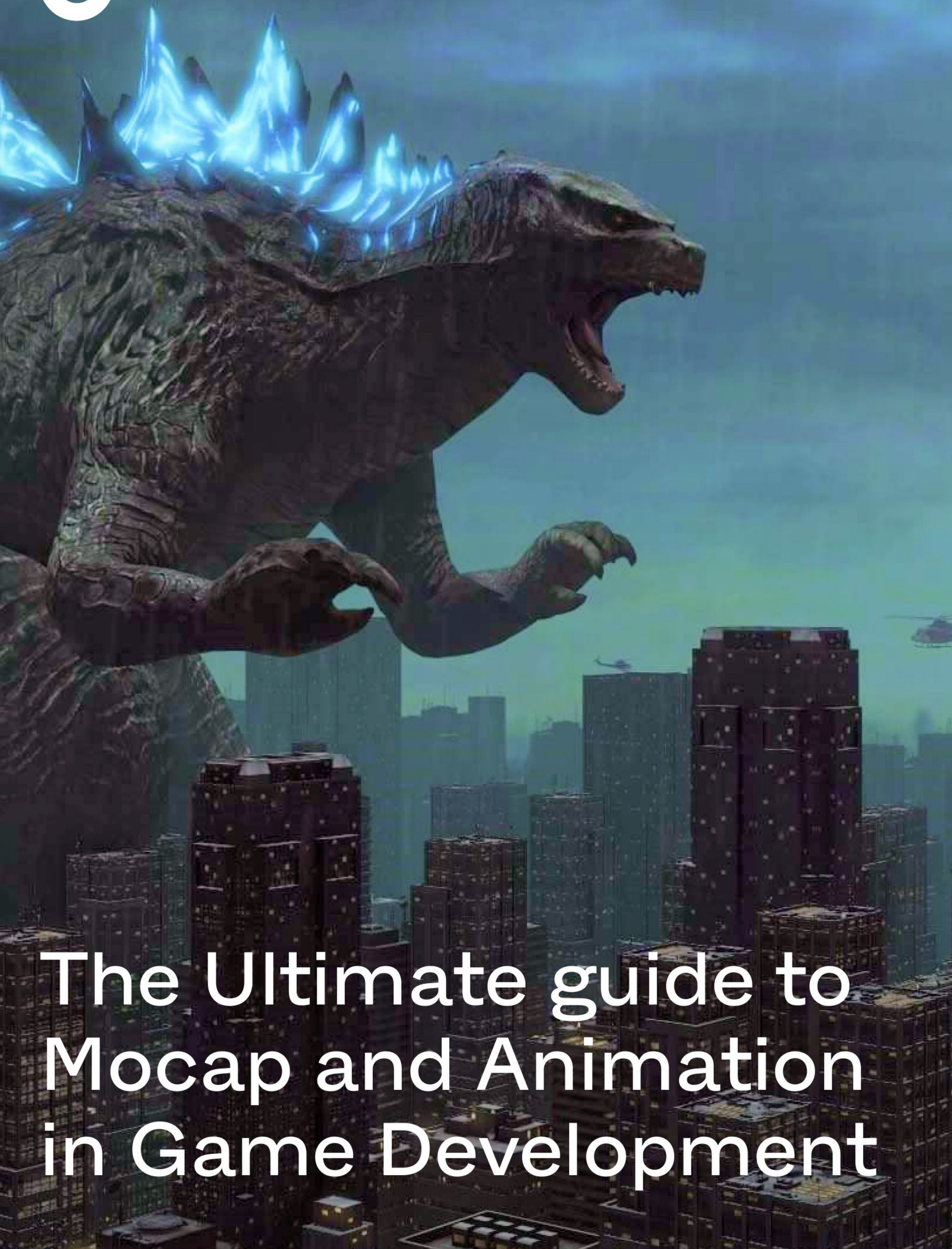
ROKOKO



The Ultimate guide to Mocap and Animation in Game Development

Introduction:

If you are new to game development and want to start doing motion capture and animation, either for your own project or in a group with more dedicated roles, then this is the document for you!

Here you will find the theoretical basis for planning your animation and mocap production, in collaboration with the game designer. We'll be exploring the basics of how animation works in game engines, how it is commonly used in a handful of games as well as some best practices for making things look and feel good.
Finally we'll round off by listing the tools and resources that are available in Unreal, Unity and some 3D rendering software to be able to implement those practices in your own game.

Be aware that this document is not for complete beginners, some knowledge of game engines and 3D rendering software is presumed. However, the resources given at the end of this document should be more than enough to get you past the initial learning hurdles!

1.0 Movement types:

When designing a game there's a few things to take into consideration with animated movement vs player control.

1.1 Animation- vs script-driven movement:

When building character movement in games, there are two general ways to do it:

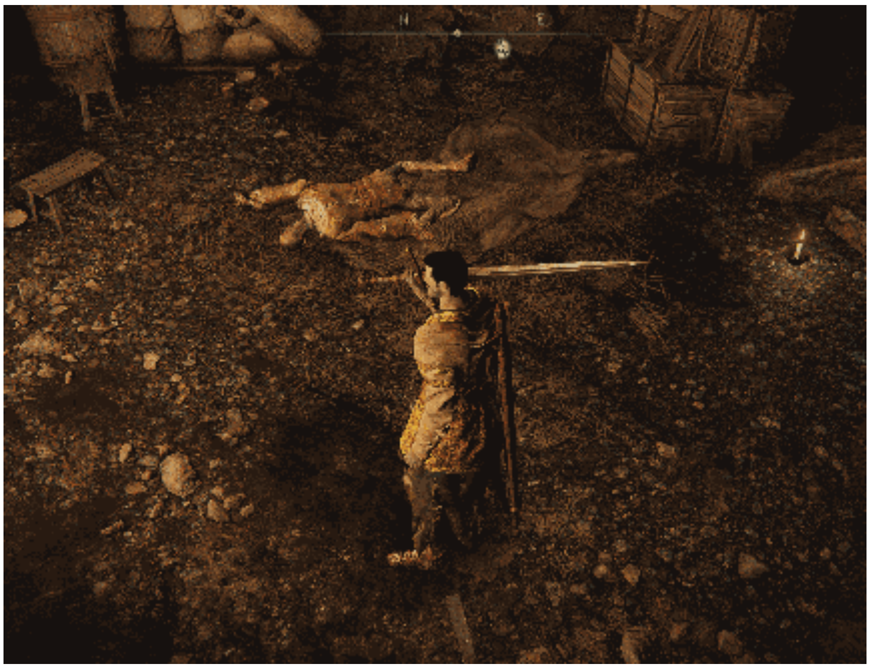
- **Animation-driven movement** means that the root motion of the animation is also what is used to move the character in the game.
- **Script-driven movement** means the root motion of the character is driven by code.

When choosing between the two, it often boils down to a question of player control vs animation fluidity. If you let the player control exactly what happens with millisecond accuracy, it may break how fluid your animations look but will let the player feel in control over what happens. If you have your animations drive the movement, the control that the player exerts is essentially to choose which animations should be played, which in turn moves the character. This can feel slower but look incredible and guarantees that the animations blend more appropriately.

1.1.1 Animation-driven movement:

Letting the animations drive the motion of the character can look amazing, but take some feeling of control away from the player. Depending on what matters in your design however, this can actually be an intentional limitation. I'll give a few practical examples:

Combat pacing:
If a game is supposed to feel slow and deliberate, using animations to drive the motions can give incredible results and make mechanics such as melee combat feel weighty and satisfying. Here is an example from Elden Ring, a game which animation-locks much of the combat to have players think more about what they do, as a button-press results in a more committed action. Here's an example of a "heavy attack" animation:



(GIF)

Notice how tightly the player's position is bound to the animation itself. This lends an incredible feel of weight to what you are doing, but also puts a great deal of responsibility on the rest of the game systems to not untimely punish players when they are not feeling in control of the character.

