Reactive Melee Combat In Unreal Engine 4

Daan Coppens

Introduction

While there are some widely discussed and proven ways to implement a ranged combat system (a gun, or a spell being fired) I feel that melee combat systems are more of a grey and less-defined area.

In this paper, we will take a look at the fundamentals of making a melee combat system work and how to make it more "reactive". The topics discussed will be universal, but some of the implementation will be focused around Unreal Engine 4.

Something as fundamental as a combat system can really dictate the genre of your game, so to get any sort of "genre debate" out of the way, we will look at this from the point of making an action oriented singleplayer game such as God Of War or Dark Souls. Games like these live and die by their combat system, and will also benefit greatly from any systems that give their attacks more impact or a better feel.

Case studies in successful melee action games

First of all, we will take a look at some games which have a successful melee combat system and try to dissect some fundamental mechanics. We will also look at some neat tricks they use to try and convey the proper feel of the combat.

Kingdom Hearts

As a nice intro, we'll be putting ***Kingdom Hearts*** under the microscope, a game which despite its simple feel really has a high skill factor and a finely tuned combat system. The more complex nature of the combat comes from the ability to cast spells however, so this leaves a simpler melee system which will give us a nice and basic start.

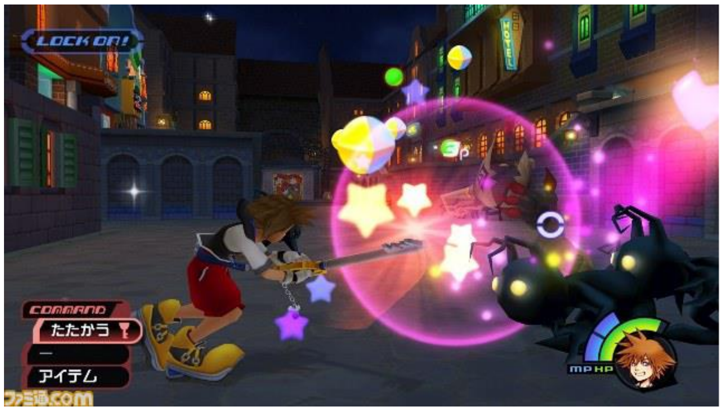
Attacking is fairly simple, press a button and your character swings his weapon. Combos are possible, but they mostly only have a chain of 3 attacks, and do not have any complex chaining mechanics (so there is no cancelling or chaining into other combos or anything).Movement during an attack is also impossible , further simplifying any advanced interactions. There are only 4 outcomes an attack can have.

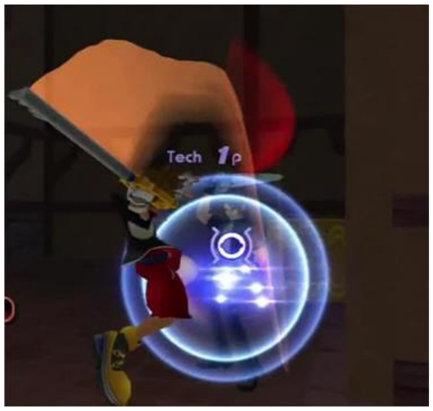
-**Nothing hit**: This simply fully plays your attack animation and either chains into your next attack or allows you to move again (you are always locked in an attack once it’s started).

-**Enemy Hit**: This also fully plays the animation, but obviously hits the enemy and plays a particle effect to emphasize the hit (but also obscure any weird hit, more on this later).

-**Enemy Parry**: A parry puts you in a retaliating animation and thus disallows any further hits of your weapon or combo. It mostly defends you against an enemy attack and sometimes leaves his defense open.

-**Blocking Hit**: A blocking hit occurs when you hit an object (most commonly terrain) that blocks your attack. It puts you in a longer retaliating animation and thus is not advisable.





1-an "Enemy Hit" and an "Enemy Parry"

Notice the big particles emphasizing the effect of your attack. They not only give the player a satisfying feeling and inform them of their success or failure. But also obscure some shortcomings that the animations may have. With the possibility to hit enemies from all sorts of directions it would be very hard to animate them correctly so they would react realistically to how they got hit. Collision detection was probably also done on things like cubes and not the real bodies, so enemies could get hit when their mesh was in fact not.

