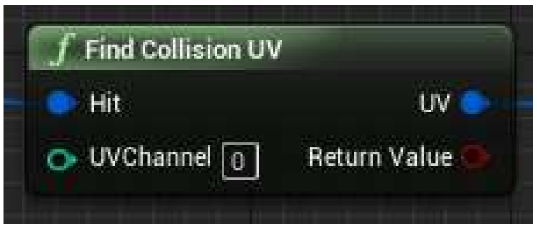
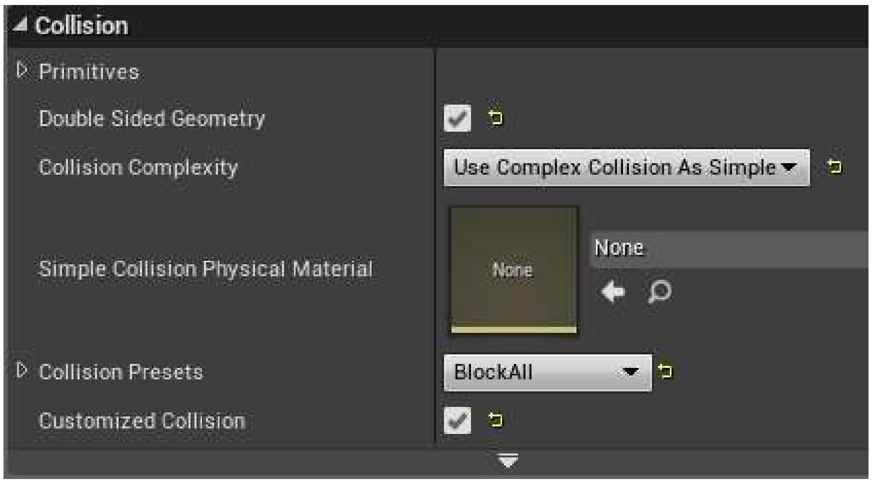


Figure 36



Collision

Primitives

Double Sided Geometry	✓
Collision Complexity	Use Complex Collision As Simple
Simple Collision Physical Material	None
Collision Presets	BlockAll
Customized Collision	✓

Figure 37

Now that we have the UV coordinate of the hit we can render the material to a render target at this UV location. This can be done by first setting the position of the brush and then drawing this material to a render target with a handy function in Unreal Engine.

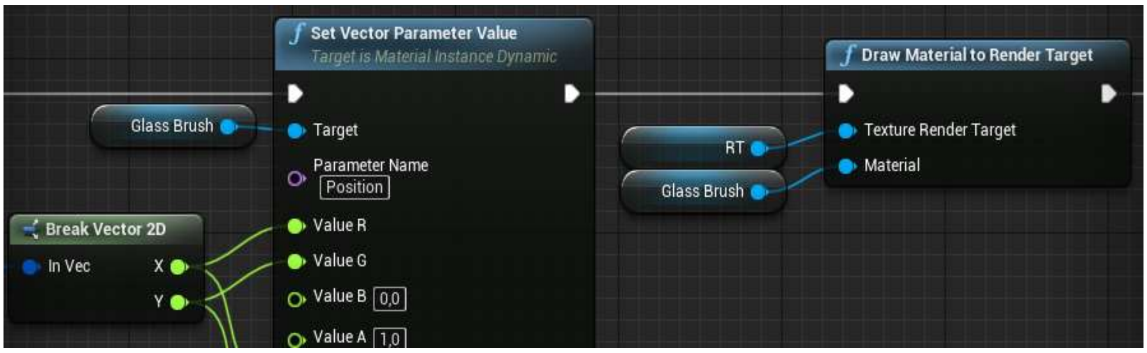


Figure 38

Setting the position parameter (figure 39) in the brush material is what actually changes the position of where the brush will be rendered on the render target. As you can see below we have a position parameter. We will then center this position and subtract the default texcoords from it. We might want to scale the brush as well to do this we use the “ScaleUVsByCenter” (figure 39) node which does the heavy lifting for us.