One way Elden Ring deals with this issue is to have modifiers in the game that change which animations are used, if certain criteria are met (eg. get stronger to swing heavy weapons better). In design terms, this is a good way to make it seem fair if players get punished, by putting the responsibility back on them; "I am not strong enough to wield this properly". They also have a system that lets players redirect an animation after committing to it, applying the input rotation to give some control over where the character ends up, which again puts responsibility back on the player by never completely ignoring their input:

(GIF)

Super-sized characters:

Outside of player-controlled characters, animation driven movement can also be really useful for massive characters. For example, in a game like Shadow of the Colossus where most of the characters are gigantic, controlling their exact motions is very important, because a misstep or weirdly blended animation would be extremely noticeable at that scale:

(GIF)

Building very large characters is generally more challenging than small ones, due to the potential for misalignment or clipping through the environment, especially if they need to react to the presence of the player. Because of that you will often see a fair amount of complex logic to determine what can be played when, as well as some degree of dynamic animation - like making sure that a hit lands in the right location without rotating the character too much and causing foot sliding. It boils down to character size, shape (and rig constraints), which animations are available for different scenarios, the speed at which they are allowed to move and which interactions they are supposed to execute. It is a dance between animation systems, AI, game mechanics and level design, to make sure everything plays nicely together. Don't let that discourage you though! The beauty in the difficulty of making this, is that the characters often feel very unique when done properly.

Non-uniform/non-linear movement:
Non-uniform motion is simply movement that is not a constant or linear speed, like a character with a limb on one leg (uneven movement). If motion is non-uniform it is often animation-driven, as that gives more control to the animator over the characteristics of whatever is moving around. Non-uniform motion is usually found on characters that have a bigger focus on a very specific game feel - like the monsters from Dead Space that move irregularly to make them harder to hit:

(GIF)

This type of movement is rarely used on player-controlled characters, as it does not correspond to the exact input that the player might give with the joystick.

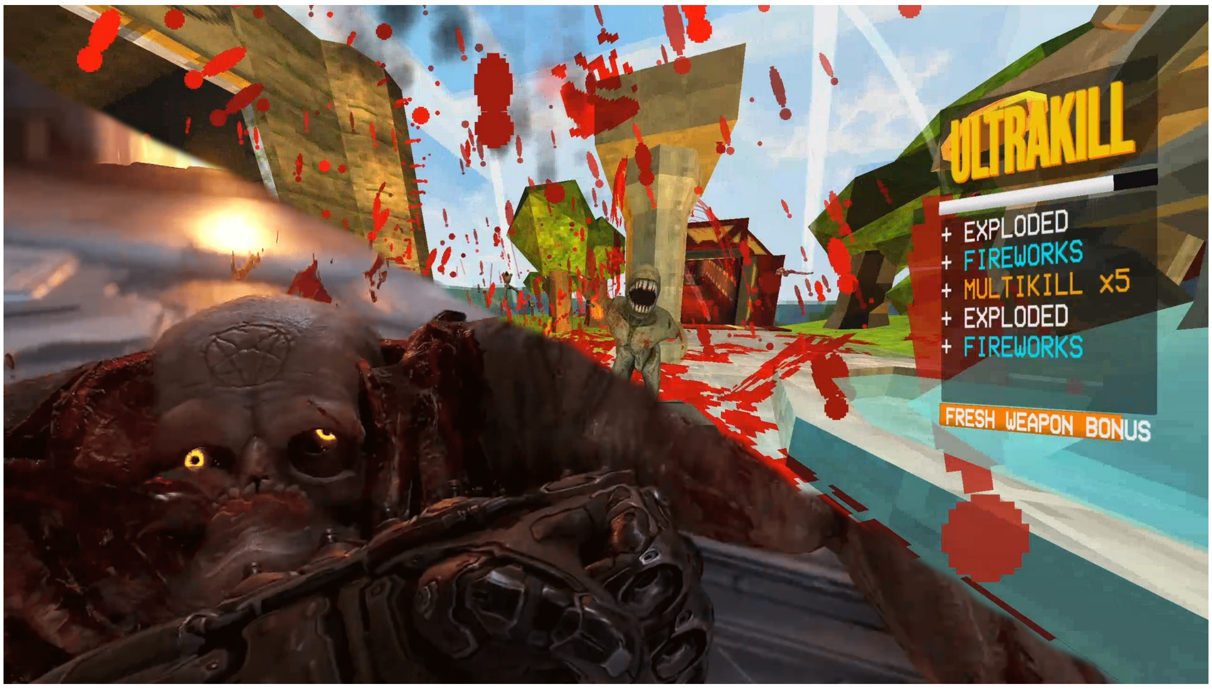
1.1.2 Script-driven movement:

Script-driven movement is used when the exact positioning and momentum of a character takes precedence to how the animation looks. This commonly involves some other layer of logic to match animations with the player input, with animation blending sometimes happening in milliseconds or less.

Competitive games in realtime:
As with physical sports, competition outcomes often boil down to miniscule factors; reaction time and split-second decision-making. In fighting games like Tekken and Smash Bros, the speed at which the players are able to counter-react to the inputs that their opponents are making should always determine the outcome, much more so than the animation speed of their characters. As such, driving the motions with scripts and forcing the animations to follow suit, is the predominant way of doing things in these scenarios.

(GIF)

First-person games:
In first-person games we only see the hands and weapons, so it makes sense to drive the motion with a script instead of animation-driven root motion. There is also a very strong argument for shooters, which are heavily reaction-driven, of input-to-action speed being a priority in the same way that it is for fighting games. This is especially true for movement shooters like Doom and Ultrakill, which actually take some cues from the fighting game genre themselves with combos and finishing animations.



Both a bit too bloody to show in motion, lol

There *are* also first-person games where the full body is shown and animated from the player's perspective, but the prevailing method still seems to be script-driven movement, as long as player reaction time is a big part of the experience.

Precision platformers:
These games are all about tricky traversal, with precision being the keyword. This genre is fairly broad, ranging from the 3D Mario games and similar, to more fast-paced interpretations such as Super Meatboy and Celeste. The key factors are timing and momentum, so having a script-driven movement system that adjusts the animations correspondingly, is almost always preferred.
This level of responsive control also lends to the formation of speedrunning communities forming around these titles. Like Super Mario 64 which came out in 1996 and is still really popular, due to how much the responsive controls allow the players to break the game in new and creative ways:

(GIF)

1.1.3 Deciding what to use (+ some design tips):

As you may have noticed by now, the above distinction between script-driven and animation-driven movement is not always so simple and will depend on your game design. Maybe a middle-ground between the two suits you better, as fast animations and snappy transitions can also be incredibly responsive for animation-driven characters. Or script-driven movement can be used to make very deliberate motions with mathematical curves that track well with the animations.

The best way to begin making a game and figuring this out is doing so in small, verifiable steps. It is likely that you already have an experience in mind and your decisions about how you can best convey this experience, should be what informs the type of movement systems you lean into. The animation systems tie heavily into the game design and you'll often find that as you iterate, the animation systems may also influence your design through discovery of limitations or new possibilities as the prototyping comes along.

1.1.3.1 Starting a Game Design Document:

If you do not have a design document, take a step back and ask yourself; *not about what game you want to make*, **but what experience you want the player to have**. Think about what defines the essence of that experience and how you can encapsulate that in the mechanics, aesthetics and narrative. One of the best ways to figure out what your design should be is to ask yourself questions. The tricky part is asking the right questions, there's a whole book about it, as a matter of fact. But being increasingly granular about it is a great way to get the basics down:

- What does it mean to be a pirate?
  Sailing, swashbuckling and drinking rum!
    - What makes sailing fun?
    - What makes swashbuckling fun?
    - What makes drinking rum fun?
        - How does drinking rum affect sailing?
        - How does drinking rum affect swashbuckling?
            - Etc. etc.

From questions like these you can begin to draft the basic mechanics, which defines the general experience.

Even if you are building a small project to experiment and mess around with, I still recommend writing a design document for practice. **The document is supposed to evolve as the design comes along and changes over time,** so don't overthink it if you really have no idea at the start - just write whatever comes to mind.

1.1.3.2 Prototyping:

Once a basic game design document has been made, it will be the basis from which you will begin building your prototypes. Prototyping is all about answering your design questions, to test if your proposed solutions are actually what solves those problems. So with the above examples, we'll probably want to build something like:
- Basic sailing mechanic
- Basic swashbuckling mechanic
- Rum mechanics

It is when you begin to build the game in smaller pieces like this, that you also discover new information like:
- Is sailing and/or swashbuckling even fun (according to your own sense of fun)?
- Does a rum mechanic add or detract from the fun?
- Would just sailing or just swashbuckling be enough?
    - Are there other ways to compensate for these "being a pirate" elements, without necessarily needing direct player control over them?