A small particle effect may have also properly conveyed a hit. But this could've left flaws visible or genuinely feel weird. While this may not have entirely been the developers' idea, it does have a very nice side-effect and attacks never make you feel like "Oh my god! That didn't hit at all!"

The second effect that really helps the combat feel great is the nice tracing effect on the actual blade for both you and certain enemies. This helps you keep track of enemy attacks and also more clearly conveys to the player whether his attack was a near or far miss.

Mordhau

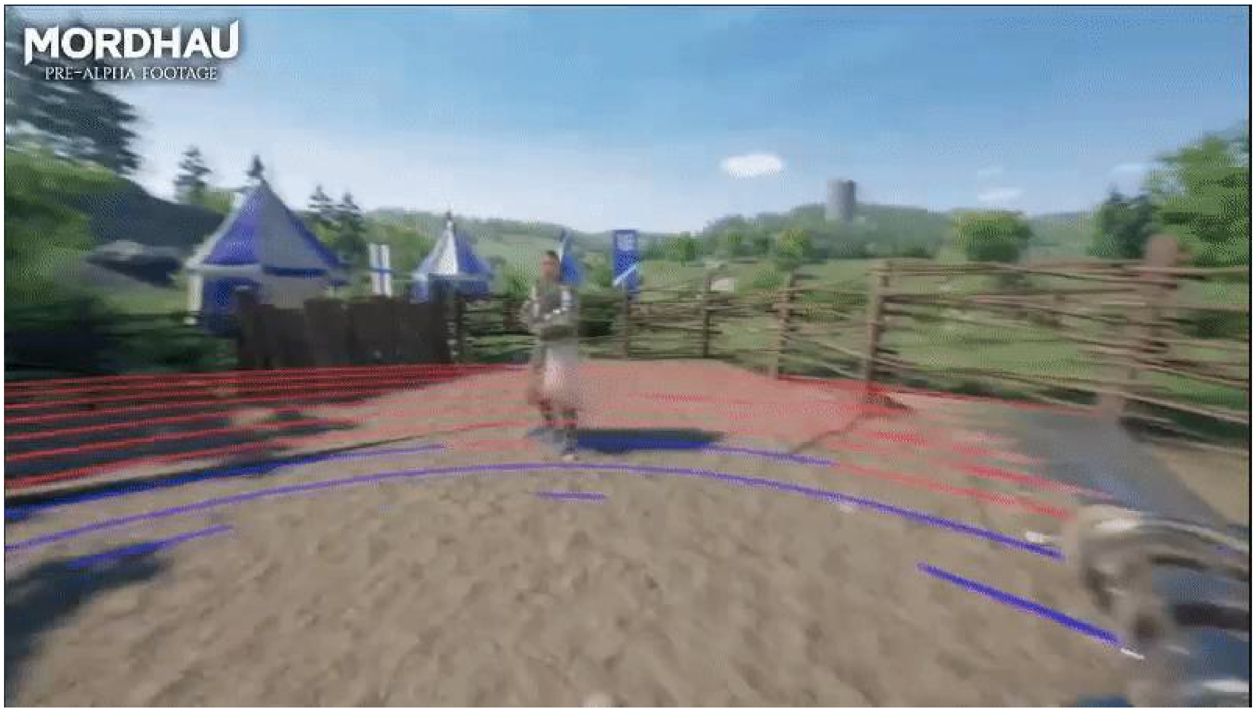
Raytracing for hits, modern implementation, first person, but can also be applied to third person. raycast allows for normal extraction and velocity of weapon.

A second game we will be looking at is ***Mordhau***, a multiplayer melee combat game not unlike ***Chivalry: Medieval Warfare***. As of writing it hasn't been released yet, but it gives us a neat look into development and specifically has some nice insights on collision detection. It also allows us to take a look at a more modern approach on melee combat.

Now that we have looked at some basic systems, we will be looking at how to actually implement the collision detection we need for our combat. Mordhau does this with multiple raycasts being positioned on different parts of the weapon. This has several advantages over just using a collider. As a first, this gives us the ability to effectively know the exact location of a specific hit. This can later help us in determining how to react to this hit depending on how or where a specific object or character was hit. As an added bonus, raycasts give us almost all the information we need to replicate an actual physical hit. Namely, we can extract the hit normal and even to a certain extent the force of the impact. This in turn all helps us to apply reactions to our characters without specifically knowing which attack-animation the character was actually performing. We will look more into this after the case studies.