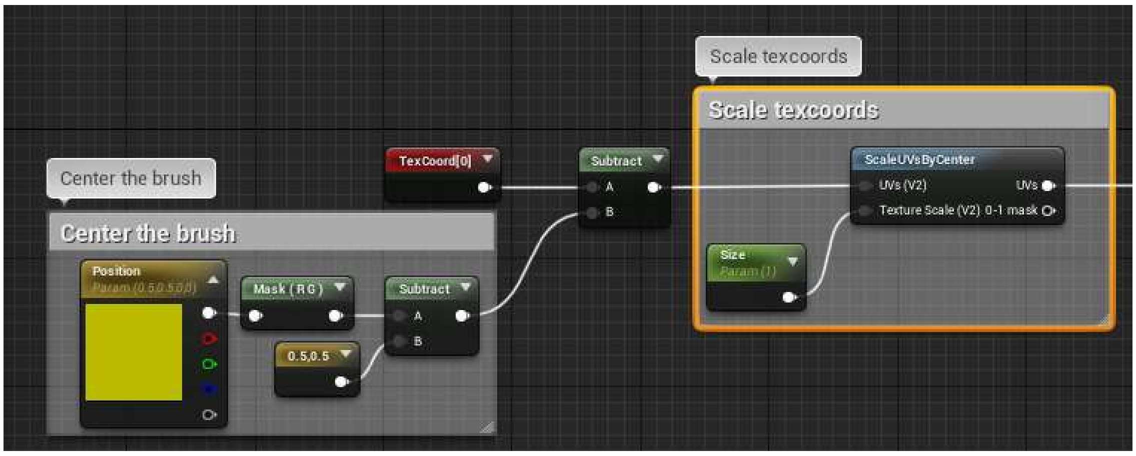


Figure 39

To make the effect a little bit better it would be nice if the impacts have a random rotation. So we rotate the calculated texture coordinates using the “CustomRotator” (figure 40) node and for the randomizer we use the “Time” node since we can’t generate random values on the GPU. Even though it’s not completely random it tricks the player into thinking it’s random.

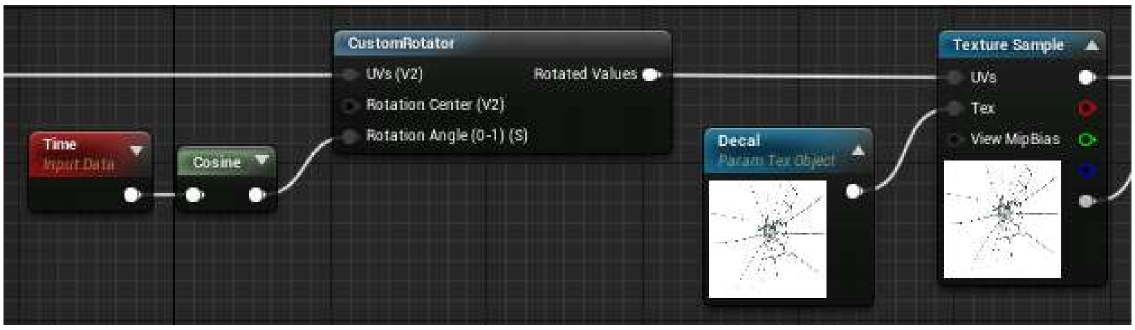


Figure 40

Now that we can properly render impacts to a render target on the correct UV location. We still have to use this render target in the glass material. This can easily be done by just making a “Parameter Texture Object” and via blueprint assigning this to be the created render target. Then feeding this directly into the base color of the material. I did the exact same but for the normal map of the crack to add some really fine detail too the glass material. This creates an effect that looks somewhat like figure 41.