If this section feels a little vague - it is deliberately so, as all games are unique and require unique solutions. You should absolutely be inspired and build upon what you have experienced in other games, learn from the industry as a whole, but you should also **make sure that the approach you are taking is one that really fits with what *you* are building.**

If you need a good example to start thinking about your design, I recommend checking out the first Subnautica game and the adherent “post-mortem” talk about design thinking and original game design:

The Design of Subnautica (& Design Thinking)

You can also find a list of common prototype approaches here:

- https://www.uxpin.com/studio/blog/why-and-how-to-use-video-game-prototyping/

In summary; ***A prototype is an attempt to answer a design question***. It doesn’t matter how you do it, as long as you think the process provides the information you need to make a decision about that part of your design, whether that involves drawing a level on physical paper, or coding some basic logic.

2.0 Game animation principles:

Now that we have started thinking about the user experience and game feel, we’ll have to begin thinking about the ways we can practically achieve those things. One of the best ways to do this is by understanding the principles which govern animation in games.

Now you might be thinking “isn’t mocap supposed to alleviate these problems?” To which I say yes it is, and it does! But understanding how you can effectively record and optimise your animations for specific use cases, not only enhances the look and feel of your final results, but allows you to use mocap beyond what you can do with regular human motion.

What I’ll be writing in this section is largely a breakdown of the animation theory principles by Jonathan Cooper, described in his Game Anim book. A summary of these principles can also be found depicted in this playlist by Video Game Animation Study, which I recommend giving a watch:

- The Five Fundamentals of Video Game Animation - YouTube

2.1 Feel

We've briefly touched on this in some of the examples of animation- vs script-driven movement, as it boils down to how in control the player feels of their character. It is very closely tied to the game design itself, as the game's mechanics and animation systems must work together to convey the game feel. It is one of the trickier parts to teach as it relies on actually implementing your animations, playing the prototypes and *then* evaluating if it works or not. The evaluation itself is highly subjective, so you will often find yourself in a situation where the test is to rely on your own impressions to verify if you've succeeded, then doing tests with other people and asking them questions to figure out if they feel the same way. This might sound daunting, but odds are if you've played a bunch of video games similar to what you are trying to achieve, you'll already have a pretty good instinct for when something works or not.

Asking questions:
Throughout development, think of "Feel" as a lens through which you are examining your game and prototypes, by asking yourself questions like:

- "Does this add to the player feeling in control?"
- "Does this movement convey the impact I want it to have?"
- "Does this align with the fantasy of playing the character we are seeing on screen?"

You'll find that saying yes or no to very specific questions like this, should lead to insights about "why":

- **Yes:** Why does that work and how can we use this knowledge to strengthen other areas of the game (eg. build to your strengths)?
- **No:** Why does this not work and can we make changes to make it work?

If you have a hard time coming up with questions, try to boil it down to something as simple as possible, which reinforces whether or not the thing you are examining is in line with the intended experience.

Form over format:
You'll generally find that form over format is key here; *the feeling of control always takes precedence over the look of the animation.* But the feeling of control depends on the context - how does everything else in the game move relative to the player? Do non-player characters also lock themselves into attack animations and can be punished for committing

to the wrong moves? There is plenty of room for long and complicated animations, but the entire experience must be able to accommodate for this, or they can detract from the game feel if they lock players into unwanted positions. But deliberately base the whole experience around the same rules of movement and then it's fair game.
There are no objectively right or wrong ways to animate your characters, only that the animations must play well with the design.

As you become more experienced with game development, mocap and animation, you can make more reliable assumptions about what probably feels good and in line with your design. However, you should always have someone else test and confirm or deny your assumptions, no matter how confident you feel. Someone who does not have your vision of the complete experience, might feel something very different from what you do.

2.2 Fluidity

Fluidity is about how well your animations are blending together. In essence, whenever a character performs an action, they will have to transition to that from whatever they were previously doing. If you do not do this transition well, the character will visibly "snap" to the new animation - which can easily ruin the immersion.
Unlike Feel, Fluidity is something we can be very systematic about. The fluidity of a character depends on how many states the character can be in, each state with individual animations.

Explaining states:
The "state" of a character is literally what they are doing, such as:
- Idle standing
- Walking forward
- Running forward
- Running forward left
- Running forward right
- Running backward

Etc. etc.

Each of these states represent something that the player can order the character to do. But each of these animations do not necessarily blend well together if we transition between them. To address this issue, it is common to have **state transitions**, indicating the transition between one state to another. And each of these state transitions can have a corresponding animation associated with them - acting as an in-between the two states.

Like a 180 degree turn from idle to run (GIF)

What we use to build and run all of this logic is called a **State Machine**. For more on how to build these in your preferred engine, see section **3.3 Animation State Machines** below!

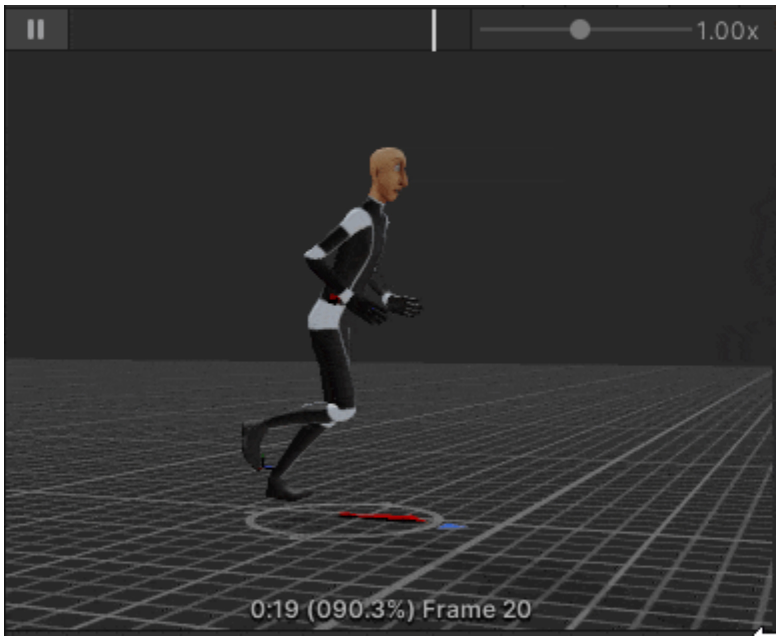
2.2.1 Loops, Links and Linears:

When talking about fluidity of animations, it is helpful to think of them in three categories; Loops, Links and Linears.

2.2.1.1 Loops:

Loops are repeating animations used for movements like walking, running, idle etc. anything that is continuous.

Looping animations and timing:
Whenever we have looping animations, it can be useful to do a little bit of planning when we record and edit, to synchronise the timing on our same-speed animations, like walking in eight directions. The cadence of the walk in all directions should always be the same - meaning each of these animations should have the same length and start on the same foot, or match as closely as possible. This allows for transitioning between most of the walking animations without needing link animations between them.
By keeping the loops as short as possible, we also simplify any logic we might attach to these animations later, like blending in and out of foot-to-surface alignment:



Loops about as simple as this (GIF)

2.2.1.2 Links:

Link animations are used to smoothen out blends between animation states, which otherwise do not blend well together. It is the link type of animation that can get a little bit out of hand if you try to account for every possible transition, so be careful not to overdo this. Here is an example of a link animation from Remedy's Control to illustrate:

(GIF)

What you see here is:

Idle → Short link → Run loop → Slightly longer link → Idle

Planning link animations:
Link animations should be considered a form of polish, as they are usually not a requirement for your basic game mechanics to work. If you do not have a clear picture of which transitions are the most important for your game feel, it is better that you minimise (or even omit) link animations in your first prototype. Play around with what you have initially and try to identify which links would be the most impactful to the feel of the game (eg. where do the transitions between the other animations feel the worst).

Iterating with link animations:
Once you begin to implement link animations in your game, if your character movement is mostly animation driven, this will have an impact on the game feel. You might find that controlling your character feels more sluggish if you have to go through the link animation first, for example. I recommend playing with the speed of the link animation to address this. You will often find that a character animating really quickly to go between states can still feel more natural and fluid in their movement, than one transitioning at a more realistic pace. Or maybe your link animation only affects some parts of the character, so the legs and root motion are unaffected.