A properly implemented raycast system can make your combat system fully animation-dependent without anyone ever having to add damage values to specific combos or attacks but calculate your damage based on body parts hit and the velocity and force of a specific weapon.

While Mordhau is a first person game, the techniques used in collision detection and animations are very universal and can just as easily be implemented in the fictional third person game we are discussing.



2- A screenshot showing off raytracing in Mordhau

God Of War

Animation cancelling , Slow-motion effect on hits

The third game we will be taking a look at is the ***God Of War*** series. As one of the early and hugely successful action games it is pretty straightforward that this title just couldn't be left out. With most of the basic combat stuff already being covered in the previous games however , we will now be taking a look at some more advanced tricks the developers used to make the combat feel impactful and reactive.

One of the biggest features of the combat system in God Of War is that combos can be cancelled in certain timeframes and chained into other combos . This allows you to respond immediately to enemy attacks and change your approach of the fight on the fly.

There is however a pretty big drawback to allowing animation cancels in your combat system. In ***God Of War 2*** there were roughly 4000 cancel branches , thus needing a lot of tweaking and of course manpower to create this system.

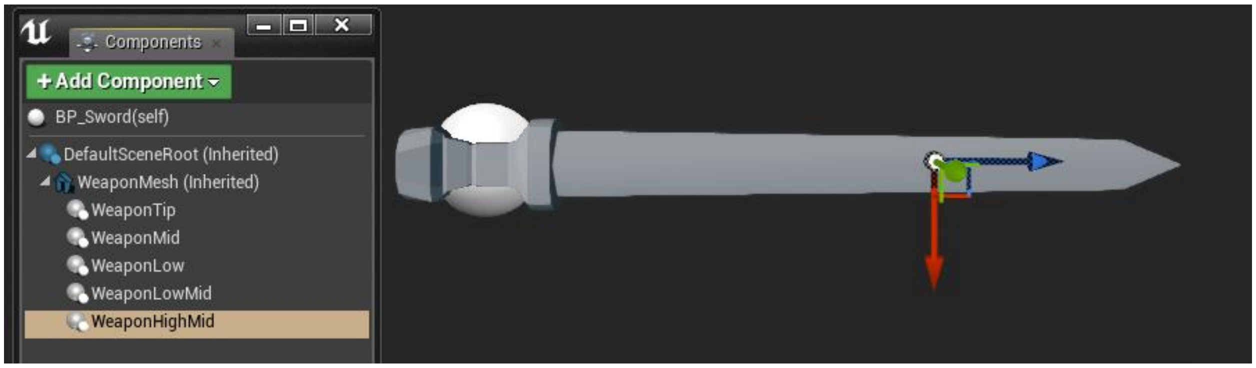
Another small feature that makes the game really feel impactful is the "hit pause" that happens whenever you hit an enemy. This is a technique that had previously been used in fighting games to make people really feel the hits. In God Of War however , the game doesn't actually pause . Rather it slows down a huge amount whenever enemies get hit, so you can really see them flying up in the air or see the blood spurting out.

Implementing a simple and reactive combat system

Now that we have taken a look at how games have implemented melee combat systems before and have a base idea of how to create a combat system , let's look at actually making this system and how to make it reactive In Unreal engine 4.