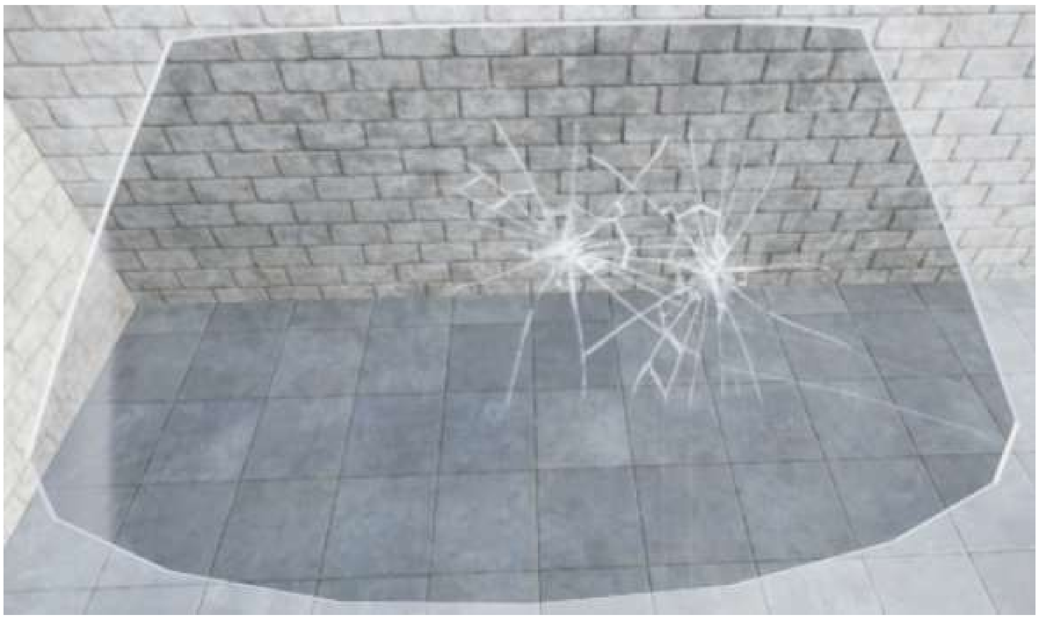


Figure 41

4.5.2.2 VORONOI GLASS BREAKING

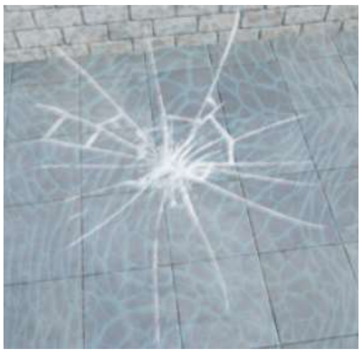


Figure 42

As you saw in figure 31 when safety glass receives an impact the glass shatters but stays together. To replicate this effect a Voronoi noise effect can be added to make it look like glass shattered around the impact. In the figure 42 the blue lines is the added Voronoi effect. It's not perfect but it adds more detail and realism.

This effect was achieved by creating a noise material that outputs an unlit Voronoi effect. The Voronoi effect itself is really cloudy so we need to get a sharper image from this before we can use it. We render this material to another render target.

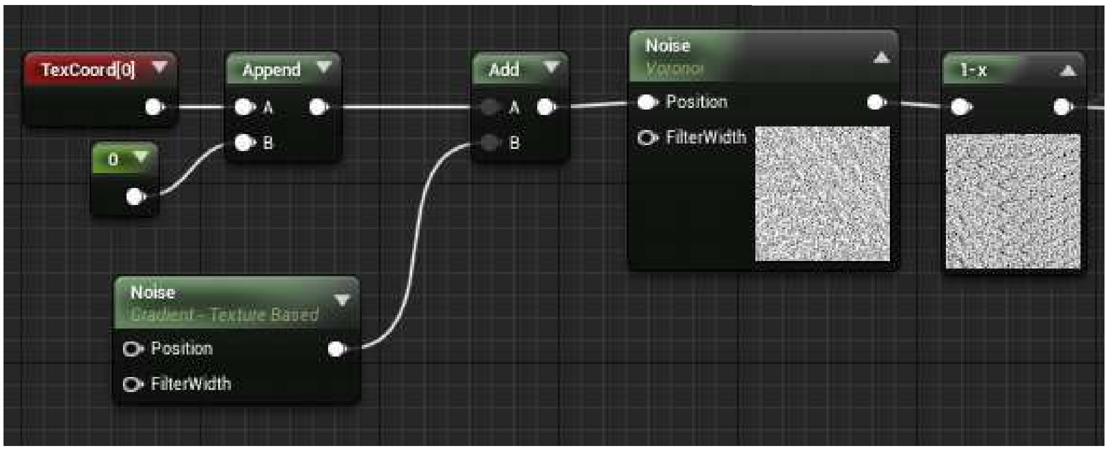


Figure 43

Now that we have a Voronoi texture we can use the "HighPassTexture" node in a different material. This node creates better contrast and filters out low frequency information out of the cloudy Voronoi texture. We then render this high pass Voronoi texture to another render target which we can finally use in our glass material. The rendering of the Voronoi only happens once so it doesn't matter how performant the Voronoi materials are. To use the final Voronoi texture we lerp between 2 colors. But now the Voronoi effect is always 100% visible but we only want cracks in a radius around the impact. To do this I created a circular brush that will draw to a render target that will be used as a mask for the Voronoi pattern. To use the mask we just multiply this mask by the linear interpolated output color. All the nodes for the Voronoi effect will look something like this in the material editor.