Issues and alternatives:
A big problem with having a lot of link animations is that they have to be updated if you ever make changes to your other animations, to make sure they all still align. As such, some crafty developers actually build their final games to use very few or no link animations at all, but instead have some logic to compensate. A great example of this can be seen here, as Gwen Frey breaks down a rig they used for prototyping at The Molasses Flood:

- Alternative Biped Solutions

2.2.1.3 Linears:

Linear animations are the "one-shot" movements that play once from start to finish, like a jump or an attack animation. These usually play from a single button-press and represent concise actions, which should happen within milliseconds of that button being pressed. Because of this need for responsiveness, blending to these animations usually has to happen very quickly.

Like when pressing jump (GIF)

Blending w. linear animations:

We have to be very mindful of how fast a linear animation plays and what it imposes on the players. Tying back into game feel, this really depends on your design and intended experience, whether you want players to commit to an action or always have millisecond-responsive control regardless of context.

Or a longer windup for a more powerful jump feel (GIF)

Or maybe you don't want the linear animation to affect the root motion at all, or only play on part of the body while another animation or script is driving the root motion. Like swinging a weapon while running as a combination of two animations - one for the legs and root motion and one for the upper body.

Dynamic animation control:

One problem with linear animation is that it is usually devoid of whatever context it is used within. You might be swinging a punch, but it goes through anything it hits - or it is wildly misaligned with what you are swinging at. It is at this point that we have to think a bit outside of the box and look at what tools are available for us to dynamically modify the animations, so they fit within the context that we use them in.

For a game like Dark Souls for example, they use a system called “hit-stop” which stops your weapon animation for just a couple of milliseconds when impacting an enemy, to simulate the weapon passing through the mass of the enemy. They differentiate the impact based on some parameters, like the mass of the enemy vs the weight of the weapon, strength of the player etc. so the animation will pause on heavy enemies, play like normal on light enemies or even launch the enemy if the weapon mass and player strength are proportionally large enough.

For other games where adjusting the pose to connect with a very specific point, blending with Inverse Kinematics (IK) can be super useful. I’ll illustrate this in section **3.1.2 Inverse Kinematics**, but the gist is that we can dynamically modify a single animation to connect with a target regardless of its position.

(GIF)

Once we start thinking about our animations in smaller pieces like this; that we can target specific body parts and how they can be dynamically moved with IK, it becomes easier to think of ways that we can create diverse interactions from the same animations, to fit in any possible scenario the players might put themselves in.

2.3 Readability:

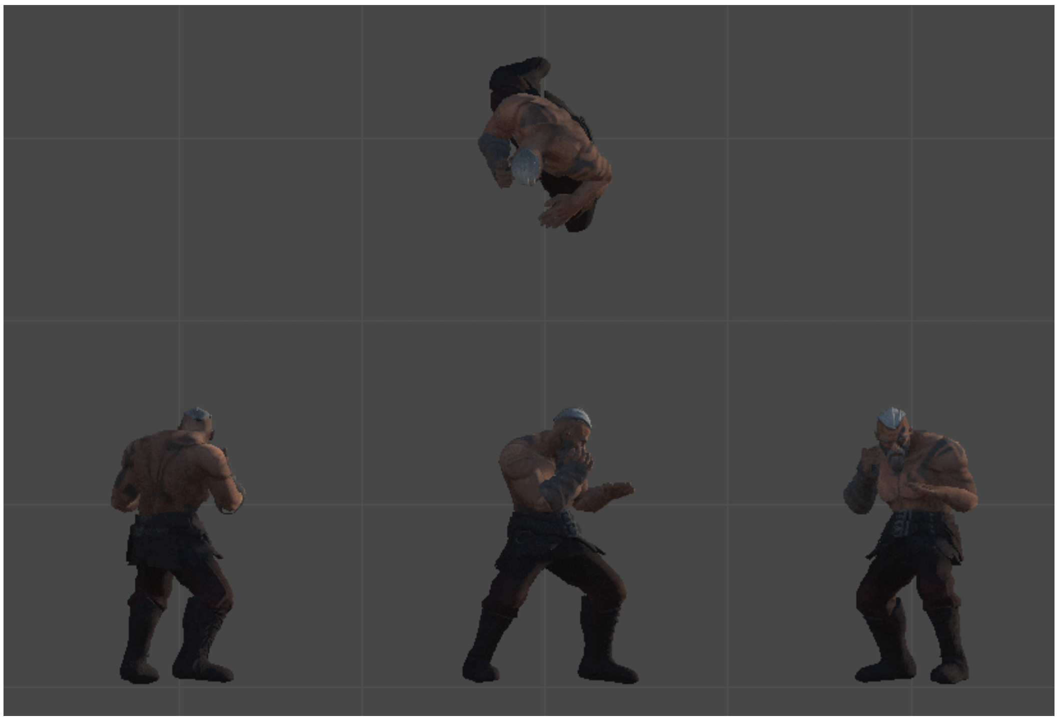
Readability is about making our animations look good from all possible angles. Unlike traditional animation where we control the position of the camera, most games let people view the characters from all kinds of angles, which makes it important that all actions (particularly linear animations) perform some motion on all axes.
This element is crucial when you are actually doing your motion capture, so making sure to preview your captures with this perspective in mind and doing iterations with the actor, will greatly improve the usability of the resulting animations.

Bringing X, Y and Z in motion:
To take the above statement into practice, whenever you have done an animation clip you should always preview it from:

- The front
- The side
- The back
- The top (if applicable - most games are not directly top-down)

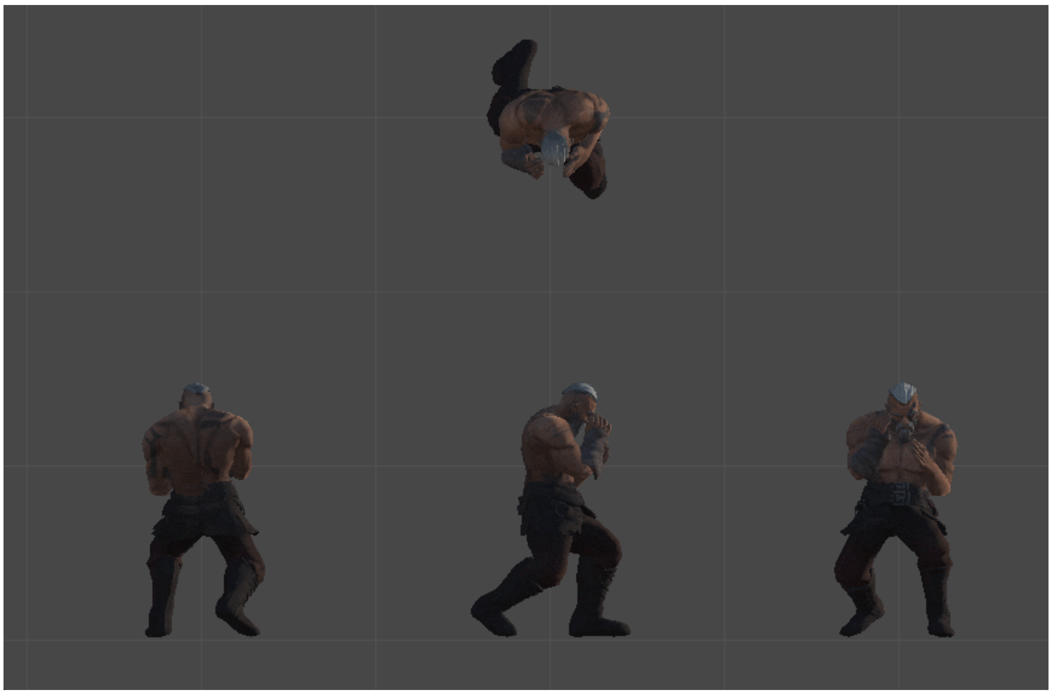
What you should look for is if it is easy to tell what action the character is doing. Though keep in mind that if your game is always seen from certain angles (like always from the side), the animation should look its best from those angles and less so (or not at all) from the others.

For comparison, consider these two animations in multiple preview angles:



(GIF)

Notice particularly on the back view, this move can be hard to read. If the game is mostly viewed from the back, a different animation with a wider motion on the axis across the shoulders is preferable:



(GIF)

This wider swing has a lot more motion on the Z-axis from behind, but the designer might note that it “feels a bit less weighty” because the mass of the body is not behind the swing anymore, and then you can start iterating from there. Maybe the solution is to lean a bit more with the upper body etc.