Raycasting Melee Weapons

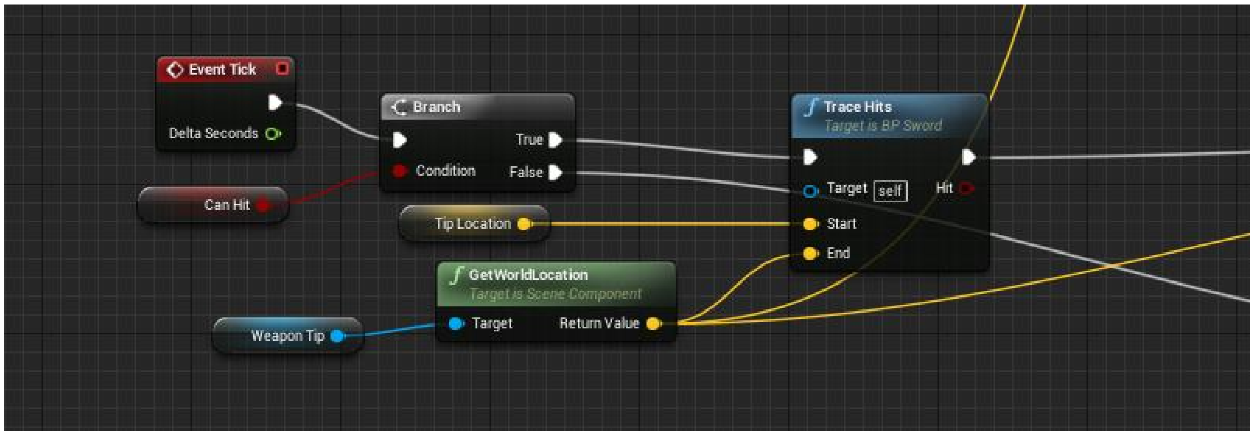
Implementing the collision detection will be the first step, and a fairly simple one at that. We will go with the way of raytracing multiple points on our sword like the developers did in Mordhau. We will start by just separating our attack logic from our actual character logic. Our character will handle our inputs and the animations that will have to be fired while our weapon will handle everything from applying damage to collision detection and reacting with the world.



Our basic weapon blueprint consist of some basic parameters such as a reference to the character it is attached to , a boolean that will decide whether the weapon should currently be registering hits or not , and some other minor stuff. For components all we really need is a static mesh. All weapons can inherit from this basic parent blueprint and contain some of their own logic.

A basic next step is the weapon itself , which contains some simple "scene" components , which are essentially just dummies to use for actually raytracing when the weapon should be colliding. Every dummy will also need a separate variable , we do this , because every frame (or every x frames , should we need to boost performance) we raytrace from the stored position to the dummy ,and afterwards store the dummy location in the variable. This way , we constantly raytrace between the previous position and the weapon's actual current position.

The raytraces will return us the necessary hits with which we can apply damage to enemies and do all the necessary interactions we need to , essentially giving us all the possible information we could need.



Whenever the character is performing an attack animation , the weapon's 'can hit' boolean will be set to true , and the raytraces will be cast. When this isn't true however , we still need to update the position variables so we do not run into any trouble when suddenly raytracing between the last attack animation's last hit frame and the new one's first hit frame.

With the basic collision detection example out of the way , let's get into how we will actually control our character's input and attacks based on animations. Animations are a very big part of these melee combat systems , luckily , Unreal Engine 4 has a very robust AnimMontage system which allows us to easily control events, parameters and blending of animations.

Now that we have our combat system thought out, let's look into actually making it "reactive". The used techniques will revolve a lot around animations , as these are usually the biggest factors in the feel of a game , together with particles and probably sounds.

Physically Based Animations

In version 4.13 Epic Games added a very interesting and extremely helpful feature to UE4 called "Physical Animation component". In short , this allows you to add rigidbodies to individual parts of your mesh's skeleton using it's physicsasset. With the rigidbodies added , you can then simulate forces on these bodies and this allows you to add forces to individual body parts while still allowing any movement and interaction to be controlled by your actual charactercontroller. The simulated physics on your character can be finely tuned and even be blended with animations or be turned on or off on the fly.

We will mostly be using this system to combine animations with physical forces when a character gets hit so it will look and feel better. You could achieve similar results by looking at how a character got hit and then applying a prerendered animation so the performance overhead would be significantly lowered. However , this would force you to make lots of specific animations and implement a whole logic system revolving around playing these animations.

With this out of the way , let's go over actually setting up a character and using this in combination with the previously discussed combat system so we can get some early results.

To get started we first need to setup a "physics profile" on our character. We will be using the third person character sample , as this will make our life a bit easier and reinventing the wheel is kind off unnecessary. First and foremost we need a physics asset on our character , but this should already be created by UE4 itself. If it's not , just right-click on your skeletal mesh and select "create -> physics asset" .