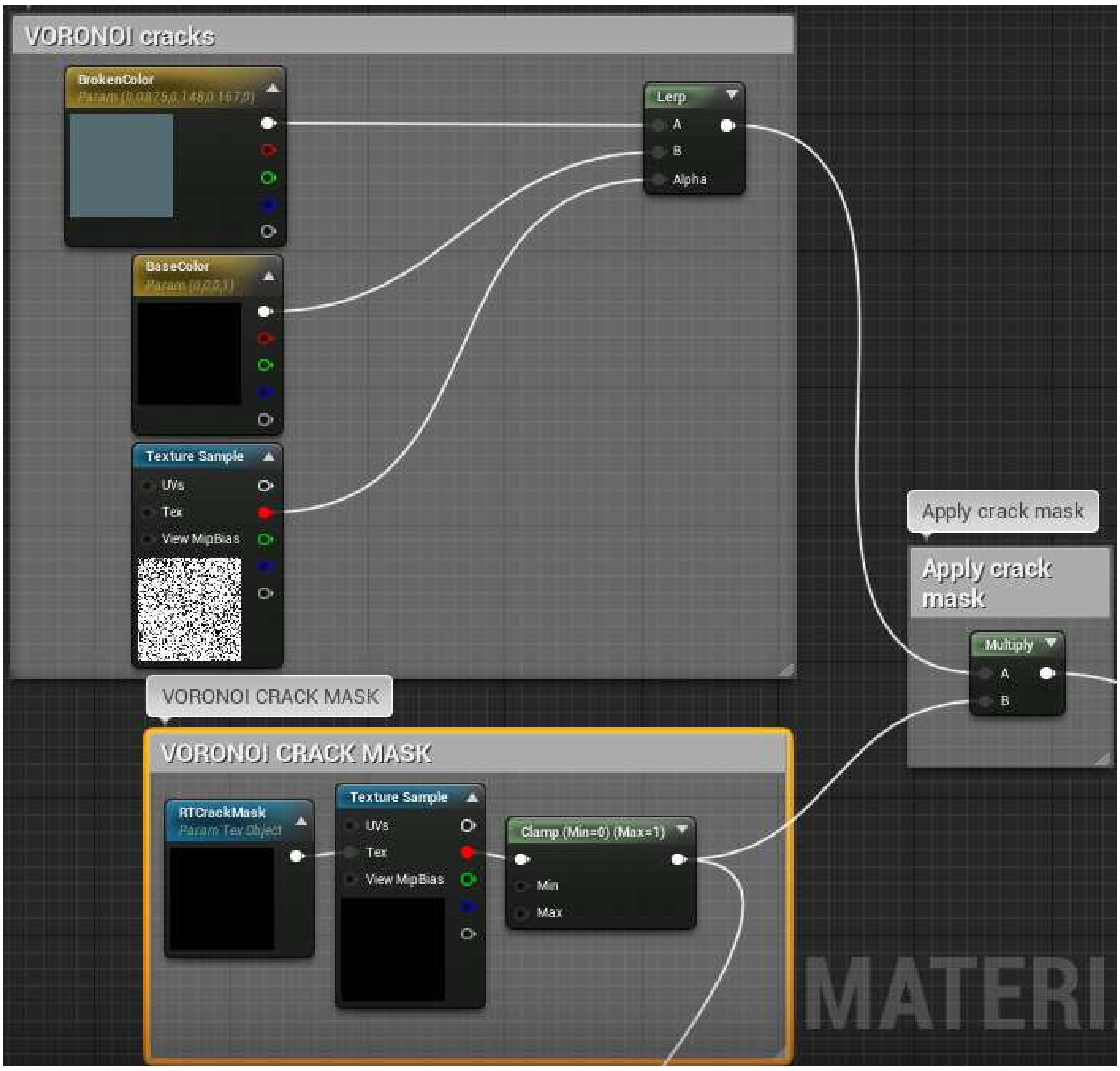


Figure 44

4.5.3 COMBINING

Now that we have two nice techniques it's time to combine them. What I did was to give the static mesh a certain amount of hits. This generates the impact rendering effect. From the moment the static mesh has had too many hits disable the static mesh and spawn in the destructible mesh. One very important part is to assign the material of the static mesh to the destructible mesh to make the transition almost seamless. Adding particles such as glass splinters flying off makes the effect more realistic. This can be seen in figure 45.