When iterating, it helps to think about the three axes that a 3D character can move in; X, Y and Z. If an animation looks stiff or unreadable from an angle - odds are it does not have enough movement along one or more of these axes. **A good mocap actor is someone with a good understanding of these dynamics and can minimise the number of iterations.**

Camera distance and acting:
Camera distance plays a role in games as much as it does traditional animation and film. It is the reason why early movies have a certain vibe, as actors that were trained to perform for a crowd that would potentially be sitting far away from the stage. People in the back should also be able to tell what was going on, so they had learned to move and talk in exaggerated ways because that was what worked at a longer distance, but feel unusual (albeit charming) when a camera is moved up close:

(GIF)

This especially notable when compared to modern actors being trained for subtlety and closeups - but subtlety that would never be visible from the back rows in a traditional theatre.

Camera distance is just as important for motion capture and animation in games. If you are performing for a context where you know the camera is always far away or up close, this should be portrayed in the performance. A character that is very small on the screen should have exaggerated movements, whereas someone in a closeup should really be careful not to over-act.

2.4 Context:

Context is about how well the animations convey the personality of the character and their role in the game. Think about whether the movements being portrayed feel in line with the character’s visual design and personality description, as well as whether or not the animations are for a unique character or for repeated use on less distinctive non-player characters.

Distinction vs homogeneity:
The concept of differentiating between character animations based on how unique they are, is what Jonathan Cooper calls “distinction vs homogeneity”; which means differentiating if the animations apply to a distinct character or to an assortment of similar characters (main character vs a group of pedestrians, for example).

If you have more than one unique character, particularly if they are playable, they might be able to perform the same actions but should not have the same animations for those.
But for characters that are physically similar and appear in multiples at the same time, you’ll instead want to create sets of animations that can be reused for all characters in that group.

Animation layers, additive and dynamic animation:
One problem that tends to happen when reusing animations for similar characters, is that it becomes easy to tell that they move the same. This is especially true if two characters not only play the same animation, but do so with similar timing. Our brains are extremely good at spotting synchronised movements, so this is guaranteed to break the player’s immersion if it happens.

To solve this problem, we’ll resort to using layers of animations on top of a baseline animation, to generate variety and avoid the synchronisation issue. We’ve briefly touched upon this in the **Fluidity** section where I mentioned dynamic animation adjustments and targeting specific body parts. We can use that exact same system and approach to generate subtle diversity and avoid this issue:

(GIF)

From left to right:

1. The original, unmodified animation
2. The original animation + a layer of a separate animation on the limbs and head
3. Original + layer + dynamic look direction with inverse kinematics

Informing the mocap actor:
Being in the position of animator, game designer or both, it is very important that you are able to convey mannerisms, character rig limitations and potentially context to the actor, preferably with visual examples (clips from other games or even just you mimicking the motion yourself). It can also be very helpful to have a clear view of the rigged character during the capture, so the actor understands how their movements are manifested. Much can be cleaned up and animated after a session, but the closer you are to the intended result already in your mocap stages, the shorter the time to do the cleanup and editing afterwards.

2.5 Elegance:

Elegance is all about the usability of your animations, to simplify your game systems and reduce how many animations are really needed to achieve your goals. Achieving elegance requires collaboration between the animator and game designer, because the things that have to be simplified are not the animations themselves but rather the contexts in which they are used.

Unifying scale for reusability:
What unified scale means is that you design your game environment and interactions to fit with a handful of specific animations, to limit how many different animations are needed.

For example, if you make a platform game that has a ledge grab animation and/or very specific jump heights, then all of your game's environments will benefit from sticking to the same platform and ledge heights. Or if you have animations for handling a certain type and size of an object, designing more objects that fit within that scale can allow for those animations to be reused.
A hidden benefit of this form of standardisation is that it can increase the clarity of your game design. If all platforms stick to certain height levels relative to where the player can jump from, it becomes much easier for the player to understand if a platform is reachable or not.

Prioritising what matters:
In the end, whether you choose to spend your time creating a lot of animations for specific contexts or unifying your game environments to reuse animations (or perhaps build more dynamic solutions), is up to you:

- It may be down to the game designer to figure out what has the biggest impact on the experience, then direct the mocap and animation efforts from there.
- It may be the animator with a very concise budget that evaluates how they can most effectively do their job, which in turn informs the game design.

I highly encourage this form of communication between departments. Or if you are a solo developer or happen to be both a designer and animator, then keep asking yourself if you are really making game design and animations work together in the most efficient way possible, to achieve the desired experience. The more each side understands each other and the intended vision for the experience, the more often you will find opportunities to mocap, animate and design efficiently.

3.0 Common tools and techniques:

Moving past the theory and planning, we'll need to know what tools we have at our disposal to actually implement these principles. I'll generally keep this within the scope of Unity and Unreal, with a focus on explaining how these tools tie into the animation principles.

3.1 Inverse Kinematics vs Forward Kinematics:

Inverse Kinematics (IK) and Forward Kinematics (FK) are the two primary ways of computing character animation. In many scenarios you will see character rigs that use both methods interchangeably to achieve the best results.

If you are new to the concepts of IK and FK, I recommend watching the first 3 minutes of this video:
FK and IK Explained - Which One to Use and When?

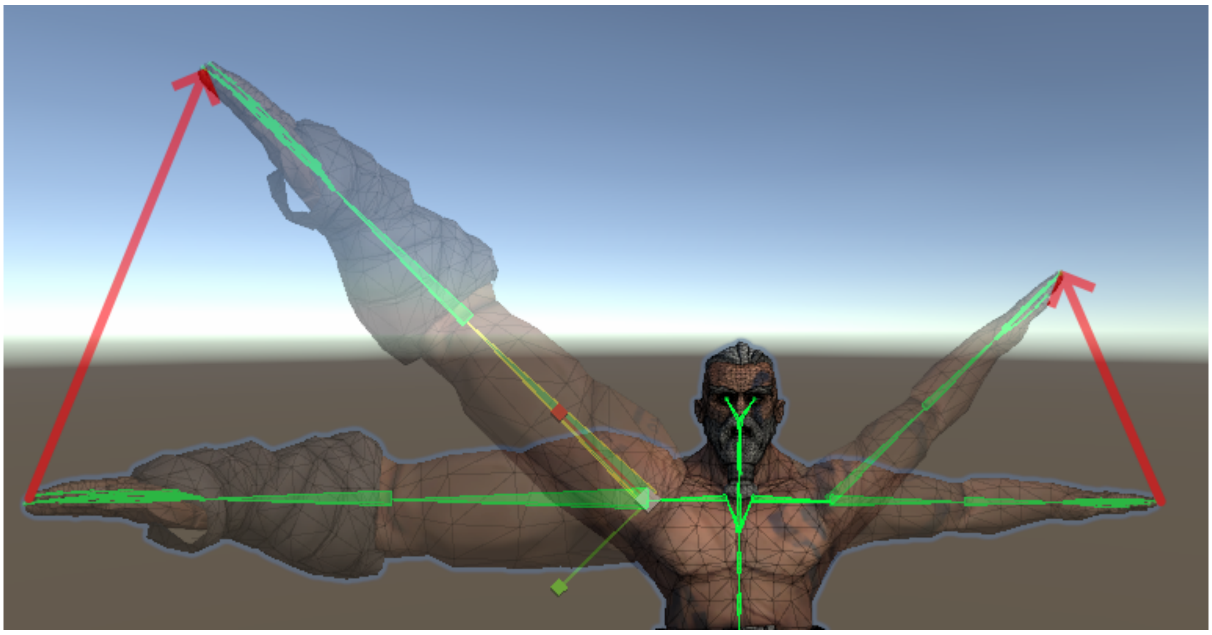
In short terms:

- FK plays back the original animation, by rotating the bones just as they rotate in the animation clip.
- IK partly or completely overrides the original animation, to instead move a limb or bone to a specific target.