Now we can actually already start coding our physically based animations and apply it to our character in-game. But we will make an actual "physics profile" so as to better understand the way these simulations will actually work. For this , we need to open our physics asset (UE4_Mannequin_PhysicsAsset) , now create a new physics animation profile by clicking window and opening the "physics asset" window where you can add a new profile by clicking the "+" button. With the new profile selected , select all bones and click the "add to profile" button in the physical animation part of your details panel. This adds all bones to your physical animation and thus will allow all bones to controlled by physics except for the root bone as this would move your mesh.

With the bones added you can see several new values popping up where the add to profile button was. For a basic setup these values are a nice start:

-*Orientation Strength*: 1000

-*Angular Velocity Strength*: 100

-*Position Strength*: 1000

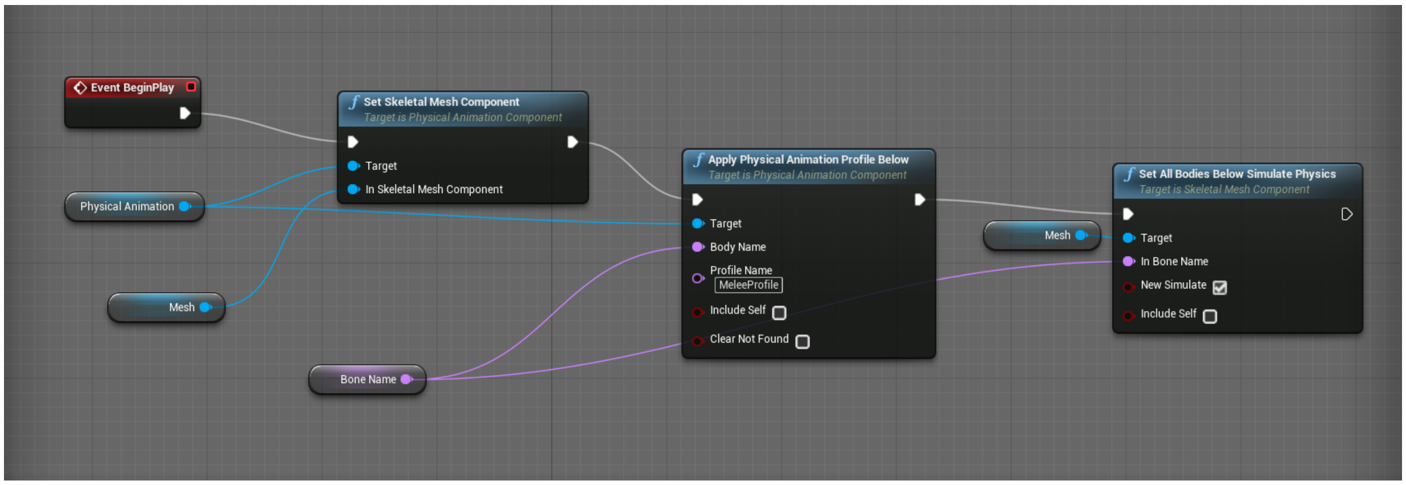
-*Velocity Strength*: 100

-*Max Linear Force*: 0

-*Max Angular Force*: 0

These values control how "stiff" your character will be , the higher the value , the more force your mesh will apply to keep its parts in the original position. For more realistic characters it is sometimes better to give the bones in the upper body and head higher values as these won't be budging as much when a person gets hit by something. Also note to turn off "Is Local Simulation" as this will make your mesh act really weird because gravity won't actually be applied.

Now that we have set up our profile and mesh it's time to actually start adding this functionality to our character blueprint. Open up the thirdpersoncharacter blueprint and start by adding a physical animation component. This component has 1 variable we have to worry about, which is the strength multiplier, the higher it is, the more our character will be affected by physics.