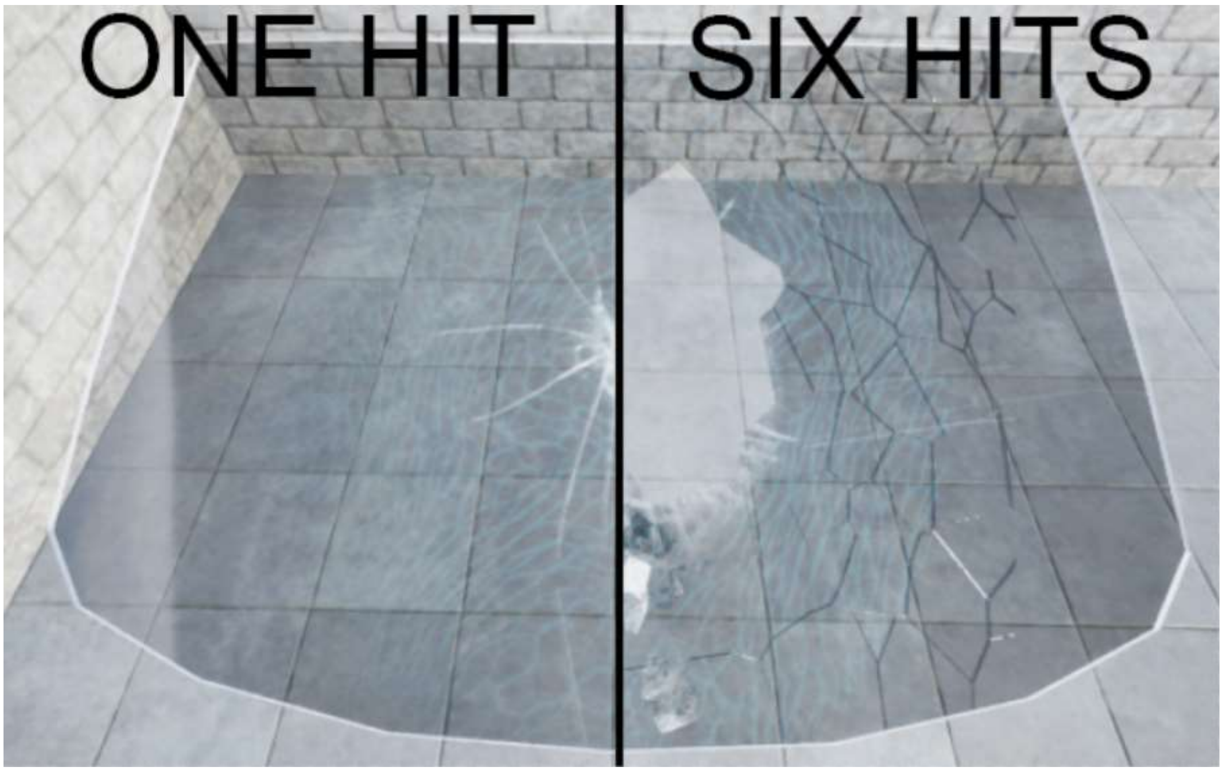


Figure 45

4.5.4 ANNOYANCES

While you could allow the normal static mesh to deform and generate proper impact rendering. When the destructible mesh gets spawned the deformation would be cancelled out and it wouldn't be a seamless transition.

4.5.5 END NOTE

This effect could be better if there is a way to fracture a procedural mesh at runtime. Instead of cooking the fracturing beforehand.