3.1.1 Forward Kinematics (FK):

Since forward kinematics is the exact playback of the bone rotations from the animation file, this is great for situations where characters are the same proportions and need to move to those exact specifications. However, there is one fundamental issue with relying on forward kinematics for certain things:

The problem with different bone lengths:
If you rotate a short bone by some amount, vs a long bone by the same amount, the tip of the bone will move different distances. This becomes especially clear if we look at a side-by-side comparison like this one:



The red arrow indicates the length that the tip of the limb has travelled, which is very different for the longer bones. This is despite the fact that both of these arm bones were rotated exactly 45 degrees in this picture.
In practice that means limb alignment becomes difficult, especially when it comes to natural movement of feet. This is due to the fact that the root motion remains unchanged, despite the feet now technically moving a longer or shorter distance, causing the feet to slide as the character walks, as the root motion does not accommodate for this difference. Interestingly, if we retarget to a character that has the exact same bone lengths as the original, we can still scale it up or down however much we want, provided that all bones and root motion are equally scaled as well.

3.1.1.1 When to use FK:

With FK being fundamental to how animation works, the question should really be "*when to use other solutions on top of FK?*" The reason I feel that it is important to highlight the nature of this system, is to dispel the idea of "my animation is just bad" or that something needs to be reshot for a bunch of different contexts. You will often find that you simply need to combine the animation with other animations, either additively with animation layers or more dynamically driven with inverse kinematics, to modify your results.

Additive animation:
As I mentioned when talking about the animation principle of context, additive animation is when we take one set of rotations (our baseline animation) and add another set of rotations

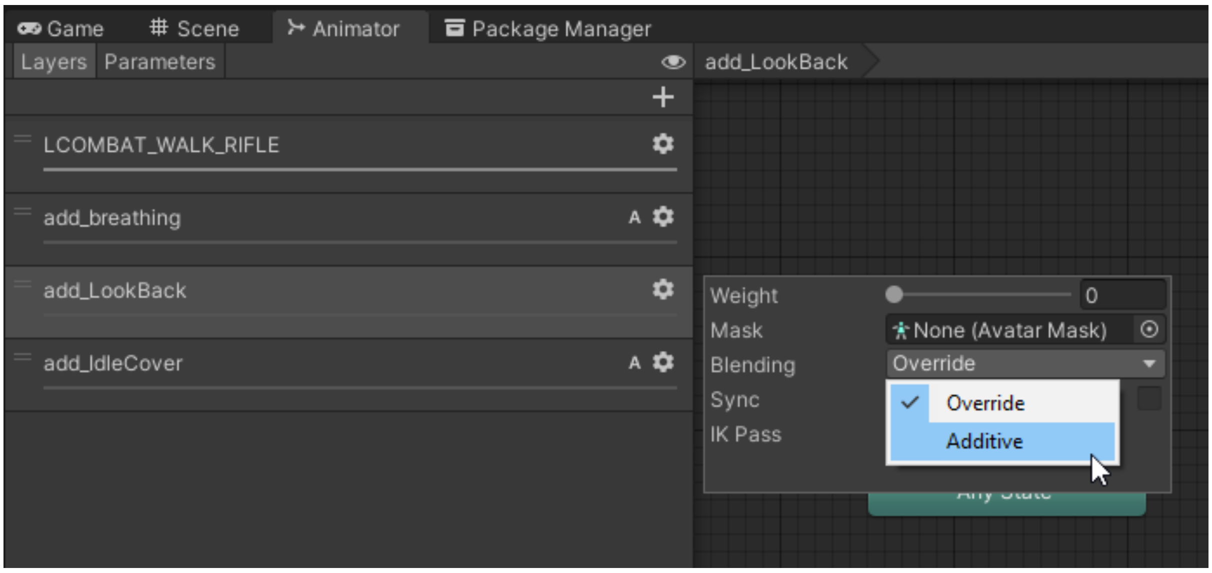
to it. This is done in the most literal sense of the word, by taking one rotation plus the other before updating the rotation of the bone. So the result is calculated as:

Rotation from clip 1 + rotation from clip 2 = rotation applied to character.

A great illustration of this can be seen in a tech demo from the original Crysis game, where you can see what the added animation looks like before it gets added, as well as the result of the addition:

- Crysis 2006 Additive Animation Example

What you see here is just like having an additive animation layer in Unity or Unreal, which they increasingly blend into the basic animation:



As shown above, Unity also has a weight property which actually makes the calculation:

Base rotations + (added rotations * weight) = final rotations.

This allows for more dynamic fine-tuning of how much the additive animation is added to the final result, as well as allowing for a smoother blend-in by not immediately adding 100% but doing a quick increase of the weight from 0 to 1, making it look more natural.

Note how the additive animations in Crysis also do not look like something you would do with motion capture but rather by hand, due to this way of direct addition. So while this approach is a great way to create minute changes in your mocap - a better way to do this entirely with mocap is by interpolation instead of addition.

Blending animation with interpolation and weights:
Interpolation is when we find a value between two other values, by some percentage (the weight). The maths behind this is a bit more complicated than adding animations together, but what you really need to know is just that this is a way to blend between two clips, with only the original clip playing at weight 0 and only the added animation playing at weight 1. So in practice:

- **Weight 0 = 100% original clip**
- **Weight 0.50 = 50% original clip, 50% extra clip**
- **Weight 1 = 100% extra clip**

More complicated forms of animation blending can also be done, such as blending between three different clips with Unity's Blend Tree system: Unity Manual: Blend Trees

Likewise, Unreal's animation blueprints has systems such as the Blend Spaces that allow for creatively mixing animations together:Blend Spaces Overview | Unreal Engine Documentation

Or on a simpler level mix two animations together with the Blend Node, in which case the "weight" that we see in Unity is the Alpha channel on the node: Blend Nodes | Unreal Engine Documentation

3.1.2 Inverse Kinematics (IK):

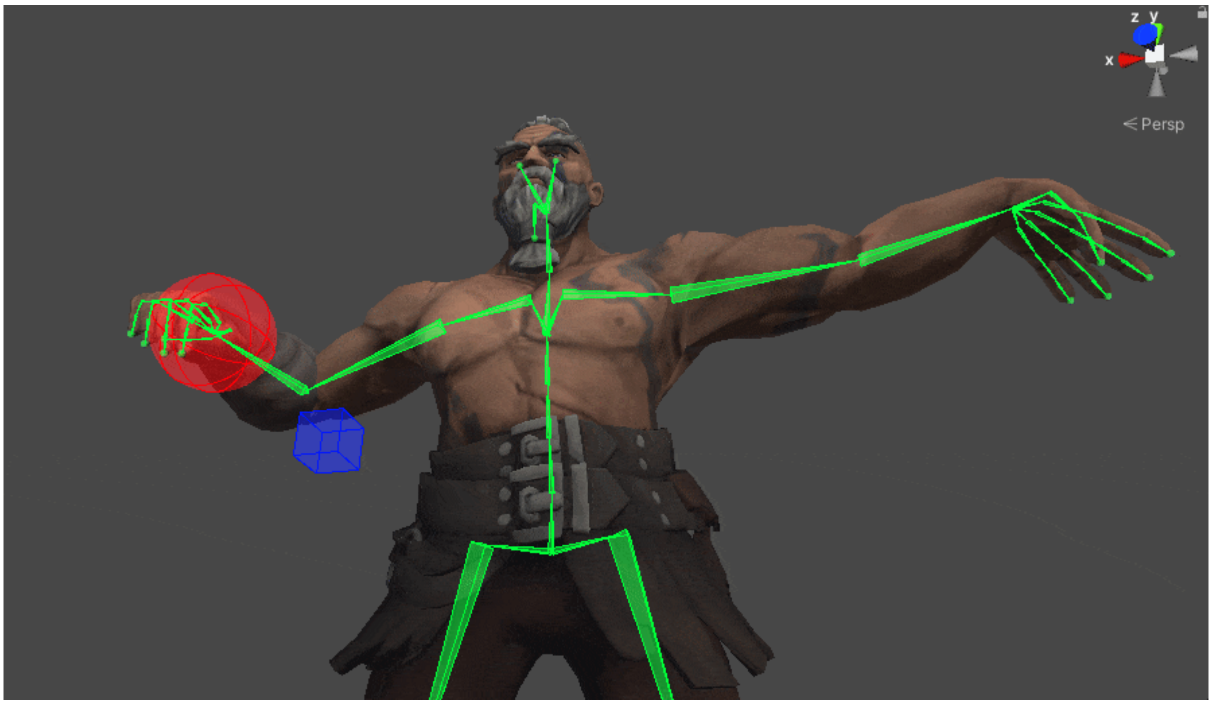
While Forward Kinematics is all about applying the rotations from the animation clip, Inverse Kinematics is all about modifying the pose of the skeleton to align with a specific target. This is what we generally use for dynamically modifying our animations to fit with external parameters, such as aligning limbs with surfaces or making hit animations that correspond more accurately to where a character was impacted.