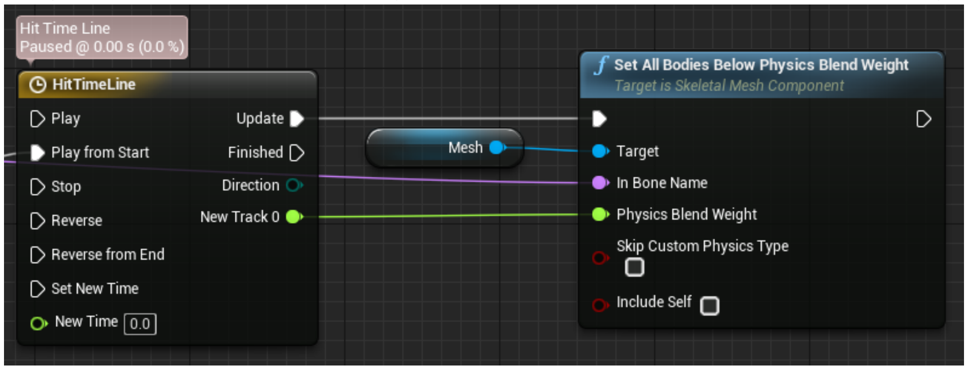

Event BeginPlay
Set Skeletal Mesh Component
Target is Physical Animation Component
Target
In Skeletal Mesh Component
Physical Animation
Mesh
Bone Name
Apply Physical Animation Profile Below
Target is Physical Animation Component
Target
Body Name
Profile Name
MeleeProfile
Include Self
Clear Not Found
Mesh
Set All Bodies Below Simulate Physics
Target is Skeletal Mesh Component
Target
In Bone Name
New Simulate
Include Self

The final step in applying our physical animations is visible in the above image. In the beginplay event, set the Physical Animation component's mesh to the character's mesh, then call the "Apply Physical Animation Profile Below" node. This node applies our profile we made (all the 1000 and 100 values on the bones) "below" a certain bone, so if we put in 'pelvis', this will apply to all of our character. And then the last node is "Set All Bodies Below Simulate Physics" which will actually start simulating physics on all these bones.

If you press play now and run around a bit you should note that, while the character still plays its animations, these are all slightly wonky and modified slightly by physics. From here on out, we essentially have all the tools we need to make our character physically react to certain events and yet still fully control the interactions and animations ourselves. For a player character , you shouldn't put the strength modifier too high as it will feel like your character is an uncontrollable octopus (unless he's octodad off course, then by all means put it at 99.999). You should also note that it is better to only apply these physics to the upper body as otherwise his feet will act weird when running up stairs etc.

If you want to make your character deliberately react to the environment, such as putting his hands on a nearby railing, or perfectly positioning his feet on stairs it will almost always be better to use an IK-based system rather try and make it work with these physical animations.

The most straightforward way to use these physical animations is to make enemies recoil when they get hit by an attack. An easy and good way to do this is to blend the recoil animation with the physical animation using a curve somewhat like a sinusoid in a timeline and outputting it's value as a blend weight. This can be done by using the "Set All Bodies Below Physics Blend Weigth" node.



Thus making your animation blend from 100% animation to 100% physics back to 100% animation over a certain period of time . The period could even be changed depending on how hard the character got hit. This will result in a smooth and realistic hit animation. It can be directly applied to the character's current animation , but it will certainly give a better effect when paired with a pretty universal recoil animation where the character for example repositions its feet to try and absorb the impact.

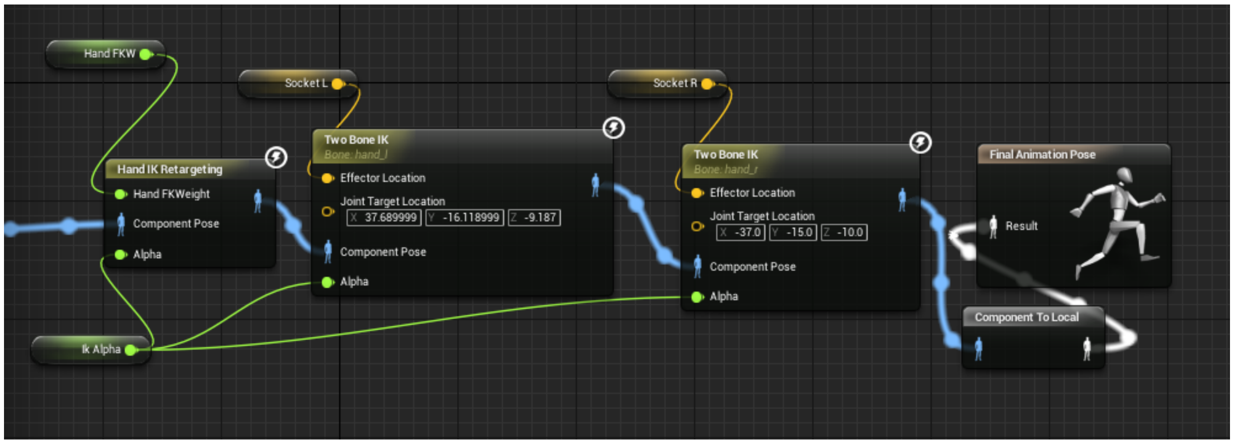
This is only a very simple example implementation. But this gives you all the necessary tools to tweak and experiment with physical animation. Combine this with some animations which can be adjusted based on weapon weight and swing stance for example and this will surely make the animations feel good and is a nice way to not have to manually create tons of animations yourself.