4.6 COMBINING TECHNIQUES

4.6.1 QUICK INTRODUCTION

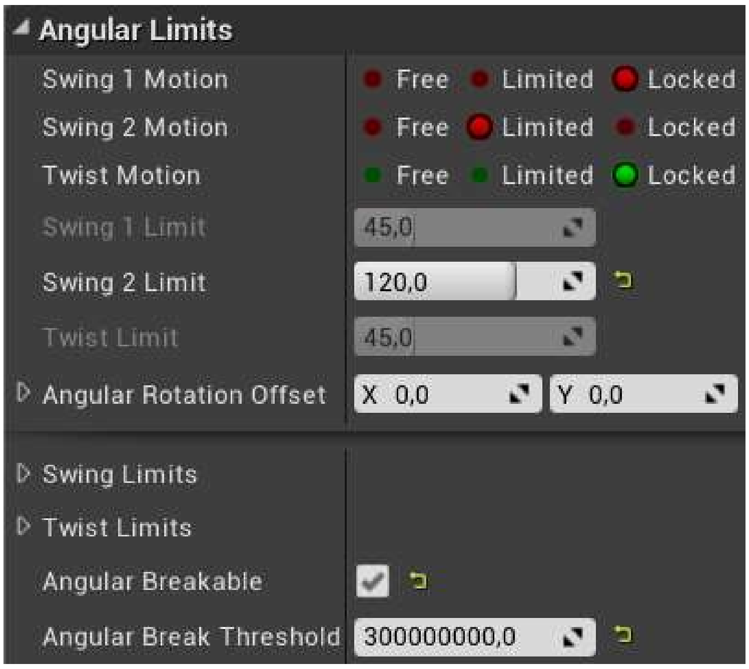
I will be combining the procedural mesh effect together with the glass breaking. The procedural meshes should dent and behave correctly with physics and have to be combined into a whole car. The car has to roll around realistically.

4.6.2 LOOSE PARTS

Some parts of the car have to be able to disconnect from the car. This is achieved using unreal engines physics constraints. You can add breakable limits for either angular or linear force.

The hood, trunk, doors use angular limits to constraint the rotational movement. A key part is the angular breakable Boolean. This allows for the physics constraint to break when the force applied on the constraint objects is to high.

Objects with different mass/size need to have higher/lower breakable limit. In figure 46 you can see the values for the hood physics constraint. The threshold have to be quite high before they stay attached.



Angular Limits

Swing 1 Motion: Free Limited [✓ Locked]

Swing 2 Motion: Free [✓ Limited] Locked

Twist Motion: Free Limited [✓ Locked]

Swing 1 Limit: 45,0

Swing 2 Limit: 120,0

Twist Limit: 45,0

Angular Rotation Offset: X 0,0 Y 0,0

Swing Limits

Twist Limits

Angular Breakable: ✓

Angular Break Threshold: 300000000,0

Figure 46

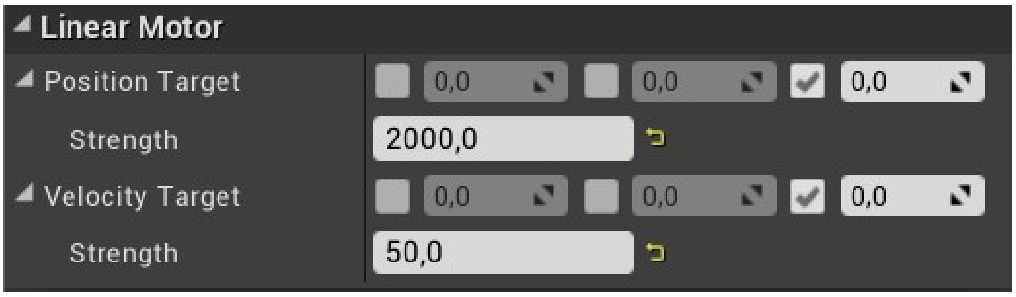
To attach bumpers, mirrors, exhaust linear limits were used since these objects are not supposed to rotate on the car.

4.6.3 SUSPENSION

To create realistic suspension for the car we need two constraints an axis constraint and a suspension constraint.

The axis constraint will make sure the wheel cannot rotate over more then one axis. In this case it can rotate freely on the y-axis.

The suspension constraint will make sure the wheel can only move along one axis. In this case it can move a little bit on the z-axis. We limit the movement by 20 units so the wheel can move 20 units up or down. One key part to get bouncy suspension on the car is the linear motor. This will try to keep the wheel in the same place thus working as suspension. The value's used to drive the linear motor can be seen in figure 47.



Linear Motor

Position Target	☐	0,0	☐	0,0	✓	0,0
Strength	2000,0					
Velocity Target	☐	0,0	☐	0,0	✓	0,0
Strength	50,0					

Figure 47

Now a static mesh should be able to roll around and react correctly to physics. But static meshes aren't deformable yet so we need to attach procedural mesh components now.