Using Inverse Kinematics generally boils down to these four things:

1. A bone chain with a tip and a tail bone, which determines how much of the skeleton is driven affected.
2. An "IK Target" which is the position/rotation that the tip of the chain is trying to reach.
3. One or more "IK effectors" which pull the chain towards them, useful for making sure a leg or arm does not bend the wrong way, for example.

4. A blend value/weight that determines to what degree the Inverse Kinematics is allowed to move the bone chain vs how much it will be driven by the original Forward Kinematics.

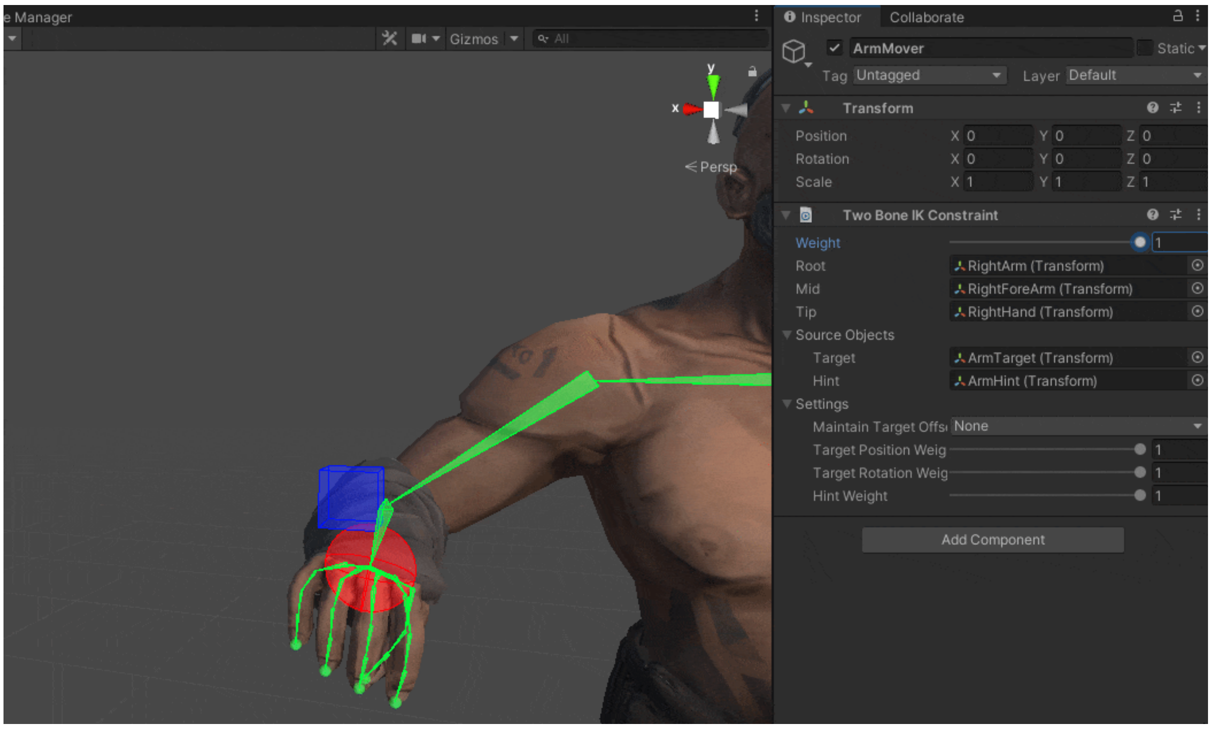
A simple illustration of this can be seen here in Unity, as I kindly asked our Viking character to idle in a T-pose:



(GIF)

- The red ball is the **IK target**, which I posed in front of the character.
- The blue cube is the **IK effector**. Though you can see it rendered through, it is behind the elbow.
- The **IK chain** has the arm bone as the root and the hand bone as the tip.

By blending in and out of the weight, which determines how much the IK drives the pose vs the original animation clip, we can go back and forth between the IK target and the original animation:



```
ArmMover                                   Static
Tag Untagged            Layer Default

Transform
Position        X 0     Y 0     Z 0
Rotation        X 0     Y 0     Z 0
Scale           X 1     Y 1     Z 1

Two Bone IK Constraint
Weight                              1
Root        RightArm (Transform)
Mid         RightForeArm (Transform)
Tip         RightHand (Transform)
Source Objects
  Target    ArmTarget (Transform)
  Hint      ArmHint (Transform)
Settings
  Maintain Target Offs  None
  Target Position Weig              1
  Target Rotation Weig              1
  Hint Weight                       1
```

([GIF](#))

Hand/feet surface alignment:

With the above in mind, consider the fact that while you are not blending into your IK chain, you can literally snap the IK target to wherever you'd like, then blend in to make the tip of the chain meet the position of the target. In other words:

If you put the target on a wall and blend into the IK chain, the hand/foot/whatever you are moving, will perfectly align with that surface, as long as it is physically within reach of the character.

The tricky part comes when determining when and where to put these IK targets and when to blend in and out of the IK chain's control. Often people will be raycasting into the environment to figure out where to place the IK targets and align it with the surfaces, then adding animation curves on top of the movement animations that blend the IK in/out as the feet are supposed to connect with the ground, for example.

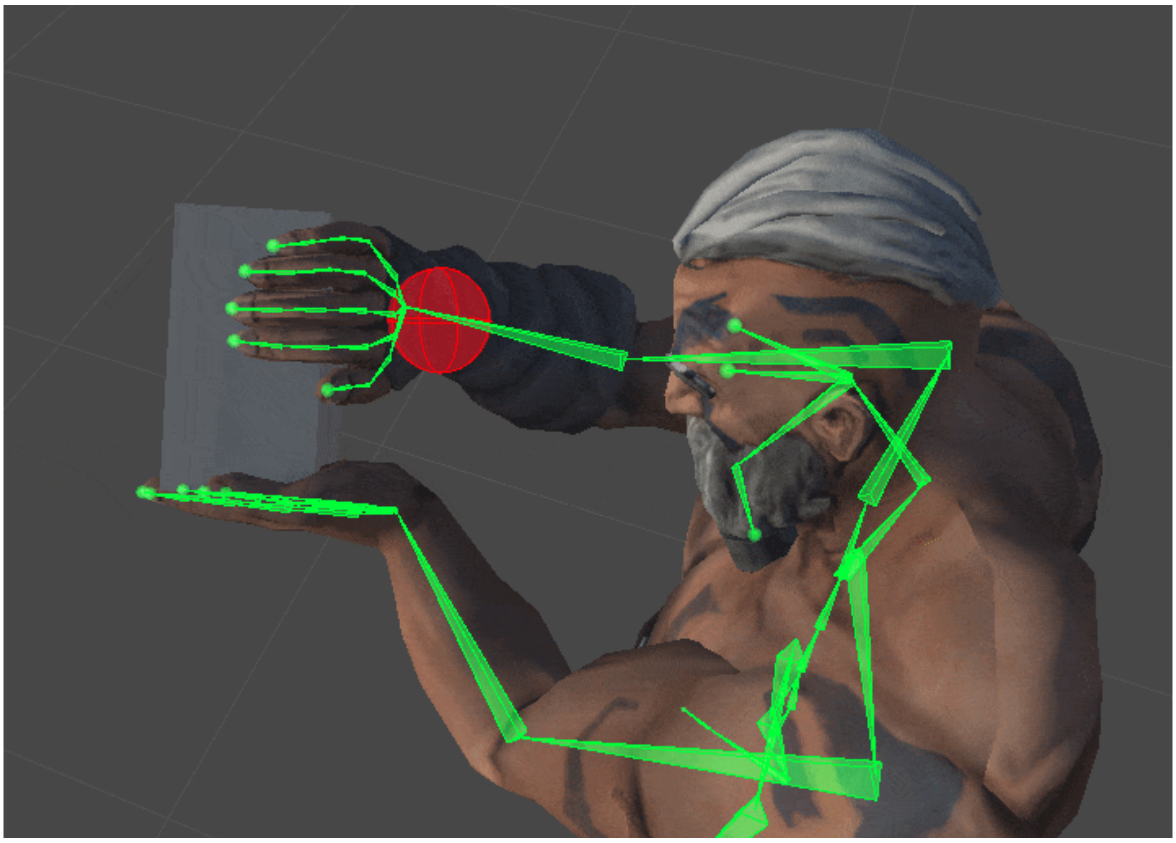
Holding objects:

Using IK chains is also a great way to do interactions such as holding an item with two hands. This can easily be done by:

1. Parenting the item to the hand that drives the motion of the interaction (eg. if you are pointing the object at something, use the hand that does the pointing).
2. Set up an IK chain on the other arm

3. Parent the IK target to the held object, at the point where the other hand should hold onto the object.

Then you can do interactions like holding onto this gun-... cube-thing:



(GIF)

And just to illustrate how robust this approach is, here is the same animation with an additional cube, and the IK target in a different location on the object (I hid the rig just for visual clarity):

(GIF)

3.2 Rigging tools:

Without getting into the step-by-step, both Unreal and Unity have really robust rigging tools to achieve what is showcased above.