Reactive Inverse Kinematics

Inverse Kinematic , or IK for short is a handy way of adjusting animations by blending bones towards certain positions. As previously mentioned , it is most commonly used to make characters' feet hit stairs or to snap hands onto certain weapon positions etc.

There are however other ways in which we can implement it to improve our combat system. Seeing as we are already heavily relying on animations for damage logic and we will probably be raytracing against colliders placed on limbs and not on a simple shape such as cylinder, we can also raytrace against things such as shields that enemies may have attached to their hands. For the enemy to properly shield himself against you , we would normally either give him a very big shield so it would block almost all incoming damage from a certain direction or we would have to implement some pretty intelligent AI.

With an IK however we use a simple blocking animation and blend his hands to a certain location where we expect the player's attack to land. This way , we no longer need to give enemies extremely big shields and neither do we need to fully code this behavior in the AI itself , we could simply define this as an "action" in which the enemy will move his shield to this location with a certain speed. This still allows the player to hit if he strikes fast enough , and will not make the AI look dumb. It also allows for smaller shields and could even give the AI the ability to try and block projectiles by moving his shield to the location where the hit is expected to land.



Implementing IK itself isn't that hard. The above screenshot is an example of how a finished "state" in the AnimBP would look. You start of with a standard animation pose, after this , you just add the "Hand Ik Retargeting" Node and then you add 2 "Two Bone IK" nodes, one for each hand. The variables such as "Socket L" and "IK Alpha" are all handled in the main animBP eventgraph, they get pulled straight from your blueprint on the actual enemy by using the node "Get Pawn Owner", then casting to the specific pawn blueprint you are using and then simply extracting the needed variables.

The variables on the pawn blueprint can be updated however you like , and you can easily "turn off" your IK by simply putting the IK Alpha to 0. A basic approach to getting some "reactions" going , is by putting a collider in front of your character's mesh in the blueprint. This collider will represent the area in which your hand with the shield could move. Whenever a specific object enters this collider , you could then update this hands position to that specific location and put the IK Alpha to 1 so the hand will move there.

This way , you can get some simple reactions going , however it will feel very artificial and will not help for an actual implementation at all. One of the possible ways to do this however , would be to periodically raytrace from objects that could hit the enemy towards it. Then, when an object would be "in range", the raycast will hit this collider and you could put the enemy in a blocking animation with his shield in this specific position. Because the sword wasn't actually there yet , due to the fact that it was just a raycast signaling its imminent arrival, the enemy will have his shield perfectly positioned for the incoming blade.

Conclusion

While melee combat systems are mostly wholly complex and require a lot of resources to make, with these 3 techniques, we can relatively easily create a basic animation-based system while not having to make a lot of pre-rendered animations. Thus, saving us a lot of resources while still having combat that feels very reactive.

Melee combat also relies a lot on small tricks and effects that happen when you attack an enemy. Things such as the particle effects in Kingdom Hearts or the slowdown effect in God of War may seem very simple and superficial at first. But they really help the combat feel great and their importance should not be overlooked.

References

http://www.gamasutra.com/view/news/108166/Combat_Canceled_God_of_War__Action_Game_Design.php

https://mordhau.com/forum/topic/281/development-blog-2-melee-combat/

http://www.gamasutra.com/view/news/261698/7_combat_systems_that_every_game_designer_should_study.php

http://www.eurogamer.net/articles/2014-05-05-developing-by-the-sword

https://docs.unrealengine.com/latest/INT/Videos/PLZlv_N0_O1ga0aV9jVqJgog0VWz1cLL5f/N1tDjbFXeOo/