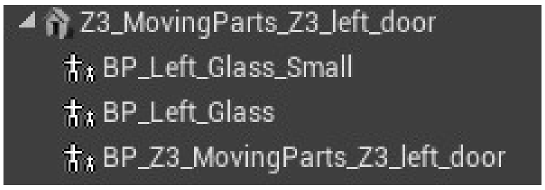
4.6.3 COLLISION LAYERS

I needed to create multiple collision layers to make sure certain parts of the car don't interact with one another. Otherwise the physics engine would explode and the car would fly off into infinity.
Procedural meshes should not collide with the simulated static meshes and glass.
Simulated static meshes should collide with everything except the procedural meshes
Glass should collide with simulated static meshes and physics objects
When joints break some actors change collision profile in order to now also collide with static objects etc.

4.6.4 ATTACHING PROCEDURAL MESHES & GLASS

To get the car to deform you have to attach a procedural mesh component for each static mesh component. To make the movement replicate just attach the procedural mesh blueprint as a child of the static mesh component. Hierarchy can be seen in figure 48.

The breakable safety glass is in a blueprint so this has to be attached as well to the according static meshes.



```
Z3_MovingParts_Z3_left_door
    BP_Left_Glass_Small
    BP_Left_Glass
    BP_Z3_MovingParts_Z3_left_door
```

Figure 48

4.6.5 ANNOYANCES

The procedural mesh deformation is hard to combine with the other effects. For example when the door gets deformed the glass should easily be shattered or move with the deformation. The same goes for the lights, bumpers etc. A way to get around this issue is to let parts disconnect very easily. This way you won't get clipping with the deformed mesh because it isn't attached anymore.

CONCLUSION

Morph targets are probably the way to go if you need something that isn't expensive to compute. For example a mobile game where you need some kind of car deformation. In that case morph targets should be able to provide a nice looking result.

Skeletal meshes should not be used for car deformation. While it might produce a nice effect it is very uncontrollable and it's also very heavy to compute because each car part needs a lot of bones.

Procedural meshes are probably the way to go since you can take this technique very far if you have the time. The extra time could be spend to create better integration of different techniques. As well as optimization by separating the mesh into multiple sections.

Material vertex displacement is a nice additive technique. But I cannot prove if the result is worth the overhead.

Glass breaking needs a combined effect. Impact rendering and actually shattering the glass. It's very hard to get a realistic representation of shattered safety glass since the pieces of glass are very tiny.

In the end while each technique by itself produces nice results it's difficult to get them to work together without clipping, physics glitches etc. Testing different methods to combine techniques is something you could put a lot of time in.

REFERENCES

[1] C. Liu, An analysis of the current and future stat eof 3D facial animation techniques and systems, 1st ed., Peking: Simon Fraser University, 2009, p. 155.

[2] D. Deryckere, "Car Destruction in Unreal Engine 4," [Online]. Available: https://gumroad.com/d/ae1bc65551b47ed3e51792b8958fae2c. [Accessed 5 10 2018].

[3] W. F. Hosford, "Fundamentals of Engineering Plasticity," [Online]. Available: http://assets.cambridge.org/97811070/37557/frontmatter/9781107037557_frontmatter.pdf. [Accessed 29 9 2018].

[4] S. Julia, Acoustic Interlayers for Laminated Glass - What makes them different and how to estimate performance, Unknown: GPD South America, 2012, p. 7.

[5] S. K.-S. Lee, "Introduction to Soft Body Physics," [Online]. Available: https://www.scribd.com/document/273373700/IntroductionToSoftBodyPhysics-Skeel. [Accessed 24 9 2018].

[6] E. Games, "Morph Target Pipeline," Epic Games, [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Content/FBX/MorphTargets. [Accessed 30 9 2018].

[7] E. Games, "Morph Target Previewer," Epic Games, [Online]. Available: https://docs.unrealengine.com/en-us/Engine/Animation/Persona/MorphTargetPreviewer. [Accessed 9 30 2018].

[8] E. Games, "Volume preserving procedural mesh," [Online]. Available: https://api.unrealengine.com/INT/BlueprintAPI/Components/ProceduralMesh/index.html. [Accessed 1 10 2018].

[9] J. Stam, "Real-Time Fluid Dynamics for Games," [Online]. Available: https://pdfs.semanticscholar.org/847f/819a4ea14bd789aca8bc88e85e906cfc657c.pdf. [Accessed 9 10 2018].

[10] J. Rajala, "Dynamic Controllable Mesh Deformation in Interactive Environments," [Online]. Available: http://liu.diva-portal.org/smash/get/diva2:635994/FULLTEXT01.pdf. [Accessed 9 10 2018].