3.2.1 Unity's Animation Rigging:

All of the custom examples that were shown in this document were using Unity's Animation Rigging system. To understand how this system works, I recommend starting with CodeMonkey's tutorial or Unity's own mini-course on the system, depending on your preferred method of learning:

CodeMonkey's tutorial:

Make your Animations DYNAMIC with Animation Rigging! (Unity Tutorial)

Unity's mini-course:

Working with Animation Rigging - Unity Learn

Official Unity documentation:

Animation Rigging | 1.2.0

3.2.2 Unreal's IK solutions:

Unreal Engine has more than one type of rig that can be used for the same purposes. The one that resembles Unity's system the most is the IK Rig. However, a newer system called the Control Rig allows for additional control and is built around the idea of animating within the engine itself. We'll be focusing on Control Rig here, as that seems to be the direction the engine and community tutorials are heading:

Jobutsu's tutorial:
[NEW] Mixamo to UnrealEngine 5 with Full Body IK (IK Retargeter)

Official Epic documentation:
Control Rig in Unreal Engine - check the "Full Body IK" section
Unreal Engine IK Rig - the original IK Rig system

3.3 Animation State Machines:

Animation State Machines are what we use to convert player input and game logic into playing back the appropriate animations. This is where we blend between our animations in all of the different ways (additive, interpolated, with masks etc.) and put restrictions on what can play when. This is very tightly bound with the game design itself, as state machines are also used to dictate what logic can occur when, on a code level.
From a practical standpoint that usually means figuring out the desired interactions first, shooting/animating the most fundamental clips (eg. no link animations) and beginning to couple the animation state machine to the game logic state machine for that character. Once this has been covered, iterating with the game designer, creating needed link animations, tweaking transitions, playback speeds etc. comes into play.

3.3.1 Unity's Animator Controller:

Unity's system for Animation State Machines is called the Animator Controller. For a solid introduction to this system, I highly recommend watching iHeartGameDev's tutorial:

iHeartGameDev's tutorial:
How to Animate Characters in Unity 3D | Animator Explained

Official Unity documentation:
Manual: Animator Controller

Rokoko documentation:
The Ultimate Guide to Character Animation in Unity - step-by-step for importing animations from Studio and working with animation in general.

3.3.2 Unreal Engine's Animation State Machine:

Unreal Engine's Animation State Machine system is easy to integrate with blueprints and works in many ways the same as Unity's Animator Controller. I recommend starting with PrismaticaDev's introduction:

PrismaticaDev's tutorial:
▶ The Anim State Machine | Adv. Anim Application [UE4/UE5]

Official Unreal documentation:
State Machines in Unreal Engine

3.4 Cleanup and editing:

No matter how great the animation tools are, we can't do much if the baseline animations aren't good as well. Thankfully there is a lot we can do to ensure that our animations are indeed good - whether it comes to removing jitter from mocap or correcting poses, it can be done fairly easily (and very importantly, for free).

3.4.1 Blender:

Blender is completely free and extremely robust for doing both cleanup and editing. To properly understand this process, I recommend going through the videos listed here, starting from the top:

3.4.1.1 Cleanup (remove jitter):

The following tutorial covers the basic process for keyframe decimation:

Stephen Scott's tutorial:
▶ Blender 2.8 Tutorial: Decimate Keyframes & Create Ghost Curve

To translate this into the context of mocap, whenever we see jitter in motion capture, it is possible to remove it by averaging the values between the keyframes and removing redundant keys. This will smoothen out the movement, giving a slightly more hand-animated look and removing any jittery motion.

Please be aware that applying this process to an entire animation clip, especially a long one or one with a high framerate, can be very taxing on your computer. If you find this to be an issue, I recommend doing the following:

1. Identify which part of the character is doing the "jitter"
2. Look for whether or not this motion is also occurring further up in the parent chain (eg. if the hand is shaking, note if that's because the arm is shaking).
3. Apply keyframe decimation on the relevant bones only.

Sometimes jitter is accumulative and hard to spot further up the chain, so if it looks like the arm is barely moving but the hand is still jittering after cleanup, try applying the process further up the hierarchy.

3.4.1.2 Pose correction:

If you need to perform a specific pose or accommodate for limbs drifting during motion capture, Blender has a system called non-linear animation that can help you with that:

Sam's tutorial (the Sam himself!):

Easiest workflow for editing mocap in Blender - Rokoko Guide

With non-linear animation strips, we can work additively with the rotations from the motion capture, throughout the timeline. This is perfect for doing pose corrections, but there are a few steps to exporting to fbx and retaining the non-linear animation corrections in the final result. Most importantly, since what Sam is showing here is additive (eg. two animation clips playing together) you will need to bake the final result into a single clip before exporting it. You can find the bake function here, from the NLA window:
Editing — Blender Manual

3.4.2 Maya:

Maya is a paid program, but an industry standard for a lot of studios due to how well it handles character animation and rigging.

3.4.2.1 Cleanup (remove jitter):

Cleaning up mocap in Maya is a pretty smooth process (no pun intended). I recommend checking out this video for a quick rundown of the filter options available:

Maya Learning Channel's tutorial:

Maya 2022 - Graph Editor filters

3.4.2.2 Pose correction:

Once again we have a video by the excellent Sam, covering how to do pose corrections in Maya:

Sam's tutorial:

Editing Rokoko Motion Capture in Maya

And finally you'll need to merge your animation layer with the changes onto the final animation, as the layer otherwise acts as a separate animation in the exported fbx. You can find the merge functionality for the animation layer editor here:
Merge animation layers | Maya 2018 | Autodesk Knowledge Network

4.0 Where to go next:

By now you hopefully have a good feeling for what to keep in mind and how to apply it. However, as we're in a field that is constantly changing, either through tools being updated or new best practices being figured out, it is important that you stay in the mindset of always learning and researching. I don't personally think I've ever done a single project without resorting to Google at least once. **Nobody knows everything and everyone forgets the things they don't always use.**

As such, the best piece of advice I can give you is to be aware of where and how you can find the information you need.

4.1 Unity materials:

iHeartGameDev's Unity Animation Systems playlist/channel:
Unity's Animation System - YouTube - overall great resource for learning Unity's animation systems.

Ketra Games' channel:
Ketra Games - YouTube - a channel with a lot of content on animation-related systems, as well as handling player input and animating the environment.

CodeMonkey's channel:
Code Monkey- YouTube - great indie game developer sharing professional insights about game production.

4.2 Unreal material:

PrismaticaDev's Advanced Animation Application playlist/channel:
Advanced Animation Application - YouTube - super thorough walkthrough of Unreal's animation systems, all in blueprints.

Jobutsu's YouTube channel:
Jobutsu - YouTube - tons of videos on Unreal Engine, updated animation and retargeting workflows for everything from Control Rig to Metahuman.

William Faucher's YouTube channel:
William Faucher - YouTube - a great channel focusing on visual presentation and rendering in Unreal Engine, which is often applicable to real time rendering as well

4.3 Animation and game design:

Video Game Animation Study YouTube channel:
Video Game Animation Study - YouTube - Lots of amazing case studies and breakdowns!

GAME ANIM book by Jonathan Cooper:
Video Game Animation Explained - The book that inspired writing this document in the first place, highly recommended!

The Art of Game Design by Jesse Schell:
Art of Game Design - An absolutely invaluable tool when it comes to asking the right design questions! Anyone remotely interested in game design should read this.